GGPLOT2

Simeon Markind

2017-03-09

Motivation

So far we have seen how to make plots using R's base plotting functions. While base plot does have a lot of maneuverability and versatility there are many plotting packages that help users more easily make professional looking plots in R. The most commonly used package for this is called ggplot2 and is written by Hadley Wickham.

Today we will explore the underlying theory and mechanics of how the ggplot2 package works, stopping frequently to work out examples. The homework for this week will require you to use the ideas touched upon in this lecture to create many different types of graphics and play around with the many syntactical options available in ggplot2 which allow you to customize your graphics to look however you'd like.

The defining feature of ggplot is it's "grammar of graphics," hence the "gg". The package works by creating graphics in layers, each additional customization to a ggplot graphic is a separate layer to the existing plot. What this means in practical terms is that layers of graphics can be plugged in and out with ease. Additionally, because each layer is separate, it is easy to adjust nearly all the features of the graph with just a few lines of code. Now let's get started.

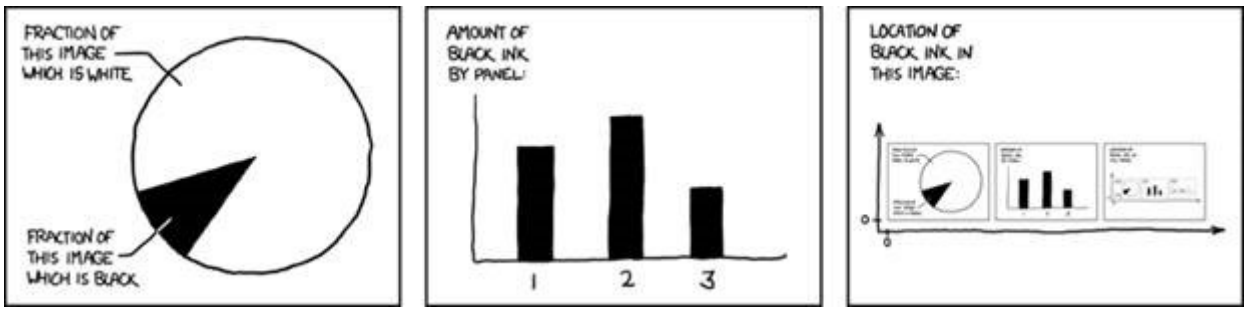


Figure 1: Some basic charts

Where to get help

Ggplot2 is one of the most widely used packages in R. The documentation within R is quite detailed and the documentation outside of R is also robust. Stack Overflow is always a good guess when you run into problems. Additionally, Hadley Wickham maintains documentation at the following website. http://docs.ggplot2.org/current.

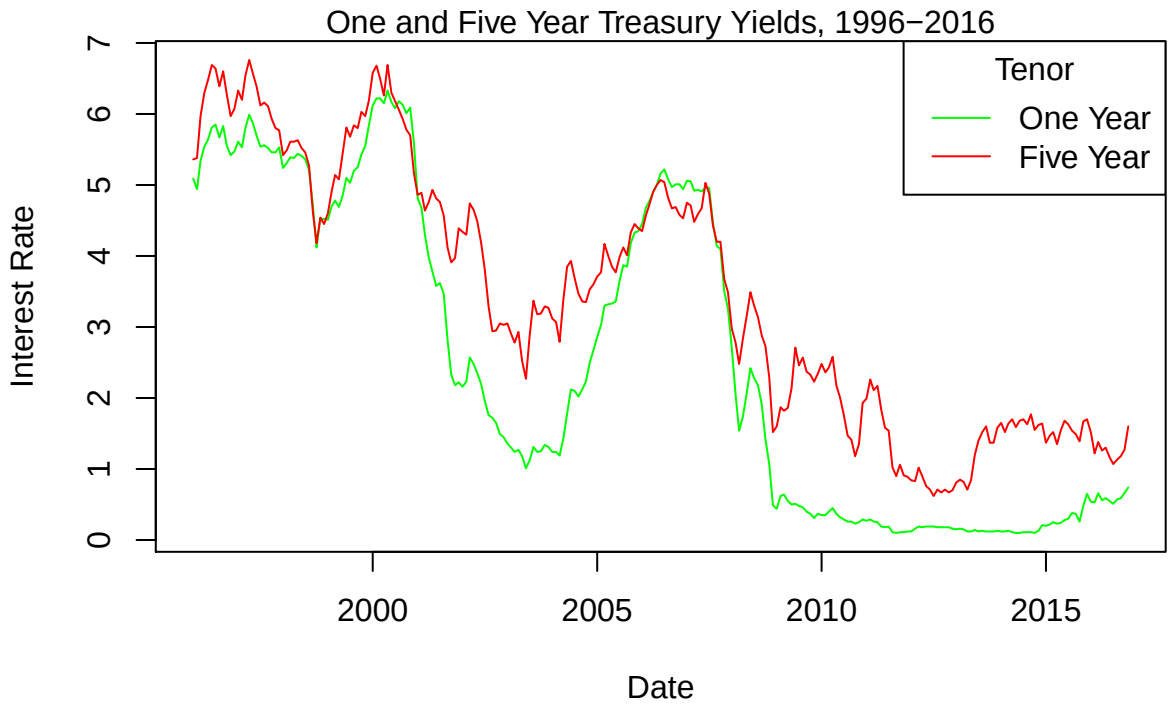
Hadley Wickham also includes two chapters in his free online Data Science Book on ggplot that I encourage you to check out.

Chapter 3: http://r4ds.had.co.nz/data-visualisation.html
Chapter 28: http://r4ds.had.co.nz/graphics-for-communication.html

Introduction

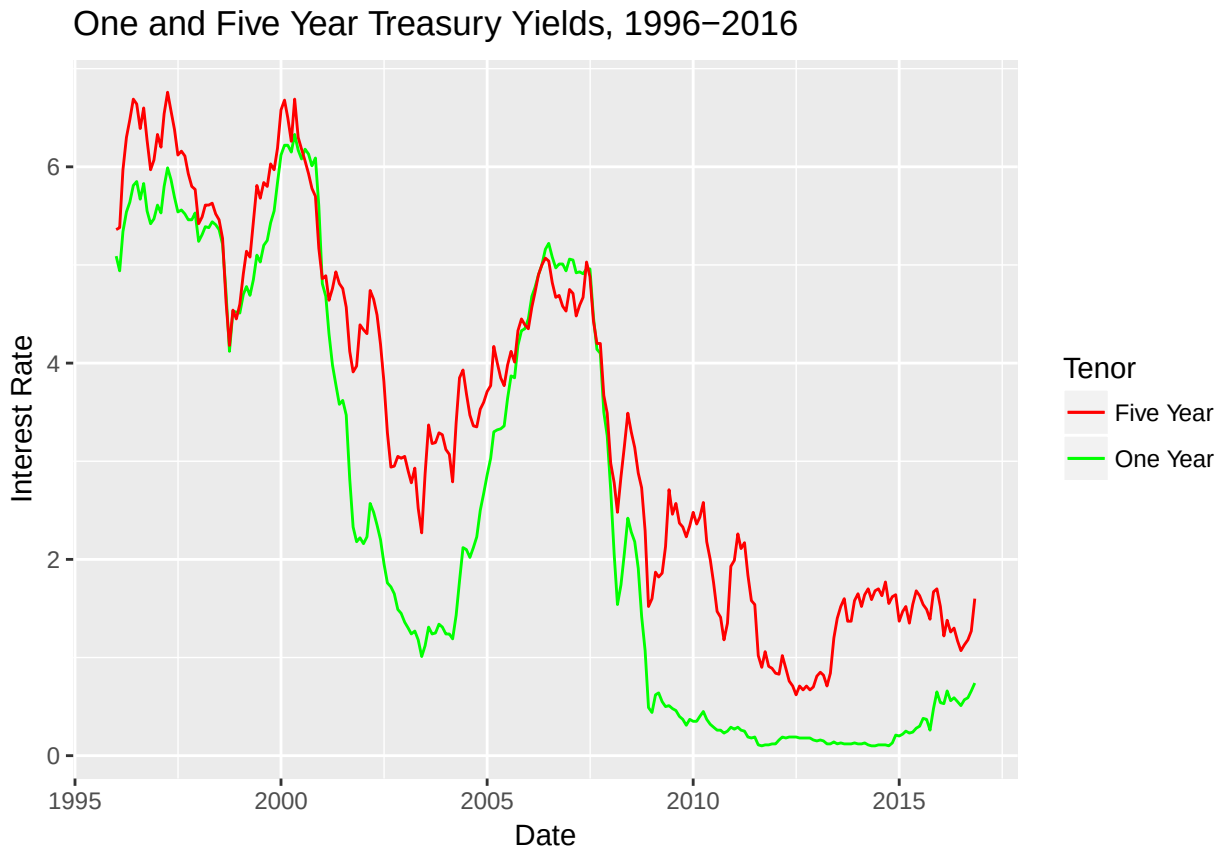
First let's start by creating two graphs of the same information, one coded in base R plot framework and one in the ggplot framework. You should be familiar with the base plot version of this from previous lectures. We will be using the treasuries.csv data we included with the lecture. Let's make a basic lineplot showing the 1 year and 5 year treasury rates over time. Note that I will be using the data.table package throughout this lecture for data filtering and subsetting.

```
## Load your libraries
library(ggplot2)
library(dplyr)
library(readr)
library(purrr)
library(knitr)
theme_set(theme_gray())

data.path <- "/mra/home/m1skm01/Howard_ECOG_314/ggplot2/Data/"
treasuries <- read_csv(paste0(data.path, "treasuries.csv"))

#we will only graph the last 20 years
treasuries <- treasuries %>%
    mutate(DATE = as.Date(DATE, format = "%Y-%m-%d"),
                      GS1 = as.numeric(GS1),
                      GS5 = as.numeric(GS5),
                      GS10 = as.numeric(GS10),
                      GS30 = as.numeric(GS30),
                      UNRATE = as.numeric(UNRATE))

plot.data <- treasuries %>%
    filter(DATE >= as.Date("1996-01-01")) %>%
    select(DATE, GS1, GS5)

## Set up parameters for the basic plot

xaxis <- plot.data %>% select(DATE)
yaxis <- plot.data %>% select(GS1, GS5)

##base plot
plot(x = c(min(xaxis$DATE), max(xaxis$DATE)),
     y = c(min(yaxis), max(yaxis)),
     xlab = "Date",
     ylab = "Interest Rate",
     axes = T,
     type = "n")
lines(x = xaxis$DATE, y = plot.data$GS1, col = "green")
lines(x = xaxis$DATE, y = plot.data$GS5, col = "red")
mtext("One and Five Year Treasury Yields, 1996-2016", side = 3)
legend(x = "topright", horiz = F,
       legend = c("One Year", "Five Year"),
       col = c("green", "red"),
       lty = c(1,1),
       title = "Tenor")
```



This is a fine chart, the code wasn't particularly difficult, though perhaps a bit long. Now we will plot the same data using ggplot:

```
## We already have our data from the first plot we made using base plot
oneFiveTbill <- ggplot(data = plot.data, aes(x = DATE, y = GS1,
                                             color = "One Year")) +
    geom_line() +
    geom_line(aes(x = DATE, y = GS5, colour = "Five Year")) +
    labs(title = "One and Five Year Treasury Yields, 1996-2016",
         x = "Date", y = "Interest Rate") +
    scale_color_manual("Tenor", values =  c("One Year" = "green",
                                    "Five Year" = "red"))

## Now that the plot is created, call it
oneFiveTbill
```



Notice a few things about this code:

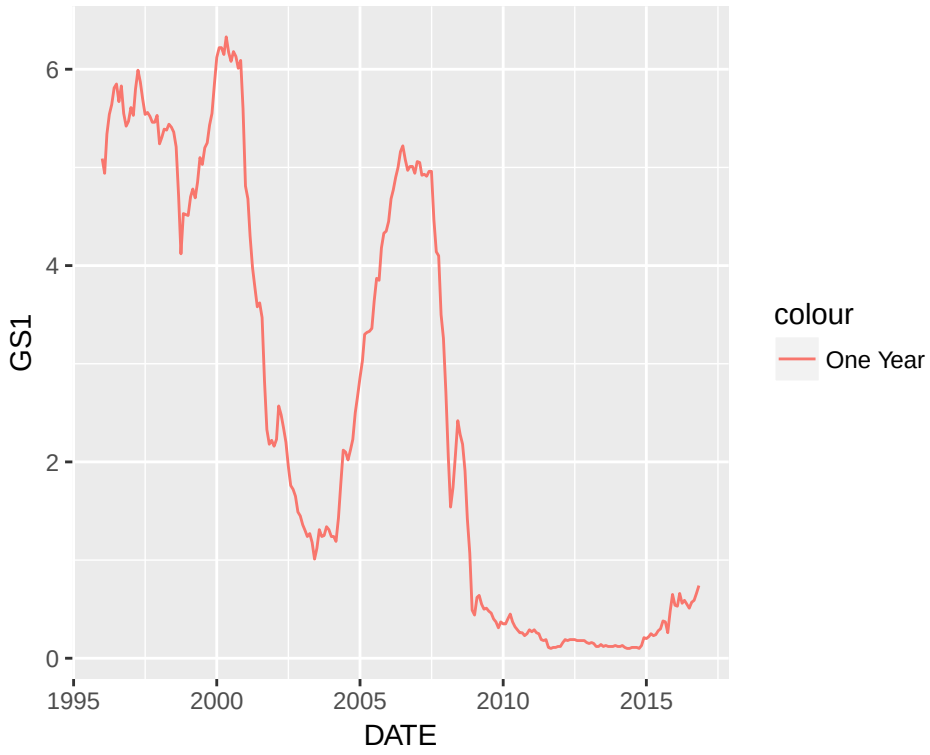
1. Our ggplot is an object. I originally stored it in an object called oneFiveTbill and then called it separately so that the plot would be displayed.
2. It seems that the aes() function is doing the heavy lifting for this plotting, we will discuss this function in greater detail later.

Although our ggplot is displaying the same information as our base plot, it has a more professional look. Looking at the syntax you should start to get a sense that there is a natural flow to the plot commands, i.e. a natural syntax of directions we are telling R to do when making this plot, each one connected via the + operator.

Layering

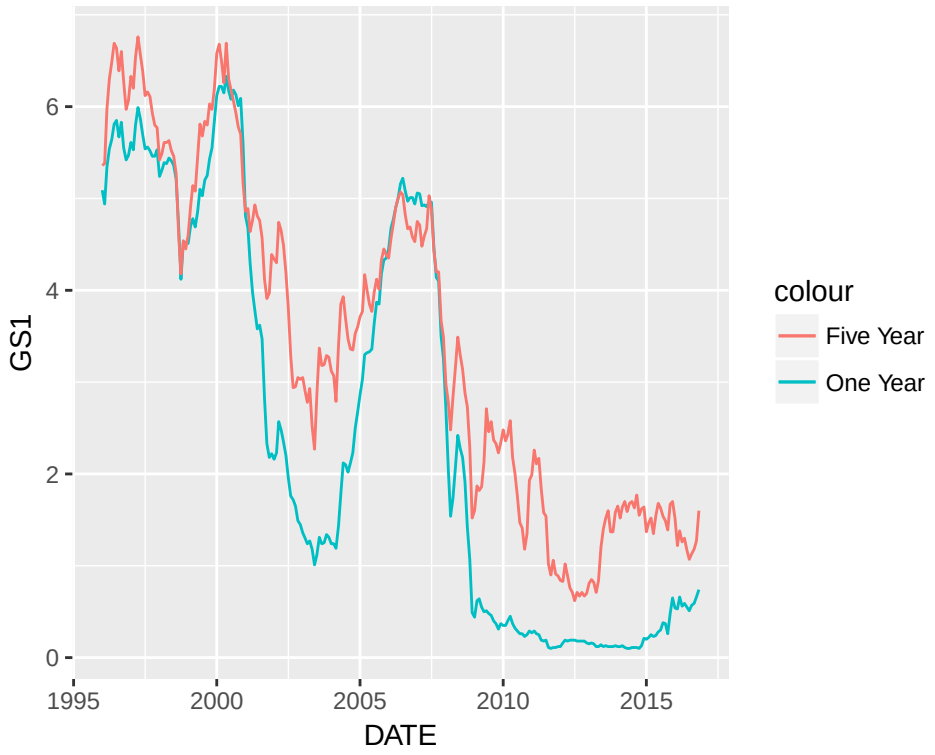
As mentioned before, the most important concept of using ggplot is the understanding that graphics work as layers atop of one another. To see this feature, let's make the same chart but this time encapsulate the code in chunks.

```
firstLayer <- ggplot(data = plot.data,
                     aes(x = DATE, y = GS1, color = "One Year"))+
    geom_line()

firstLayer
```

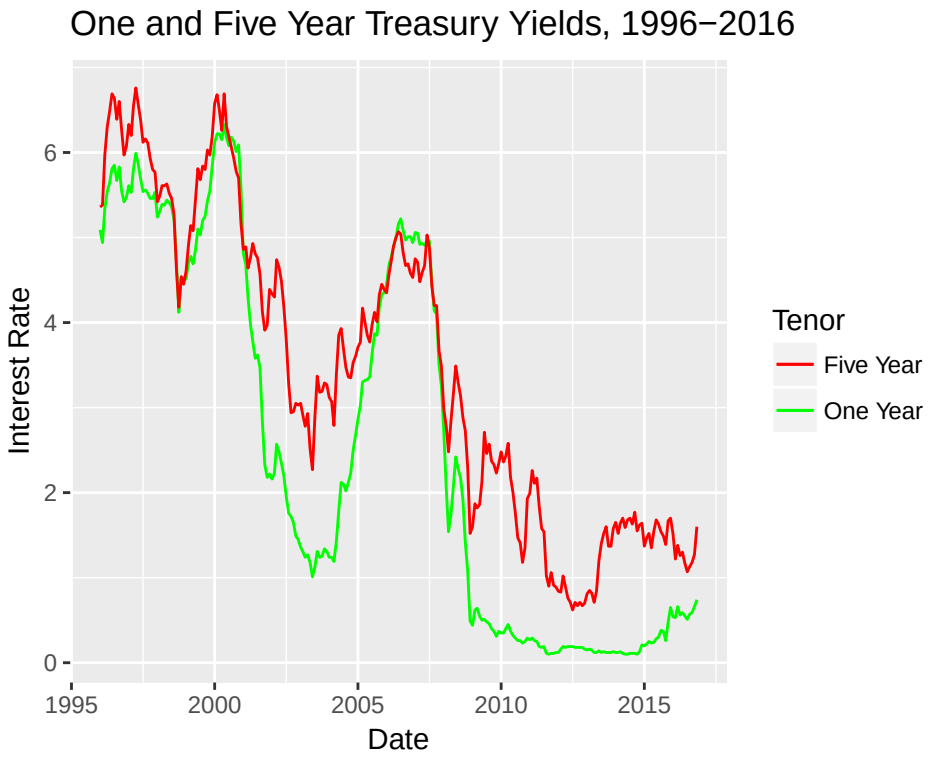


So as you can see, our first layer is, by itself a valid, if spartan, plot. Now let's keep adding.

```
secondLayer <- firstLayer +
    geom_line(aes(y = GS5, color = "Five Year"))
## We do not need to pass in an argument for the data to use or the
## x axis since we already assigned those in first.layer
## If we wanted to use a different dataset we would need to specify
## that directly in the geom_line command.

secondLayer
```




```
## Now that we plotted our data we should appropriately label our axes and title

thirdLayer <- secondLayer +
    labs(title = "One and Five Year Treasury Yields, 1996-2016",
          x = "Date", y = "Interest Rate") +
    scale_color_manual("Tenor",
                       values =  c("One Year" = "green",
                                   "Five Year" = "red"))
thirdLayer
```



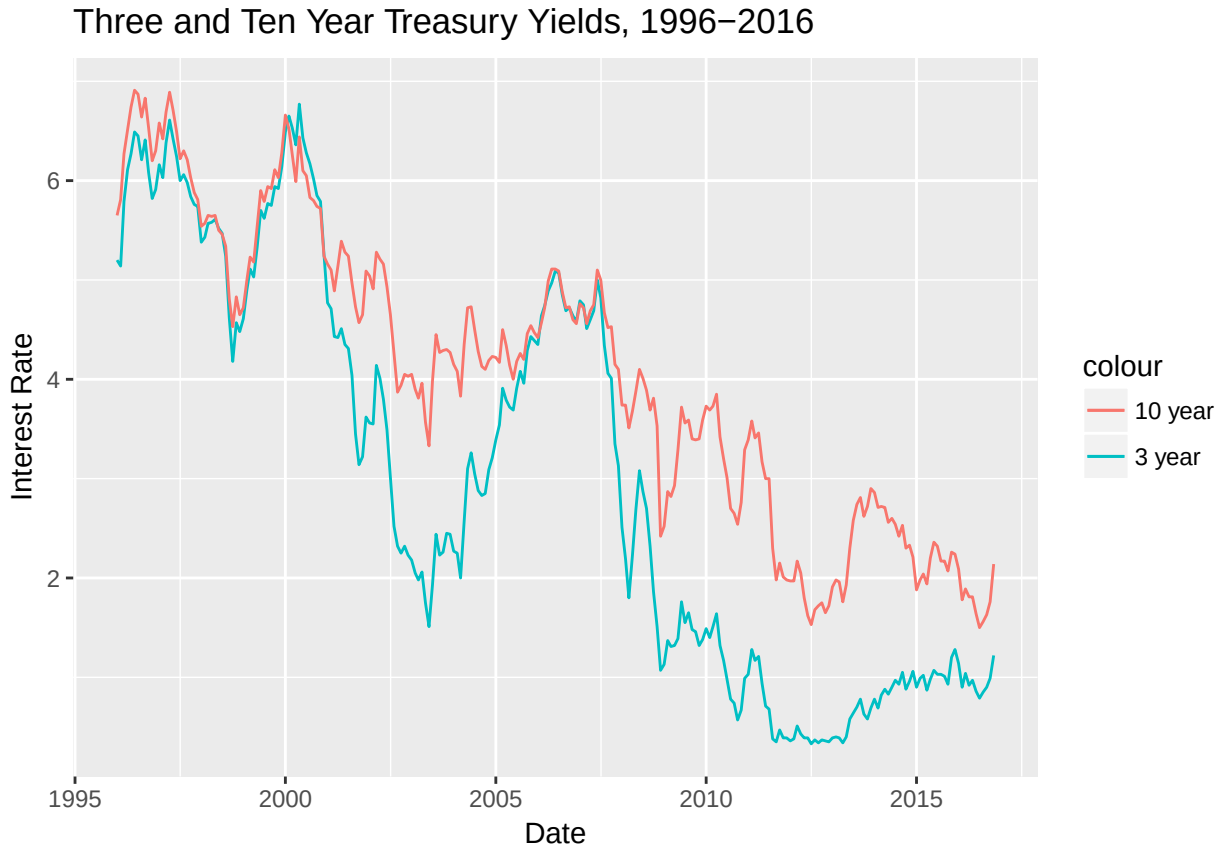
Our layers add up to the original plot. As you may have noticed, the scale_color_manual command allows me to set the name of the color guide as well as change the colors of the lines. Here we assign observations with a color value of "One Year" a non-valid color name, to instead have a color value of "green," a valid color. Note that in secondLayer, before we explicitly defined the color values, ggplot used the default color values. We will discuss scales later in this lecture.

Try it on your own, using the above code as guidance for the next five minutes make the following plots:

1) for the first layer - plot the 3 year treasury bill returns, "GS3" in the treasuries.csv file over time for the last 20 years.

2) for the second layer - add a plot of the 10 year treasury bill returns "GS10" in the treasuries.csv file for the same time period as the GS3 chart. Be sure to specify color arguments so that each line looks different.

Call your output inClass1. Your plot should look something like this:



So now that we understand the idea that ggplot works by layering objects you are probably wondering:
1. What is that "aes" function that he keeps using?
2. How come we can pass in geom_line with no arguments but also with arguments? That makes no sense.
3. Why are treasury yields so damn low?

I'll start by answering the last question. In a nutshell, the recession and slow growth of the past ten years pushed returns down across the board. Since overall growth is low, people are willing to take lower interest rates since they doubt they would be able to do better if they instead put their money elsewhere, like the stock market. For example, if you open a savings account you will probably notice that your interest rate is still below 1%, this means that the bank pays you very little for the ability to lend your money. From the banks perspective, money is "cheap", i.e. it costs the bank little to buy more money. US Government treasuries are considered to be some of the safest investment in the world. As a result the US government is easily able to raise money by selling bonds. Since interest rates are low and returns in other investments are low and uncertain, the US government does not have to offer high interest rates in order to attract investment. As you can see from the charts we made, this was not always the case, in 2004 to 2007, as the economy grew the interest rates on the treasuries was higher, in times of stronger growth the government has to offer more enticing interest rates on the bonds in order to attract customers.

To answer the other two questions we will start by looking at how ggplot2 works under the hood.

Basics of the Grammar of Graphics

We are going to go step by step through how ggplot creates a graphic. To do this it will be easier if we use a long dataset instead of the wide one we currently have.

Who here can describe the difference between a long dataset and a wide dataset?

```
plot.data <- treasuries %>%
    filter(DATE >= as.Date("2016-01-01")) %>%
    select(DATE, GS1, GS10, GS5, GS3, GS30)

plot.data <- melt(plot.data, measure.vars = c("GS1", "GS10", "GS5", "GS3", "GS30"))
```

ggplot works by mapping data to aesthetics. This is the `aes` argument that you have seen passed into the `ggplot` and `geom_line`. In order to create a ggplot object you must first call the `ggplot` function. This function takes the following arguments: `ggplot(data = NULL, mapping = aes(),...)` The data argument is the data.frame or data.table you have of the data you want displayed. In our case we conveniently called this data "plot.data" but you could call it whatever you want. **It is a good idea to always give your datasets descriptive names such as "plot.data" and not something meaningless like "x," this way you will be better able to follow your code when you go back to re-read it.**

Before we can plot the data we have to understand its format. Let's take a look at plot.data:

DATE	variable	value
2016-01-01	GS1	0.54
2016-02-01	GS1	0.53
2016-03-01	GS1	0.66
2016-04-01	GS1	0.56
2016-05-01	GS1	0.59
2016-06-01	GS1	0.55

This is what ggplot first sees when we put in `ggplot(data = plot.data,...)` but what does that mean in terms of plotting?

Scatter plot

Take a minute to think about what we need to know about data in order to make a scatter plot?

Ggplot needs the following information about each point in order to plot it:
1. x value
2. y value
3. marker (shape) type
4. color type
5. size

So for the data we just saw, what ggplot needs is a dataset that looks like the following:

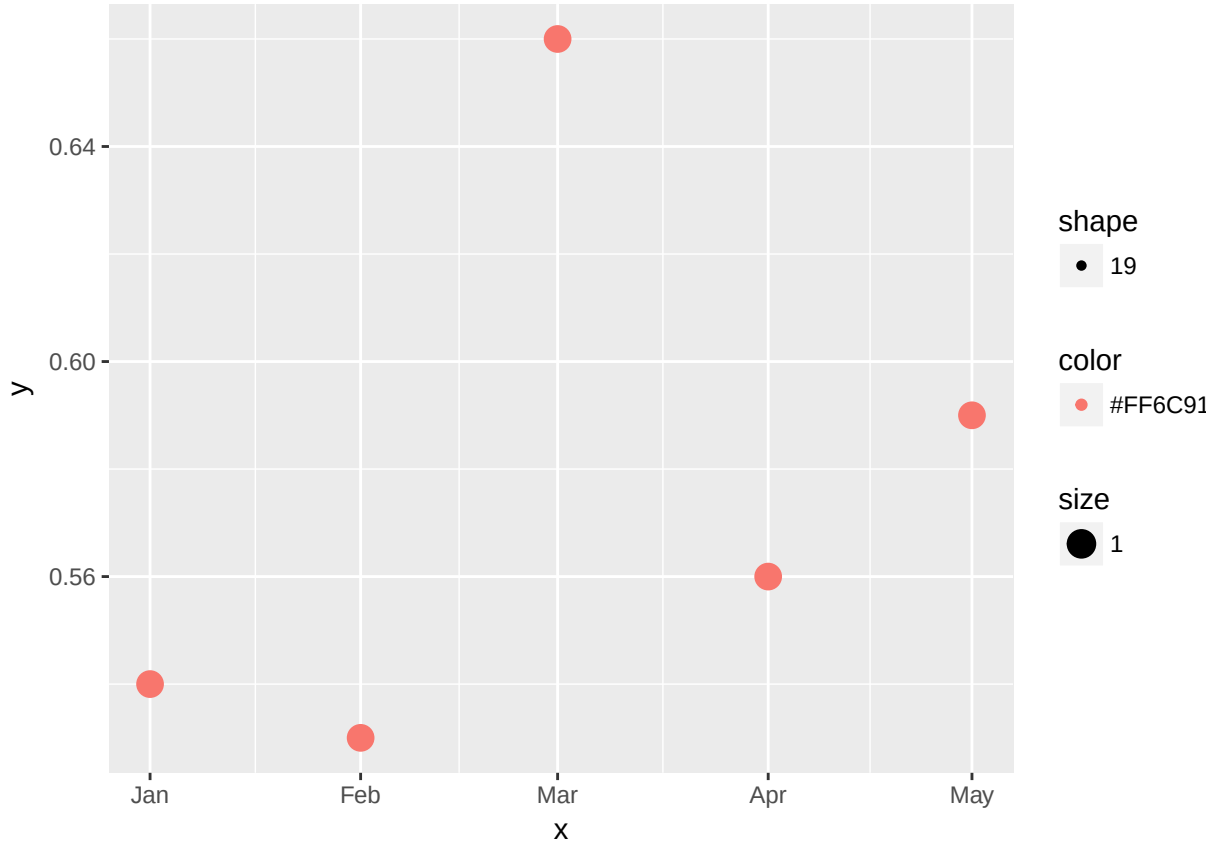
x	y	color	shape	size
2016-01-01	0.54	GS1	19	1
2016-02-01	0.53	GS1	19	1
2016-03-01	0.66	GS1	19	1
2016-04-01	0.56	GS1	19	1
2016-05-01	0.59	GS1	19	1

Ok, that's not quite it, for the shape, ggplot uses the default values of a circle, number 19 - unless otherwise specified - and we have the default size of 1. Right now we are giving R a non-sensical color, "GS1", instead we need to give R a color expressed in hexidecimal. R can also interpret simple color names such as "red" or "green." Changing "GS1" to a color R can interpret, we pass the following table to R:

x	y	color	shape	size
2016-01-01	0.54	#FF6C91	19	1
2016-02-01	0.53	#FF6C91	19	1
2016-03-01	0.66	#FF6C91	19	1
2016-04-01	0.56	#FF6C91	19	1
2016-05-01	0.59	#FF6C91	19	1

Here #FF6C91 corresponds to a red color. Now let's pass this table to a ggplot object:

```
basic.scatter <- ggplot(test2, aes(x = x,
                                   y = y,
                                   color = color,
                                   shape = shape,
                                   size = size)) +
    geom_point()

basic.scatter
```



This is the basic scatterplot that we pass to ggplot. Other additions, such as axis labels, tick marks, font size, titles, background color, etc... are all controlled and added after this basic plot is made. The reason we have tick marks, values, and titles on the axes is that ggplot used the defaults, such as labelling the x axis the name of the column of data we passed in to the x parameter of the aes() call.

Luckily for us we do not have to specify color values in hexidecimal or shape or size values, ggplot can automatically provide valid values for these arguments. This is why in the previous plots that we made, we only passed in x and y values but were still able to get valid charts out.

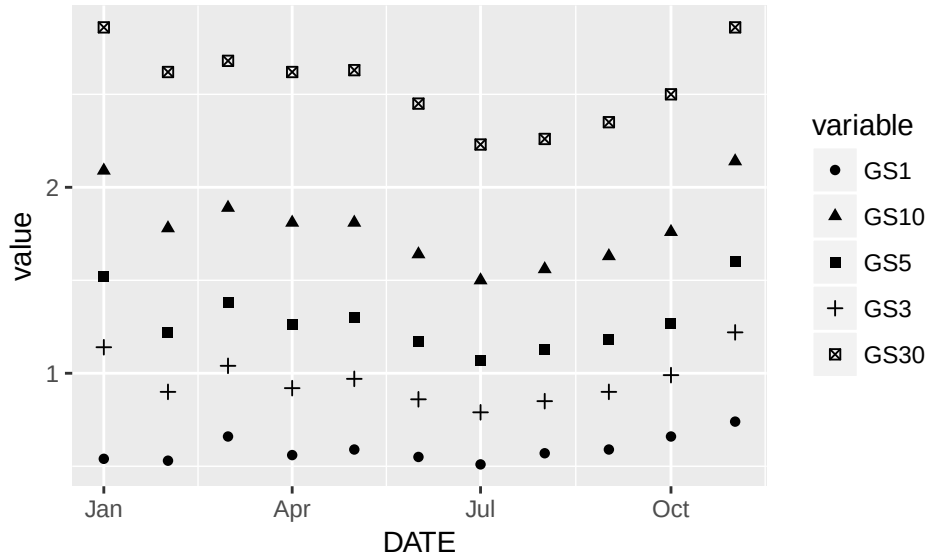
So to summarize, ggplot takes data, in the form of a data.frame or a data.table, specified by the data =

argument, and maps it to graphical output using the mapping = aes() argument. The aes() argument, short for "aesthetic" takes in all the different mapping arguments used for the plot type.

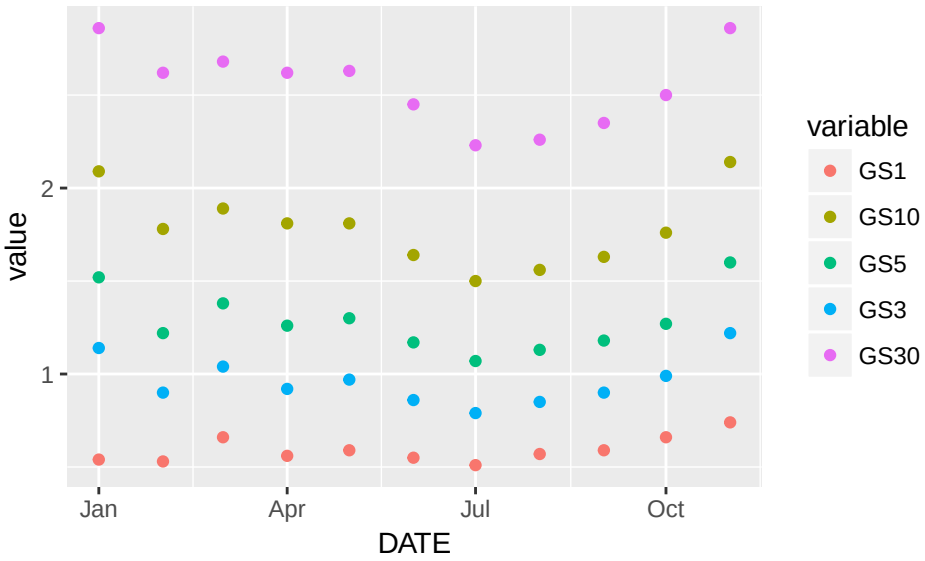
Going back to question 2 from above about why we can pass in geom_line() both with and without arguments it should now make a little more sense after seeing this call to geom_point(). When we call geom_point() with no arguments by default it uses the arguments given to the aes() function in the ggplot call. If instead we specify geom_point(aes(color = "red")) what we are doing is specifying an argument that was not made explicit in the aes() call for the ggplot function. Alternatively, we can specify an aesthetic in the geom_point() call that was given in the ggplot call. This creates an override for that particular geom_point() layer.

But just calling `ggplot` and passing in data does not create a plot. Once we have the data prepared in the ggplot object we now have to call the layer of chart that we want. In this case we are looking for a scatter plot so we call geom_point. If we wanted a line plot we could simply call geom_line. These geom_ arguments also take inputs and can also take the aes() function. Recall in the first example of a line plot we passed in the aes function to geom_line() for the second line of treasuries. Now let's make a scatter plot of all the different treasuries for 2016 and change the shape based on the length of the bond. Since we want each different type of treasury, as defined by the "variable" column of our data, to be treated as its own group, we must also specify the "group =" argument in the aes() call.

```
scatter <- ggplot(plot.data, aes(x = DATE, y = value, group = variable))

scatter + geom_point(aes(shape = variable))
```



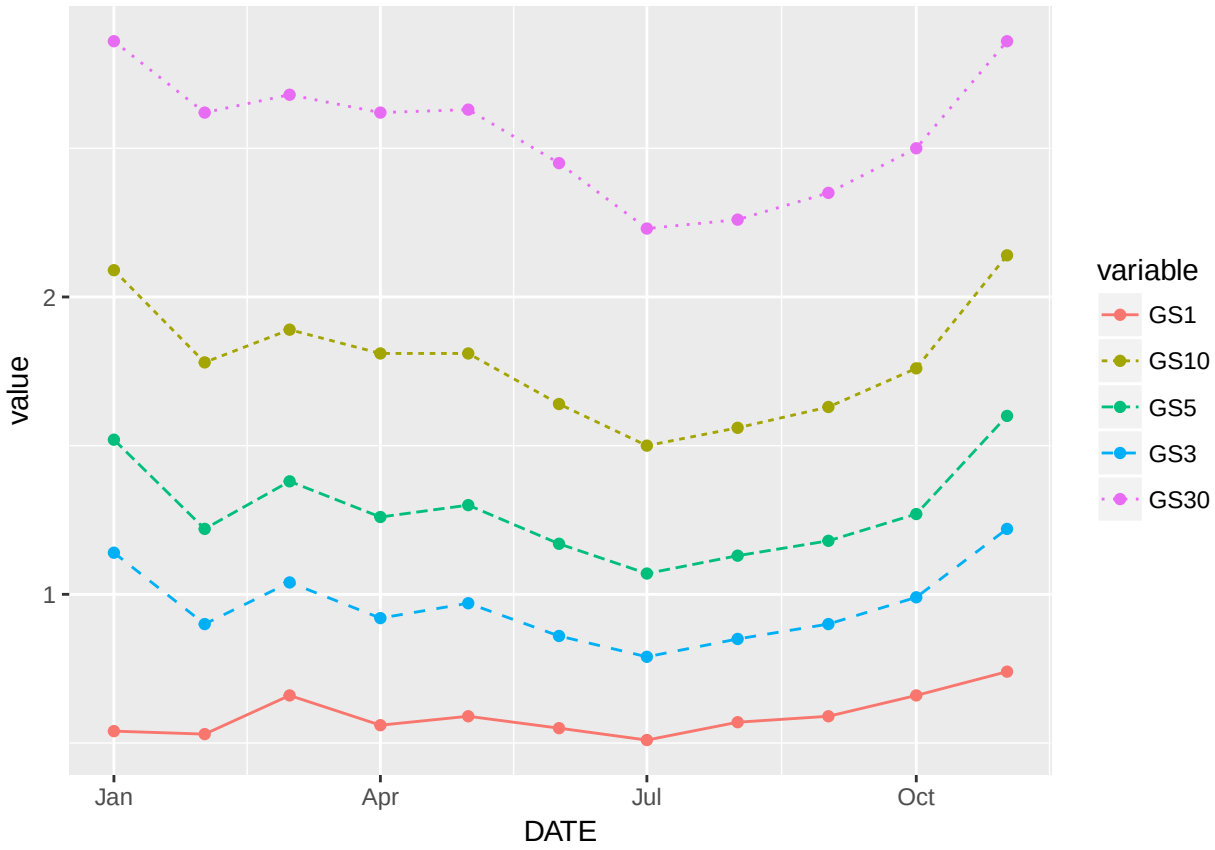

```
## Does this graph look good? What could we do instead to improve it? Is changing shape for each group

scatter + geom_point(aes(color = variable))
```



Now while the axes are quite ugly, you can see from the above that by passing the aes() function to the geom_point() calls we were able to control the shape or color. Had we used `scatter + geom_point(aes(color = variable, shape = variable))` both the color and shape of the point would change for each different bill type as shown on the legend.

Now that you have seen how ggplots are made and understand some of the basics of how the layering works, use the data table we have already made for 2016 and create a plot with points for each observation, with different colors for each point. Additionally, add lines between the points and vary the line type and line color for each treasury bill type. try ?geom_line to look at the arguments for line charts and use the "linetype" argument for the aes function in the geom_line call to control how the linetype changes. You will also need to specify "group = variable" in the aes call of you ggplot function.

Call your output inClass2. Your chart should look like this:



Although the axes for this chart are not well labeled - we will cover that later - you should notice a particular trend here. All of the lines are distinct, for the time period we have chosen, 2016, none of the treasury securities cross each other's path in terms of the rate at which they yield. Also, as the tenor of the security increases, the rate at which it pays out also seems to increase, (how would you test this hypothesis?). But let's think about why that could be:

Buying a bond, which these government securities are, means exchanging money now for a lump sum payment in the future and regularly scheduled interest payments along the way. In a sense, the rate at which the bond is trading is a forecast for what the expectation of inflation, and economic growth, until that point will be. (Since the interest payments need to offset the impact of inflation on the value of the security in order to entice people to buy it.) So looking at this chart we can see that in October of 2016 people expected average inflation of roughly 0.66% over a single year and long-term average inflation of 2.50% over 30 years. It seems that November was a good month for treasury yields as the securities longer than one year all showed steeper positive slopes than before. This makes sense as November 2016 saw the stock market surge to record high closings.

Scales

As you've seen from the last few graphs we made, putting the data into the plot is only part of the work. Adjusting the axes to be both aesthetically pleasing and informative is another important part of making any plot. With the plot we just made we will now look at how to control the axes in ggplot.

What Scales Do

At a basic level scales take data and turn it into something that can be visually perceived. In ggplot scales control the size, color, shape, position, linetype, etc... Notice that in our previous plots the variables plotted that were not measured on the x and y axes were described in the legend. In ggplot scales are, unsurprisingly controlled by the family of functions beginning with "scale_". For example there are scale_x_NNN commands for controlling the x axis, scale_color_NNN commands for controlling the color scale, scale_size_NNN for controlling the size scale, etc... In addition to giving the user greater control over the scale values, the scale commands are also used to help make the guide or legend that is displayed alongside the chart. Let's take a look at the different scaling options for the x and y axes.

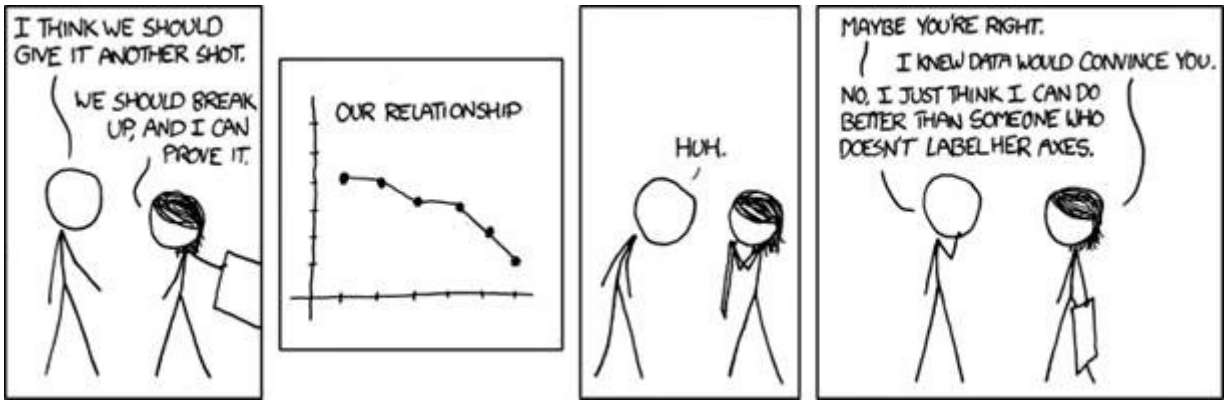


Figure 2: Always label your axes

scale_continuous

The `scale_x_continuous` and `scale_y_continuous` are the most common types of scales used for the x and y axes. Unsurprisingly these scale types are for use when each variable is for continuous data. (e.g. numbers). Let's take a look at the documentation for scale_x_continous.

```
scale_x_continuous(name = waiver(), breaks = waiver(),    minor_breaks = waiver(), labels
= waiver(), limits = NULL,  expand = waiver(), oob = censor, na.value = NA_real_,  trans =
"identity", position = "bottom", sec.axis = waiver())
```

Note that as we saw before, this command does not have to be specified in the ggplot command in order for the chart to be made, ggplot will try to figure out what is best based on the data if scale commands are not made explicit.

Let's step through the arguments:

`name` allows you to pass in a character string for what you want the axis to be labeled. i.e. for a plot of age vs. height we would say `name = "height"`.

The `breaks` command controls where the main tick marks come through on the axis. For the last graph we made we see the breaks at "Jan", "Apr", "Jul", "Oct". If we want to change where those breaks are we can pass in a vector to the `breaks` argument. `minor_breaks` similarly controls where the secondary tick marks appear.

The `labels` argument allows you to pass in a character vector of the same length as the `breaks` argument which controls what label will appear at each break. I.e. for our example of age vs height we could pass in breaks of `c(0, 10, 20, 30, 40, 50)` but we would want each break to be labeled as `c("0 years", "10 years", "20 years", "30 years", "40 years", "50 years")`.

The `limits` command allows you to set the minimum and maximum value for the scale. (This will limit the data that gets plotted). The `limits` argument takes a vector of length two, `c(min, max)` or `NA` which will use the default.

The `trans` argument describes a transformation we can use on the data. Say instead of graphing age vs. height, we wanted to graph age vs. log(height) on the y-axis. We could call the `scale_y_continuous` function with the `trans = "log"` argument and get the transformation we are looking for. (Note that some of these transformations have shortcuts. For example, scale_y_log10 has the transformation already built in.)

The `position` argument controls where the axis is located. i.e. "left" or "right" for vertical scales and "top" or "bottom" for horizontal scales.

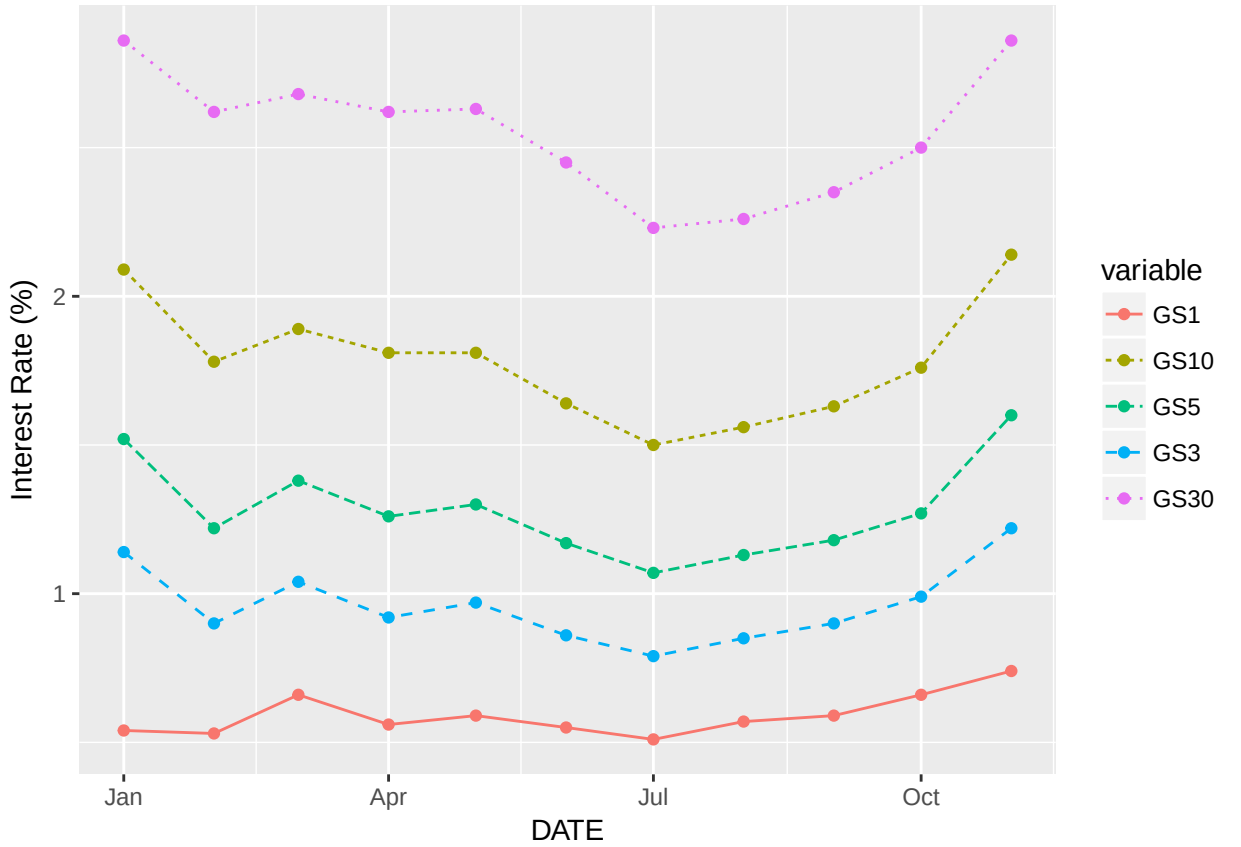
We will not go over the other arguments for scale_continous in class but they are worth familiarizing yourself with as they can be very useful.

What are some other useful scale types you can think of for x and y axes?

Try the following: `?scale_x_date`
`?discrete_scale`
`?scale_x_reverse`

Now let's go back to the last plot we made and change the vertical scale to better fit the data, we will call this plot inClass3:

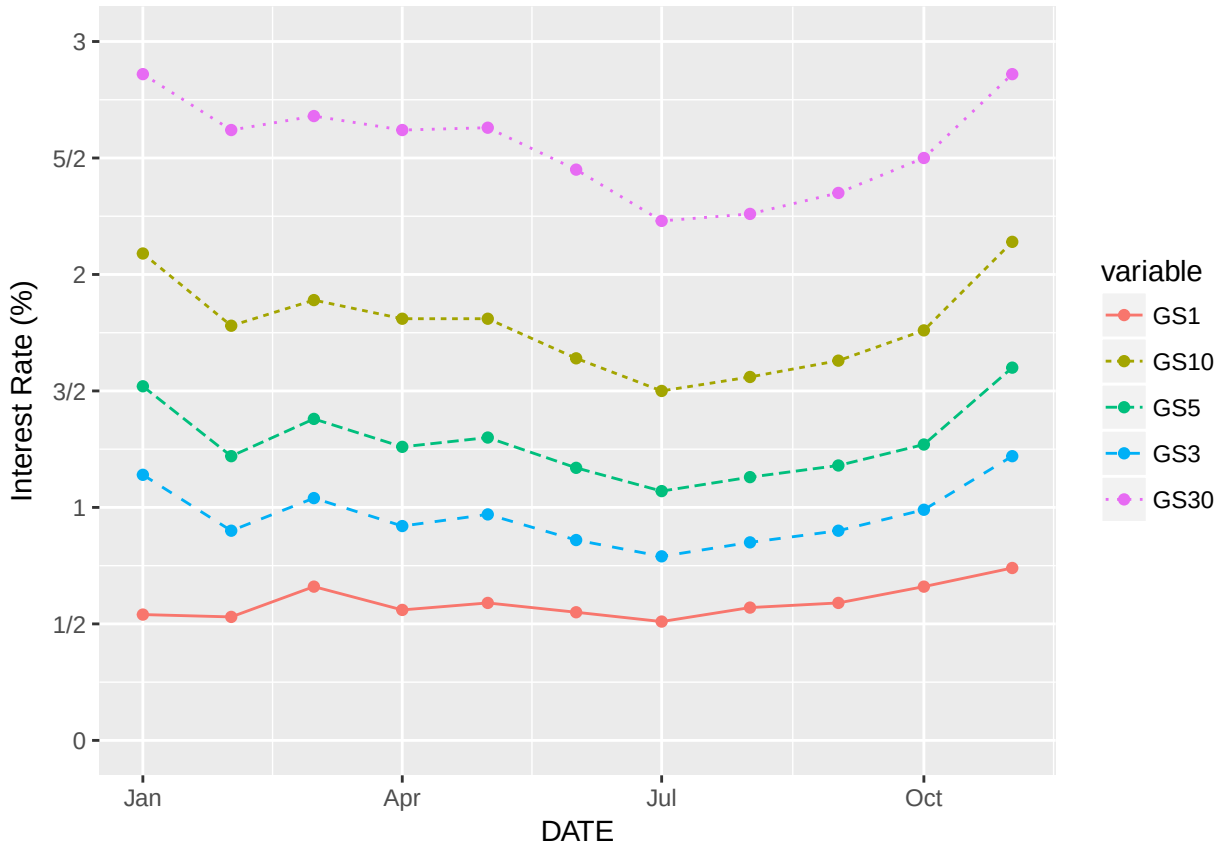
```
inClass3 <- inClass2 +
    scale_y_continuous(name = "Interest Rate (%)")

inClass3
```



Hmm, well that's not good. When we run `typeof(plot.data$value)` we get double which is not a continuous variable type. So now we have to go back to our original data and change it to be continous before we can

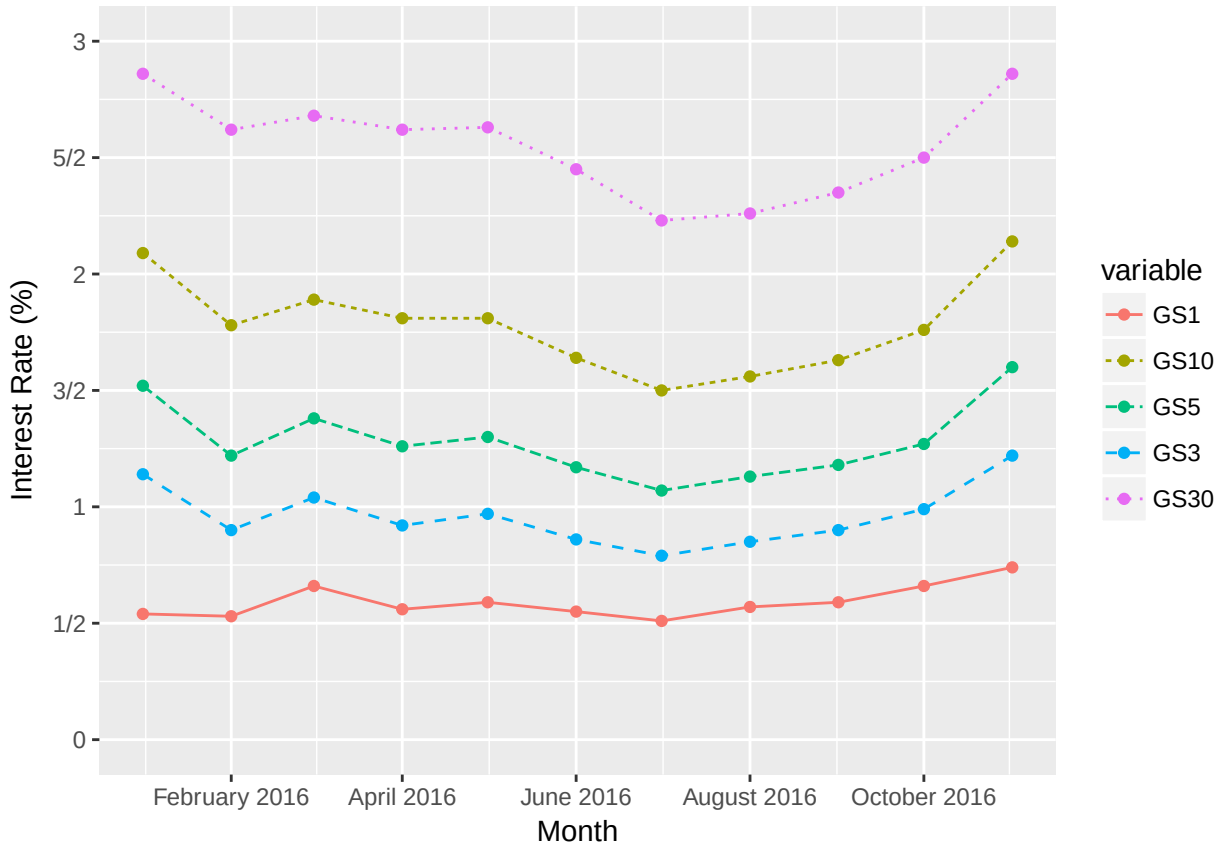
use the continous scale type.

```
plot.data <- plot.data %>% mutate(value = as.numeric(value))

inClass3 <- ggplot(plot.data, aes(x = DATE, y = value, group = variable)) +
    geom_line(aes(linetype = variable, color = variable)) +
    geom_point(aes(color = variable)) +
    scale_y_continuous(name = "Interest Rate (%)",
                       breaks = seq(from = 0, to = 3, by = 0.5),
                       labels = c("0", "1/2", "1", "3/2", "2", "5/2", "3"),
                       limits = c(0,3))

inClass3
```



Don't forget about setting the limits! The `breaks` command does not override the default limits, so if you set a `breaks` command with values outside of the default limits, those breaks will not be plotted unless the limits argument is supplied as well. Also, I would recommend against using fractions as your scale labels; in this case I merely did this to provide an example of the `labels` argument.

Now take a look at changing the x scale for the chart scale_y_advanced. Since this scale shows dates, use the scale_x_date command. Be sure to read the help documentation for the function arguments, especially the date_breaks and date_labels arguments. Try to change the date formatting to be every two months and give the full month name and year. Call your output inClass4. Your chart should look like this:



Other Scales

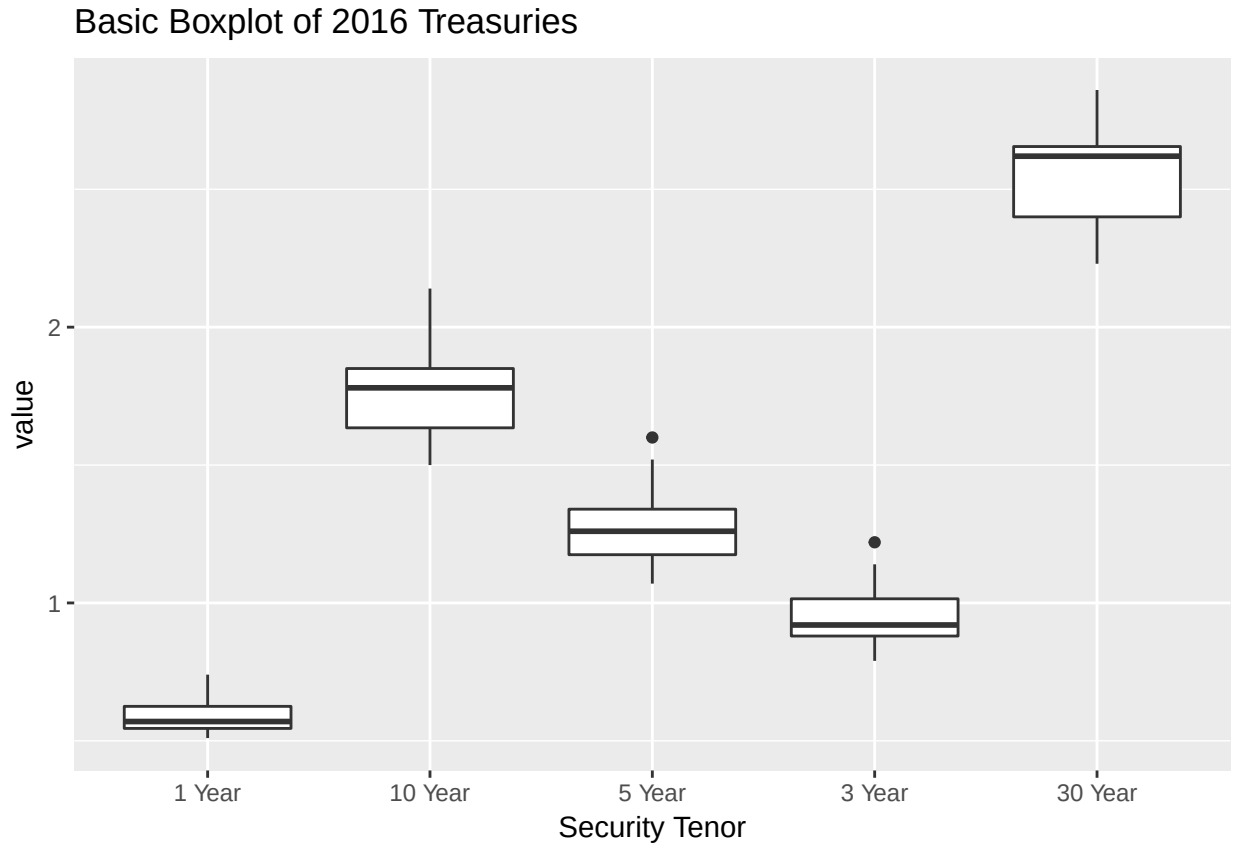
Every aesthetic that is passed in to ggplot can have a scale. Think back to the table we passed in to get our first scatter plot:

x	y	color	shape	size
2016-01-01	0.54	#FF6C91	19	1
2016-02-01	0.53	#FF6C91	19	1
2016-03-01	0.66	#FF6C91	19	1
2016-04-01	0.56	#FF6C91	19	1
2016-05-01	0.59	#FF6C91	19	1

We just spent a long time looking in depth at scaling for continuous variables. Unsurprisingly we can also control scaling for discrete variables. (i.e. "binned" values). This type of scaling doesn't make sense for a line chart, but if you wanted to plot boxplots, which measure distribution for a variable, for each security type next to each other we would want a discrete scale. In fact, let's do that now.

```
basic.boxplot <- ggplot(plot.data, aes(x = variable, y = value)) +
    geom_boxplot() +
    scale_x_discrete(name = "Security Tenor",
                     labels = c("GS1" = "1 Year",
                                "GS3" = "3 Year",
                                "GS5" = "5 Year",
                                "GS10" = "10 Year",
                                "GS30" = "30 Year")) +
```

```
    ggtitle("Basic Boxplot of 2016 Treasuries")

basic.boxplot
```



Notice that in this chart, for the discrete labels we could directly specify what we wanted each bucket to be called. Additionally, note that if we wanted the data sorted by tenor we would have to either sort the data before plotting, or use a numeric value instead of character so that R would know that "GS5" is greater than "GS3" since now the character strings are essentially arbitrary, which is why we get 1 year then 10 year in the above plot.

We already saw that we can change the scaling for the x and y columns. We can also change it for the color, shape, and size columns. In the case of this table, all of the color values are the same, the shape values the same, and the size values the same, so there would be no point in setting scales manually and creating guides. However, in the above line chart some of these columns do differ and therefore setting up a scale manually is appropriate.

Take a look at the line chart, inClass4, we just made for the 2016 treasuries. What other type of scaling options could we control?

Scale Color

Like the position scales (x and y), the other aesthetic scales also have a range of options allowing users pinpoint control. For a full list of valid color names in R refer to Rcolor.

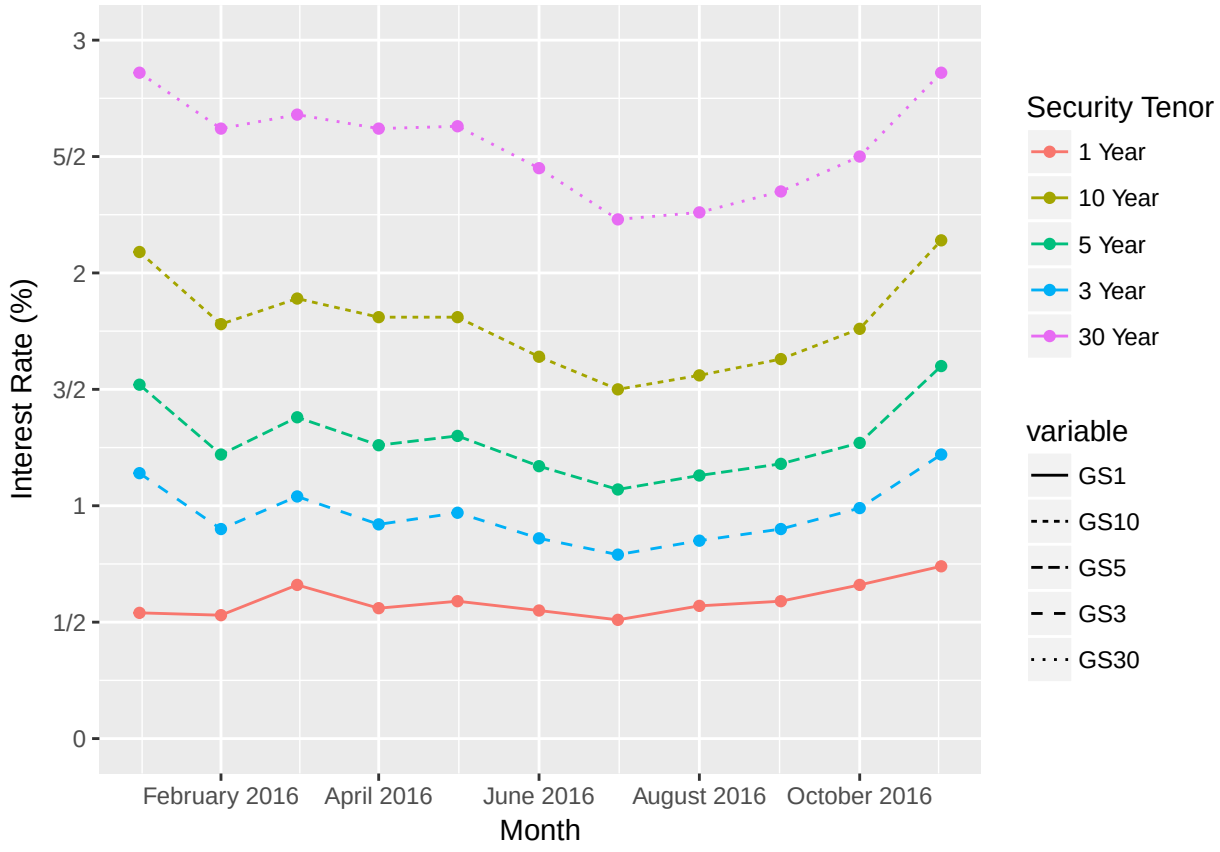
For example, the color scale can be discrete or continuous as well. For continous color scale, we use `scale_color_gradient` or another function in that family. Try `?scale_color_gradient` to see the other continuous options for the color scale. For our line chart above the colors correspond to discrete security tenors, so we would want to use the discrete scale command for color. This command is `scale_color_hue`.

If you type `?scale_color_hue` you'll notice that there are not very many arguments, in fact we have not seen any of these arguments before. `scale_color_hue(..., h = c(0, 360) + 15, c = 100, l = 65, h.start = 0, direction = 1, na.value = "grey50")`

The `h, c, and l` arguments control the hues, chroma, and luminance of the colors respectively. We do not have to worry about that. Most of what we will want to change will actually be passed in through the `...` argument which passes argument to the `discrete_scale` command. Look at the help for `discrete_scale` if you are interested in what this command does. For our purposes all we need to know is that the `...` argument in `scale_color_hue` allows us to pass in options we are already familiar with such as `name =` and `labels =`

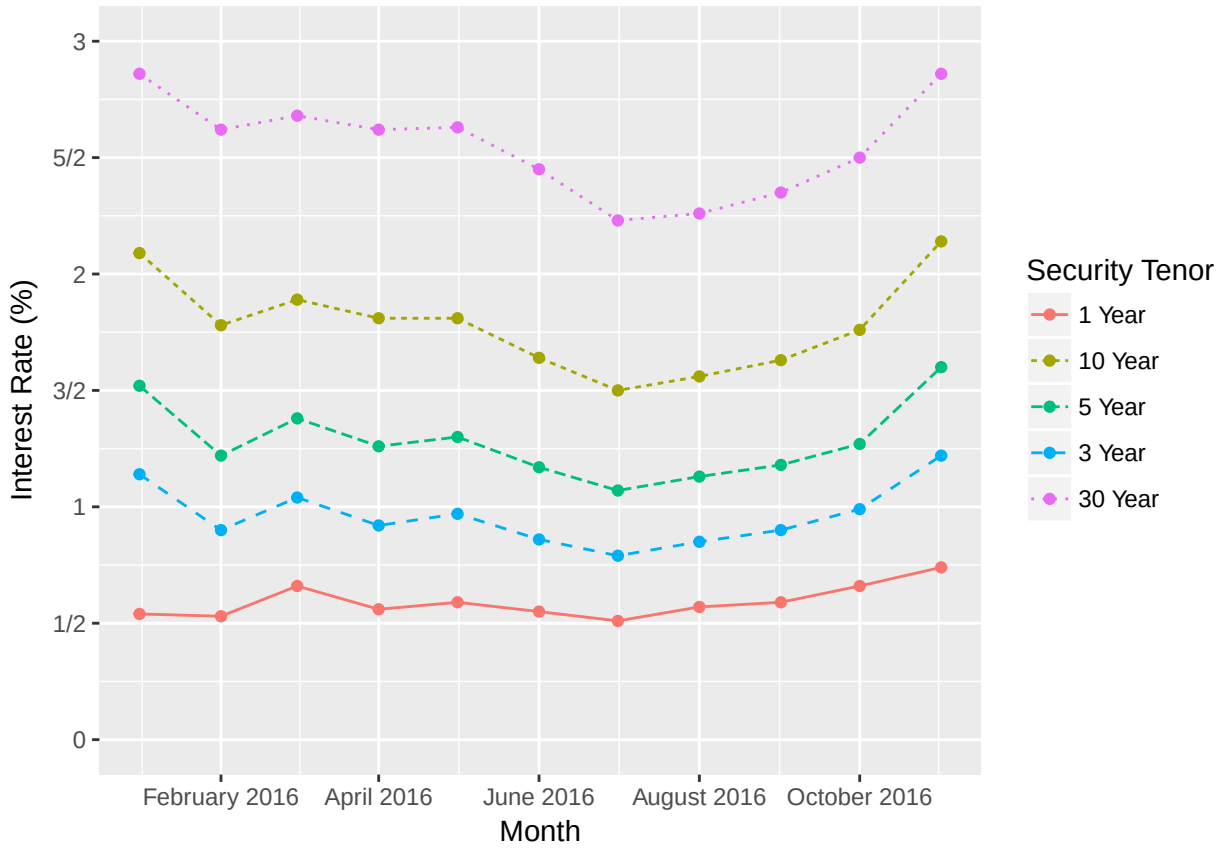
Let's use the scale_color_hue command to clean up our color scale, we will call this plot inClass5.

```
inClass5 <- inClass4 +
            scale_color_hue(name = "Security Tenor",
                    labels = c("GS1" = "1 Year",
                               "GS3" = "3 Year",
                               "GS5" = "5 Year",
                               "GS10" = "10 Year",
                               "GS30" = "30 Year"))
inClass5
```



Notice now that the color scale shows the different colors corresponding to the tenors as we wanted, but before we only had one legend while now we have two. What is the other aesthetic that we mapped the the "variable" column?

Use the appropriate `scale_linetype_` command to adjust the linetype scale accordingly. Assign your output to inClass6. Your chart should look like this:



Hint - notice that if we pass the same arguments for name and labels to the linetype scale as we did the color scale we go back to having a unified guide

Similarly to our ability to control the color and linetype scales we can also control the size, shape, etc... scales. Start typing `?scale_size` into your console and you will see all the different built-in options for the size scale. `?scale_shape` will return similar results for the shape scale.

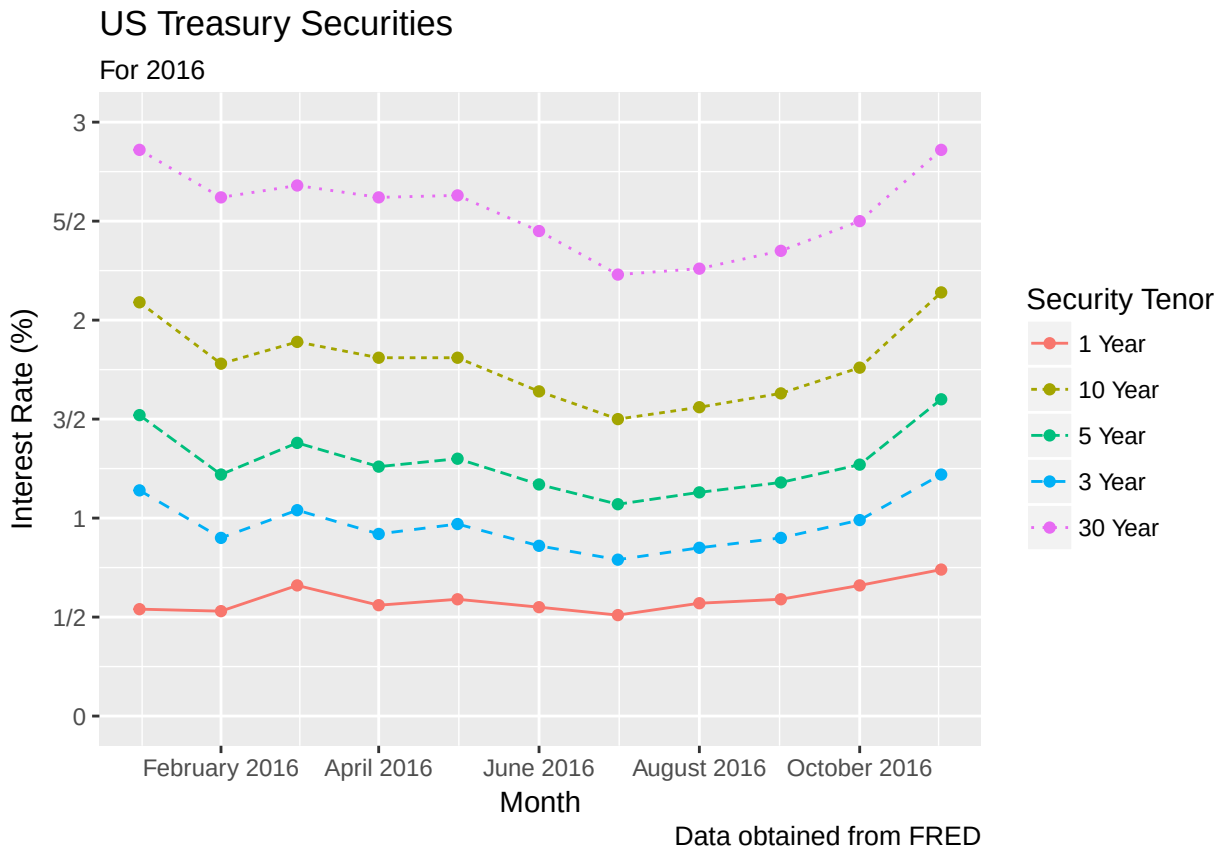
Titles, Labels and Guides

Titles

Titles in GGplot can be assigned in two ways, either via the `ggtitle` function which also includes the `subtitle` argument as well as via the `labs` command.

The `labs` command allows for labels to be added for a title as well as for any scale. In the last plot we made we passed the name of each scale to the scale_ command, but instead we could have put `labs(x = "Month", y = "Interest Rate (%)", color = "Security Tenor")` and gotten the same result. The labs command also allows for a `caption` argument which appears in the bottom right of the chart.

Add the title "US Treasury Securities" subtitle "For 2016" and caption "Data obtained from FRED" to the linechart inClass6. Assign your output to inClass7. Your chart should look like this:



Now we are just left wondering how to center our title, which we will get to later.

Guides

A Guide is also known as a legend. So far we have controlled what is displayed in the guide by changing the scale_ commands. Notice that the `name =` argument ends up being the name of the legend for the colors, the linetype, etc in the line chart we made of 2016 treasury yields. Previously in base plot we were able to control the legend by using the `legend` function. Control of the guides is via the scale functions. To change the shapes, colors, or text that appear in the guide you must do that by changing the corresponding scale functions. Additionally ggplot has the `guide` function which controls the mapping between the scale and the guide. We will not explore this function in class.

Graphing from Multiple datasets

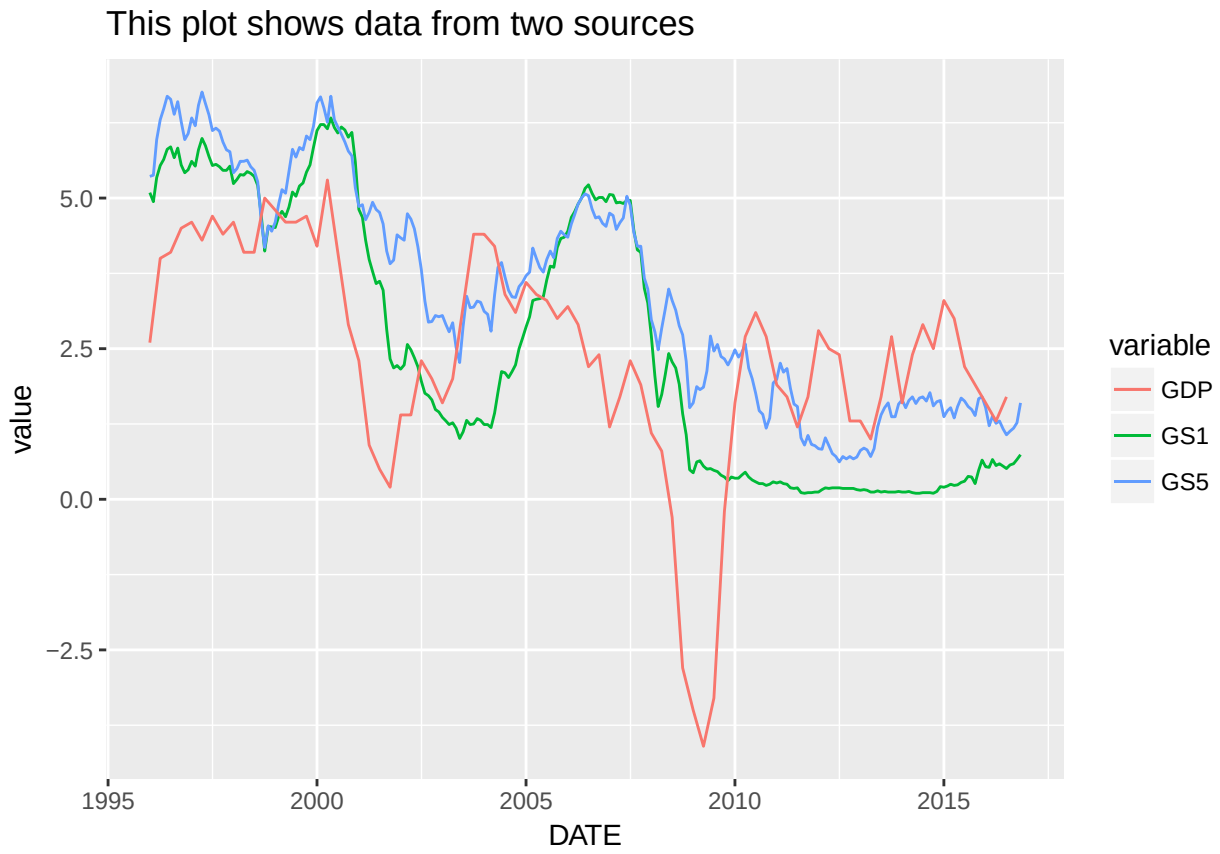
One of the nice aspects of having the `data =` and `mapping =` arguments in the geom_XXX commands as well as the ggplot command is the ability to put data from different sources on the same plot. As long as the x and y values are of the same type, data from different sources can be put on the same plot. (This is one of the huge benefits of ggplots being built up of multiple layers.) Let's now make a chart called multiple.sources where we will add US GDP data to our treasury data. GDP data are found in gdp.csv file.

```
## Getting the gdp data and formatting DATE column to be of same type as treasuries data
gdp <- read_csv(paste0(data.path, "gdp.csv"))
```

```
Output > Parsed with column specification:
Output > cols(
```

```
Output >   DATE = col_date(format = ""),
Output >   `Real GDP Change` = col_double()
Output > )
```

```
gdp <- gdp %>% mutate(DATE = as.Date(DATE, format = "%Y-%m-%d"))

## We will graph 20 years of treasury data again for the GS1 and GS5

treasury.plot.data <- melt(treasuries %>%
                             filter(DATE >= as.Date("1996-01-01")) %>%
                             select(DATE, GS1, GS5),
                           id.vars = "DATE")

gdp.plot.data <- filter(gdp, DATE >= as.Date("1996-01-01"))

colnames(gdp.plot.data) <- c("QTR", "delGDP")

## first layer
multiple.sources <- ggplot(treasury.plot.data, aes(x = DATE, y = value,
                                                   color = variable)) +
    geom_line()

## add a layer
multiple.sources <- multiple.sources + geom_line(aes(x = QTR, y = delGDP,
                                                     color = "GDP"),
                                                 data = gdp.plot.data)
## final version
multiple.sources + ggtitle("This plot shows data from two sources")
```



Look at our axis labels, they are still carrying the same theme values that we set earlier with custom.theme. To go back to the defaults we would have to change the theme.

Notice also one other interesting quality about this chart. Check out the frequency of data for gdp vs treasuries.

```
kable(head(gdp.plot.data,3))
```

QTR	delGDP
1996-01-01	2.6
1996-04-01	4.0
1996-07-01	4.1

```
kable(head(treasury.plot.data,3))
```

DATE	variable	value
1996-01-01	GS1	5.09
1996-02-01	GS1	4.94
1996-03-01	GS1	5.34

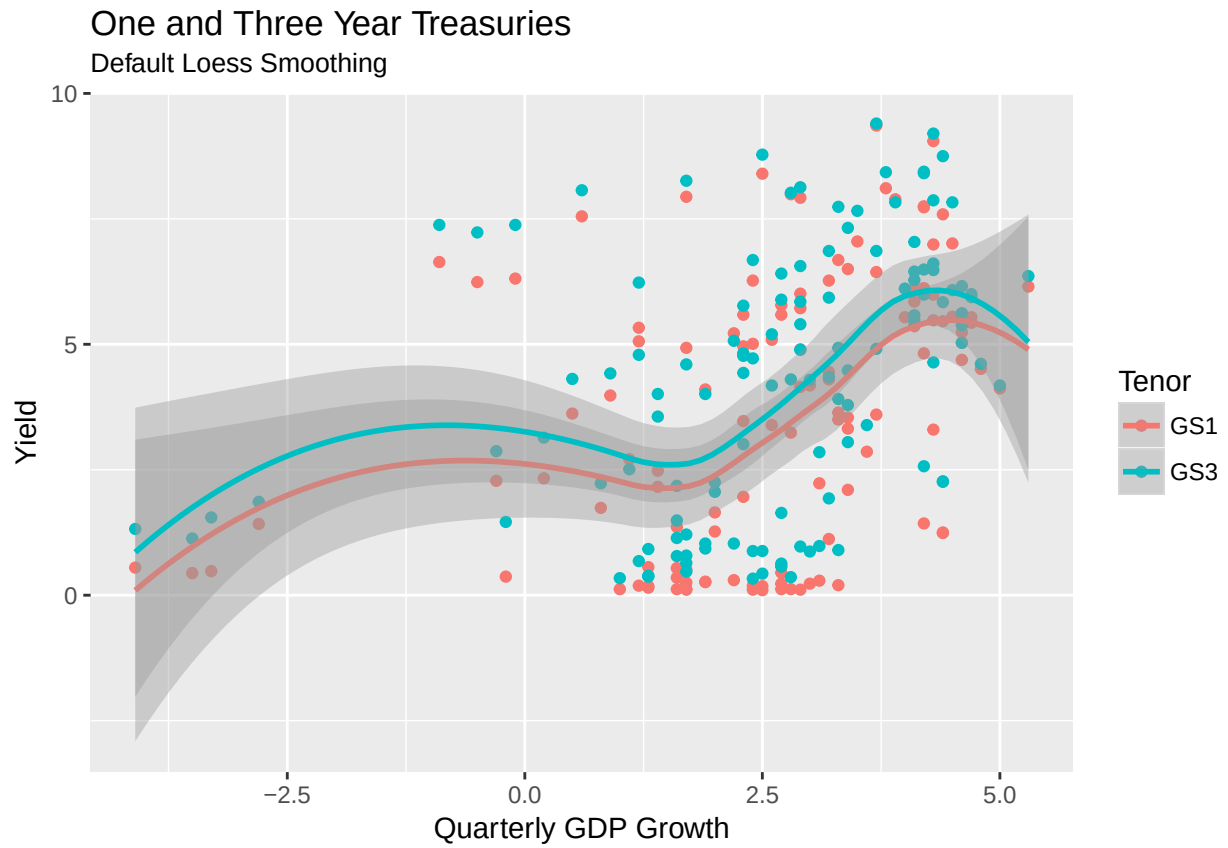
Our GDP data is quarterly while the treasury data is monthly. However, since we are plotting them along x axes with values of the same format, in this case dates, ggplot is able to create a line plot of the gdp data and simply stick it on top of the treasury data since the x and y axes take the same values. If however our x axes were not of the same type we would not be able to plot one on top of the other.

Adding Statistics to Plots

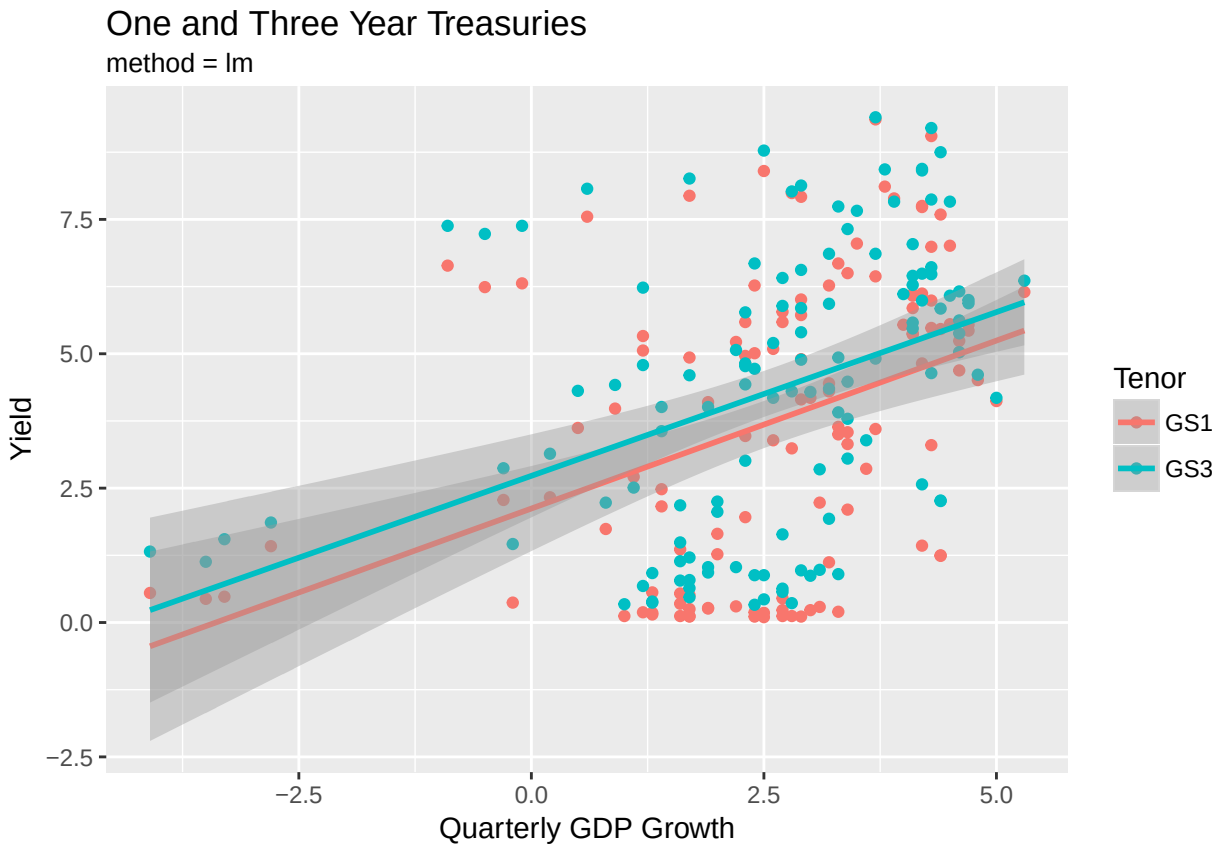
The `geom_smooth` ggplot layer allows you to add trend lines to your graphics. Detailed documentation of ggplot smoothing can be found here. For datasets with under 1,000 observations, the default smoothing algorithm is "loess," for n > 1,000 the default method is "gam," (for generalized additive model). We will look at what the default method looks like as well as what the linear model, "lm" method returns. Let's start with a plot looking at whether gdp growth seems to have an impact on the yield of 1 and 3 year treasuries.

```
## This time we need to merge our data before plotting since we need a scatter plot instead of a time s

theme_set(theme_grey())

plot.data <- merge(treasuries %>% filter(DATE >= as.Date("1986-01-01")) %>%
                       select(DATE, GS1, GS3),
                   gdp %>% filter(DATE >= as.Date("1986-01-01")),
                   by = "DATE", all.x = F, all.y = T)

## Go from wide to long
plot.data <- melt(plot.data %>% select(-DATE), measure.vars = c("GS1", "GS3"))

colnames(plot.data) <- c("GDP", "Tenor", "Yield")

inClass9 <- ggplot(plot.data, aes(x = GDP, y = Yield, group = Tenor)) +
    geom_point(aes(color = Tenor)) +
    labs(x = "Quarterly GDP Growth")

inClass9 + geom_smooth(aes(group = Tenor, color = Tenor)) +
    ggtitle("One and Three Year Treasuries",
            subtitle = "Default Loess Smoothing")
```

```
Output > `geom_smooth()` using method = 'loess'
```




```
inClass9 + geom_smooth(aes(group = Tenor, color = Tenor), method = lm) +
    ggtitle("One and Three Year Treasuries",
            subtitle = "method = lm")
```



The shaded grey areas of the plots are the standard errors associated with the model. (This can be turned off using the `se` argument to geom_smooth).

Look at the two models we showed, do you think these are predictive? Does it seem to you like we could confidently say that there is a strong relationship in this data?

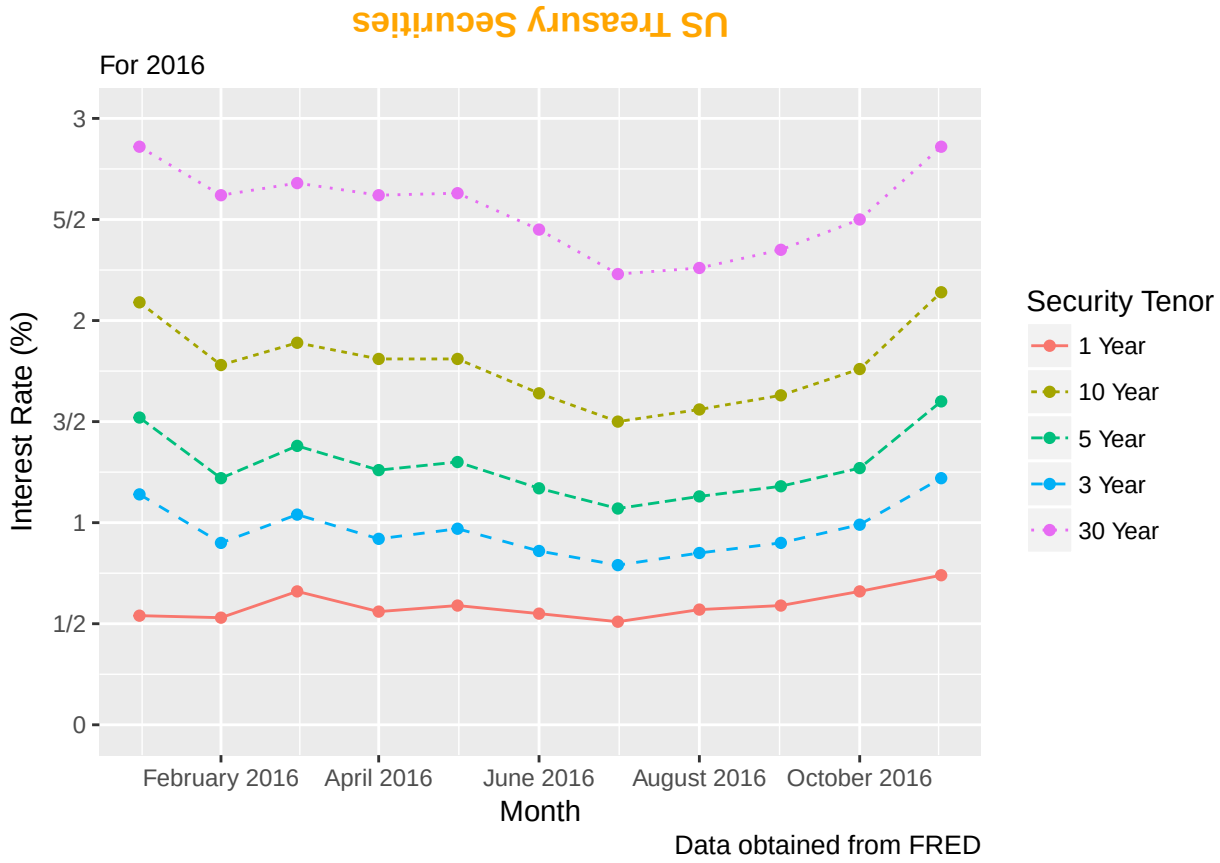
Controlling the Theme

In the title section I asked how we can control the centering of the title and subtitle. In ggplot this element is controlled via the `theme` command. Type `?theme` into your console and take a look at the myriad options available to you via that command. Try not to get overwhelmed since we do not have to specify any of these arguments and will generally not want to specify them.

To change the plot title justification we will need to set the plot.title argument. To change the axis titles we will need to change the axis.title argument. To change the alignment of the legend title we will change the legend.title.align argument. (See where this is going?) Ok, so let's change the title. To do this we will also need the `element_text` function which will be passed as an argument.

Look at the help documentation for element_text

```
aligned.title <- inClass7 +
    theme(plot.title = element_text(face = "bold",
                                    hjust = 0.5, #50% of chart width
                                    angle = 180,
                                    color = "orange"))

aligned.title
```



Now I would not recommend making your title upside down, or even using orange, but it gives a good example of how to use the theme commands.

Using the theme arguments make the following adjustments to the chart you made before with the un-centered title:

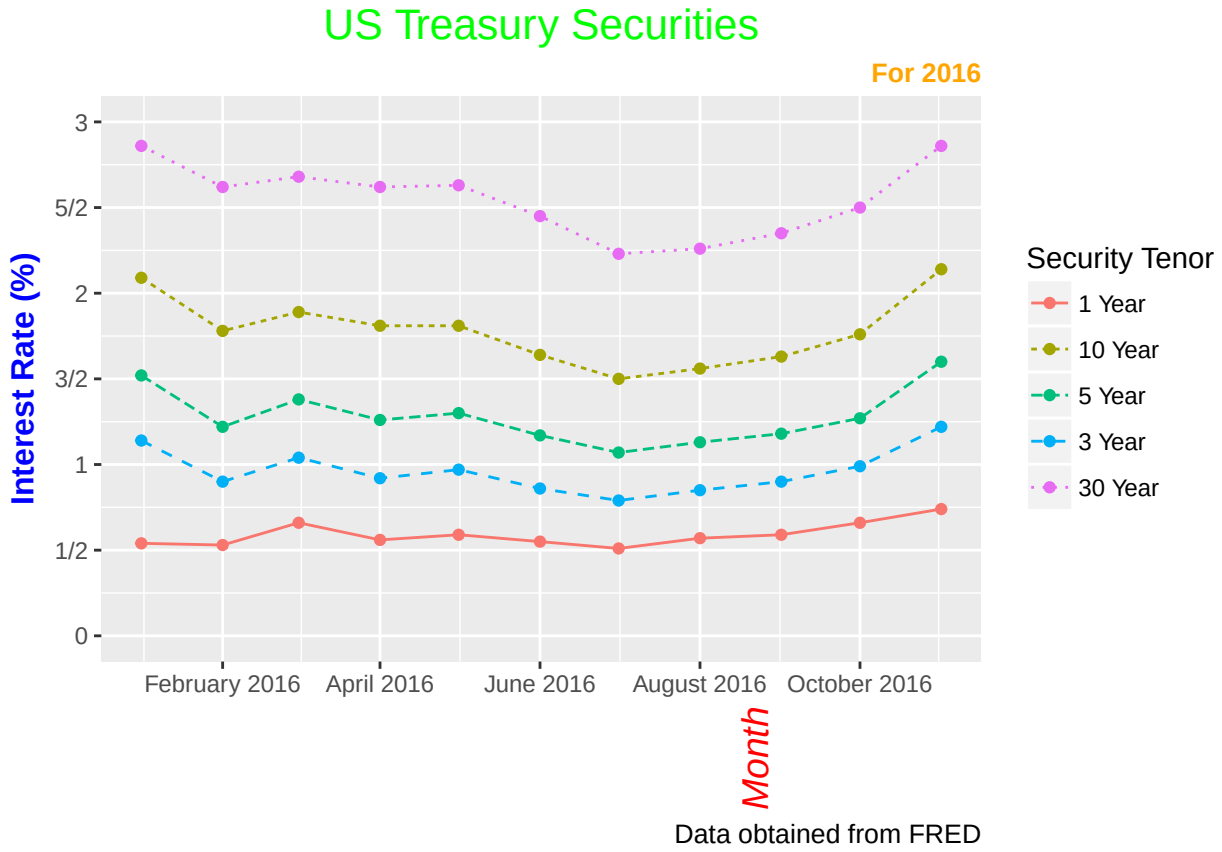
1. center the title and make the font size 16, change the color to green

2. Make the y-axis title color blue, bold the text, and size to 12

3. Make the x-axis title size 14, tilt the text 90 degrees, change color to red, move the text 3/4 of the way right across the axis (vjust argument), and italicize the text.

4. Make the subtitle size 10, bold, orange, on the right

Assign your output to inClass8. Your plot should look like this:

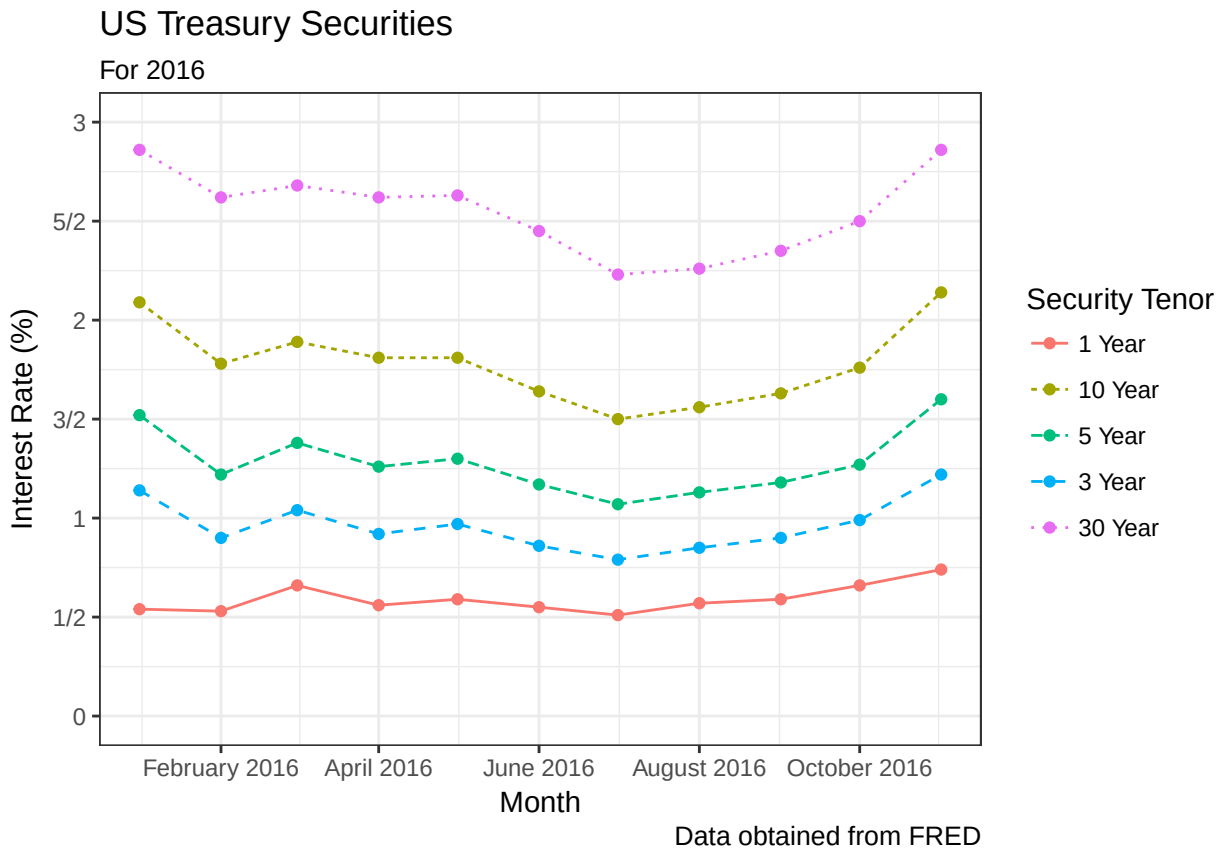


This is an undeniably ugly looking chart and I would recommend against changing your scale labels in this way, but I hope you now understand how ggplot controls the text labelling, as well as many other options for the theme.

Now let's say for some reason you are making a whole suite of charts, (perhaps for your homework with this lecture or for your midterm project), and you want a set of custom theme options to be applied to every chart that you make. i.e. you always want the title to be blue, size 16, and bold for every chart. This is where the theming mechanics of ggplot shine. Note that so far all charts produced have been grey backgrounds with white tick marks. This is the default but can be changed.

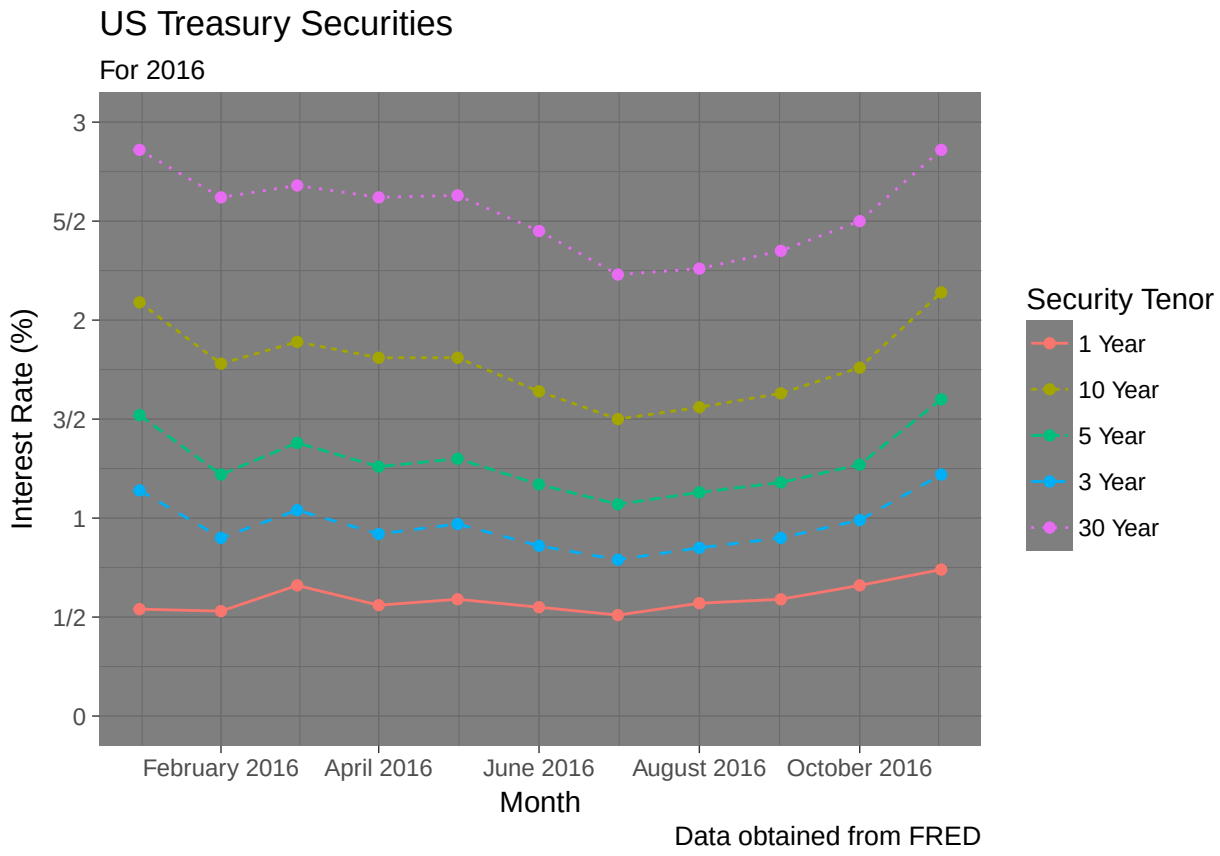
Let's take chart inClass7 and change the theme associated with it.

```
theme.bw <- inClass7 + theme_bw()

theme.bw
```



Calling theme_bw() merely changed all the theme commands for the single plot to be replaced by the default parameters specified in theme_bw(). However, we can also change the theme for the entire environment, this would make it so that all plots would have a new default theme. To do this we can use the theme_set command

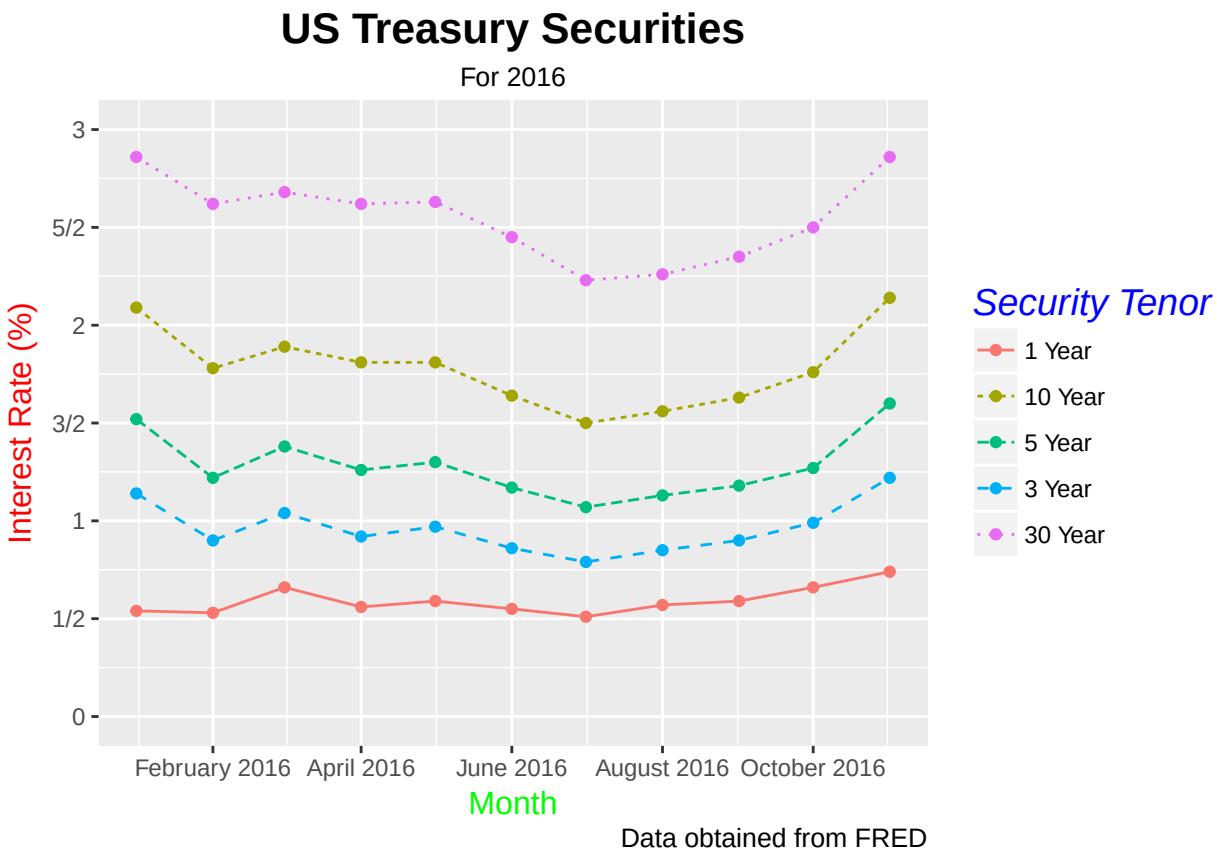
```
theme_set(theme_dark())

inClass7
```



Here we did not change the theme for the chart itself, instead we changed the global theme and then called the chart which now inherited the new global theme.

We can also do this with custom themes

```
custom.theme <- theme_grey() + theme(plot.title = element_text(size = 16,
                      legend.title = element_text(size = 14, face = "italic",
                                                  color = "blue"),
                      axis.title.x = element_text(size = 12,
                                                  color = "green"),
                      axis.title.y = element_text(size = 12, color = "red"),
                      plot.subtitle = element_text(hjust = 0.5))

theme_set(custom.theme)

inClass7
```

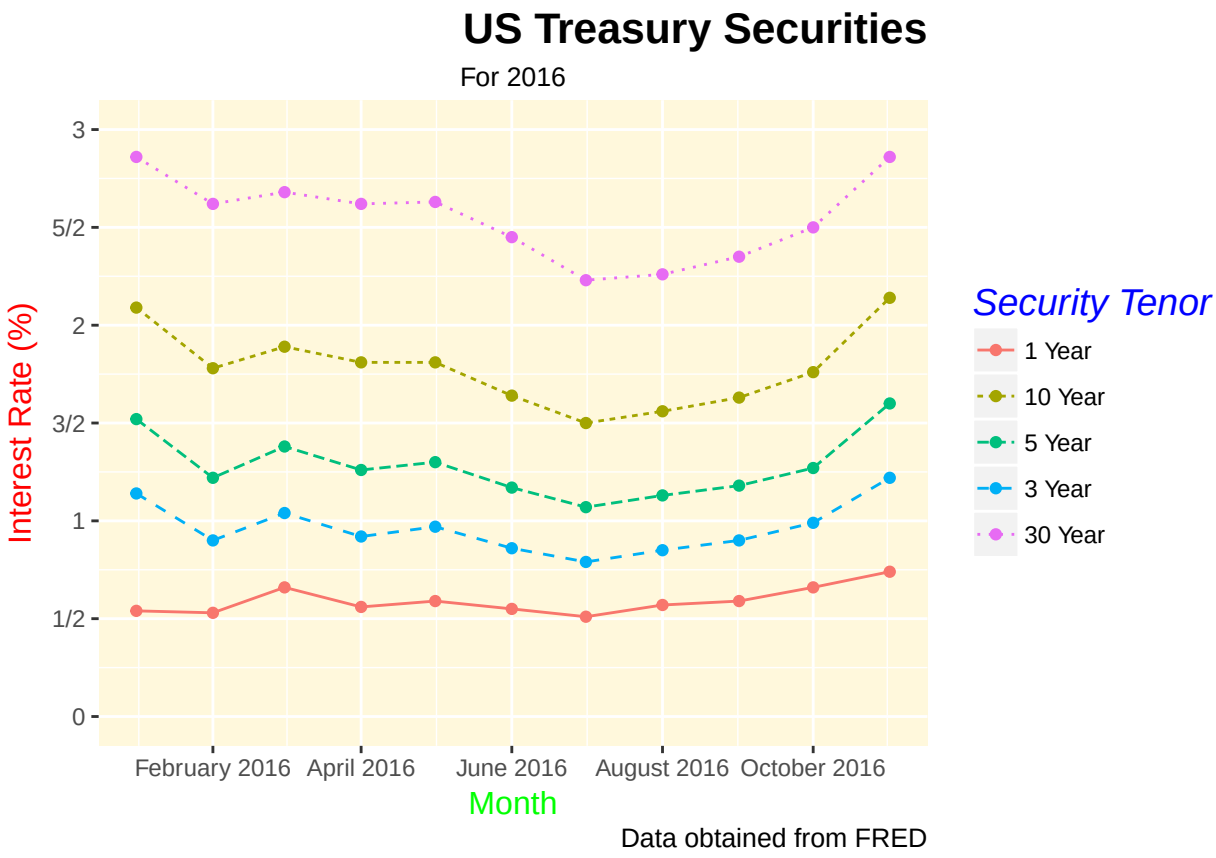


Notice that we had to call theme_grey() and then use the + operator to modify the basic theme_grey() commands. We then set custom.theme like we did above with theme_dark(). We did this because when setting the theme, we must pass in a complete theme. I.e. there must be a value passed in for all the arguments of the `theme` function, otherwise when we go to plot our data ggplot will not know what to do. By calling theme_grey(), (one of the default themes), we are assured that we have a complete theme on which to edit. Had we merely just called `theme_set(theme(plot.title = element_text(...)...)` we would have received an error upon plotting since we would be missing arguments.

If you would like to check if a theme is complete simply use the attr function around the theme. I.E. `attr(theme_grey(), "complete")` returns TRUE.

Now create your own custom theme in a similar vein to the one I just made. Be sure that your new theme changes the title text to appear on the right and changes the background color to "cornsilk" using the panel.background argument. (Note, the panel is not a text element, what element_ command should you use? You should also feel free to change your x and y axis appearance from the default as well as any other changes you'd like to make. Call your new theme custom.theme2.

Check to see if your new theme, custom.theme2, is complete. Once it is complete change your default theme to custom.theme2 and graph inClass7 again. Your plot may look something like this:



Other Themes

GGplot2 comes with multiple complete themes already installed. Just start typing `theme_` in your console and you will see the other themes that come with the ggplot2 package. Additionally, packages such as "ggthemes" and "ggthemr" include more ggplot theme options.

Making Maps in R

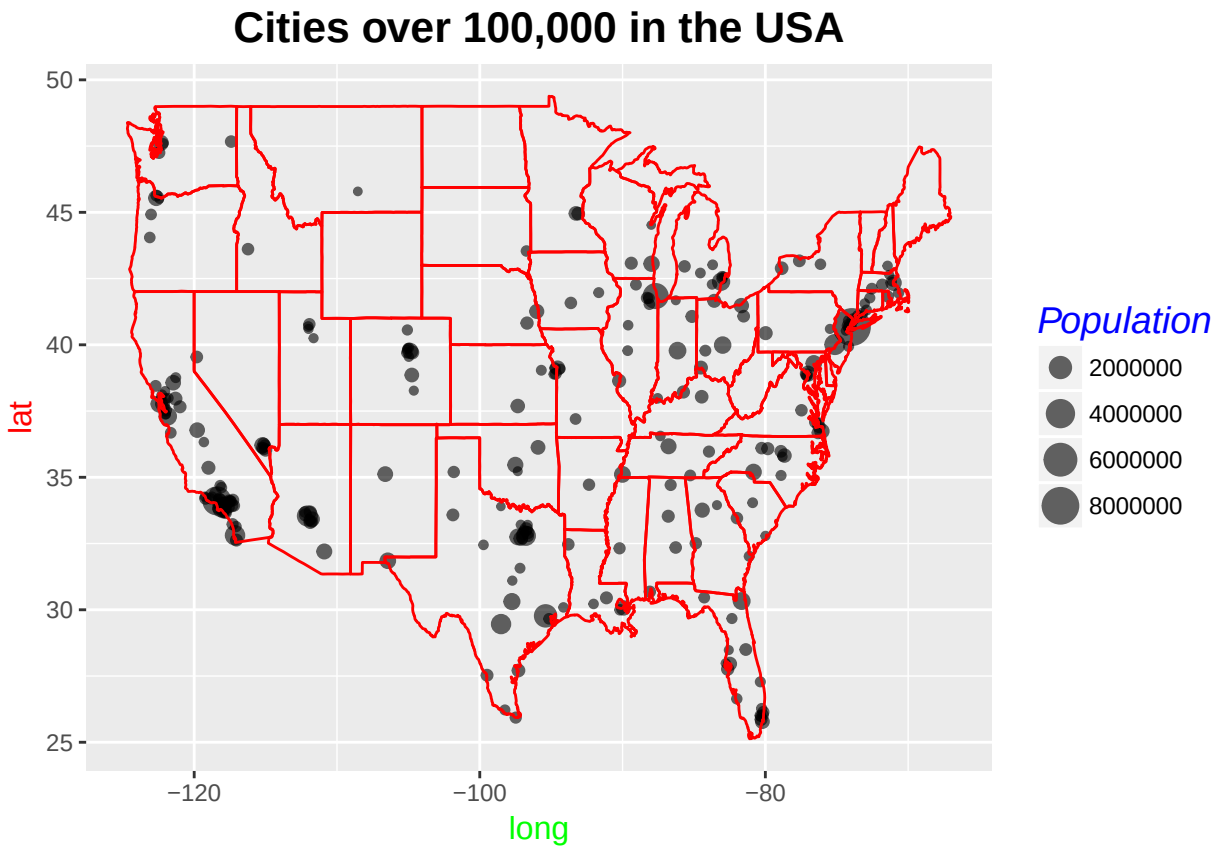
In addition to making line, scatter, barcharts etc... GGplot has the ability to plot comlicated geometries. The most valuable of which is maps, of the US, of different states, etc...

The primary function controlling the map appearance is the `borders()` function we will pass in as a ggplot argument. In order to make these maps we will need map shapefiles. Unfortunately generating these files is beyond the scope of this course, but luckily there are libraries on CRAN with pre-made files ready for us to use. Check the help for `borders()` before attempting to make your own map.

```
library(maps)
```

```
Output >
Output > Attaching package: 'maps'
Output > The following object is masked from 'package:purrr':
Output >
Output >     map
```

```
## Since AK and HI are far from the lower 48 we will not plot them
plot.data <- us.cities %>% filter(pop >= 100000, !(country.etc %in% c("AK", "HI")))

majorCities <- ggplot(plot.data, aes(x = long, y = lat, size = pop)) +
    geom_point(colour = alpha("black", 0.6)) +
    borders("state", colour = "red") +
    ggtitle("Cities over 100,000 in the USA") +
    scale_size_continuous(name = "Population")

majorCities
```



This map merely scratches the surface of what can be done with mapping in R. In the homework we will explore more mapping with ggplot2.