Practicing with ggplot2

2023-06-27

We're continuing following along with The Book of R, but now we're getting familiar with the ggplot2 package. There are distinct advantages to using this package to create and manipulate graphics as compared to the base R graphics. The main distinction being that ggplot2 stores plots as objects.

First, lets get our environment set up by loading the packages we'll need as well as storing the data we'll be plotting.

```
library("tidyverse")
```

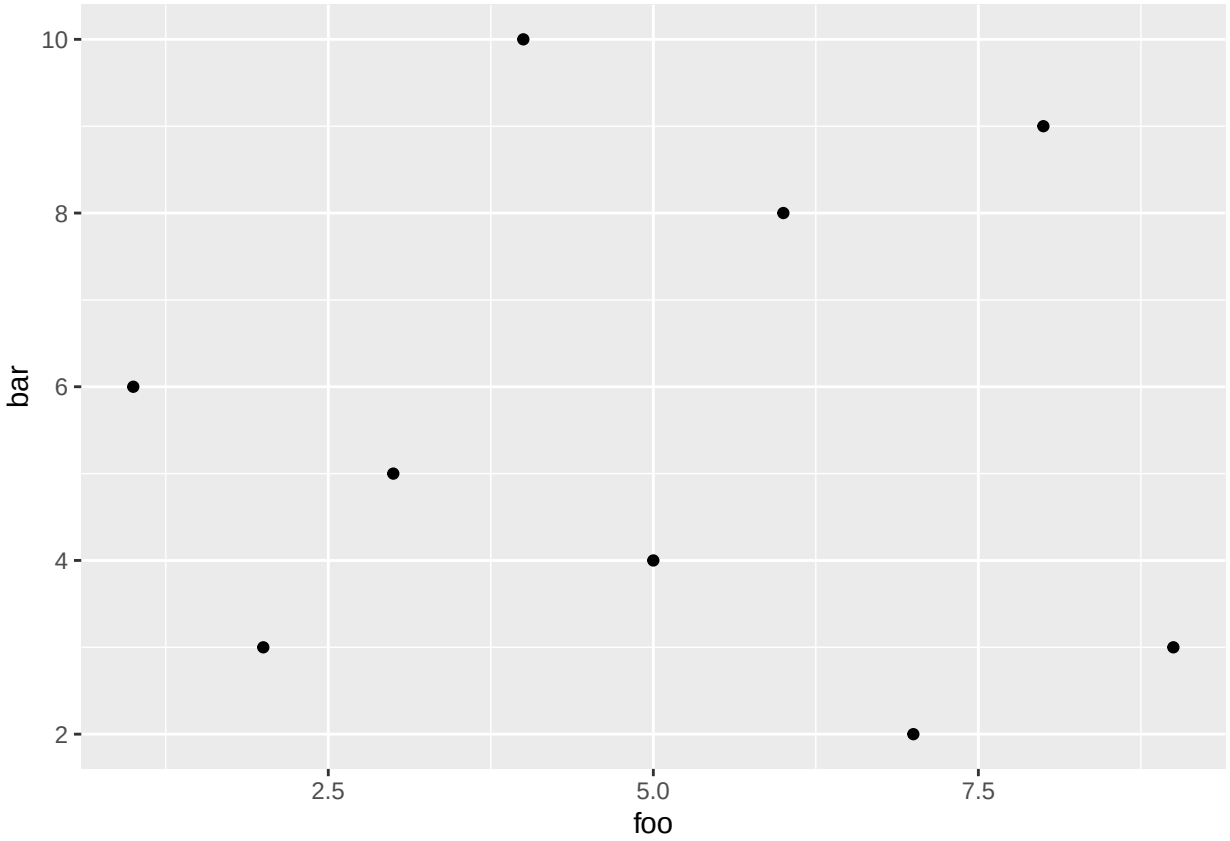
```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.3     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
foo <- c(1:9)
bar <- c(6, 3, 5, 10, 4, 8, 2, 9, 3)
fbar <- cbind(foo, bar)
```

Perfect, now let's see the basic ggplot2 plot fxn 'qplot' in action.

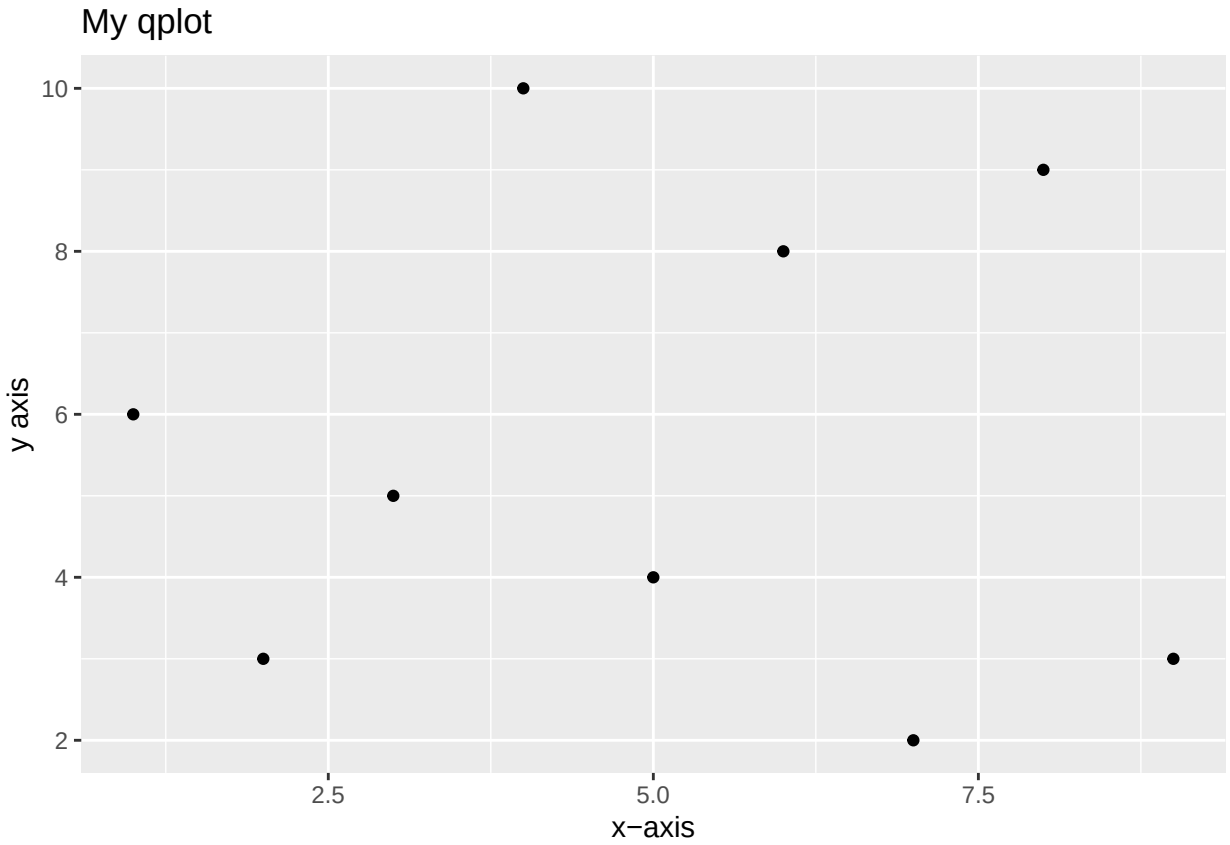
```
qplot(foo, bar)
```

```
## Warning: `qplot()` was deprecated in ggplot2 3.4.0.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```



Okay, cool. We have a nice collection of points. We can also assign this plot to an object, and add some labels in the same way we did with R's basic 'plot' fxn.

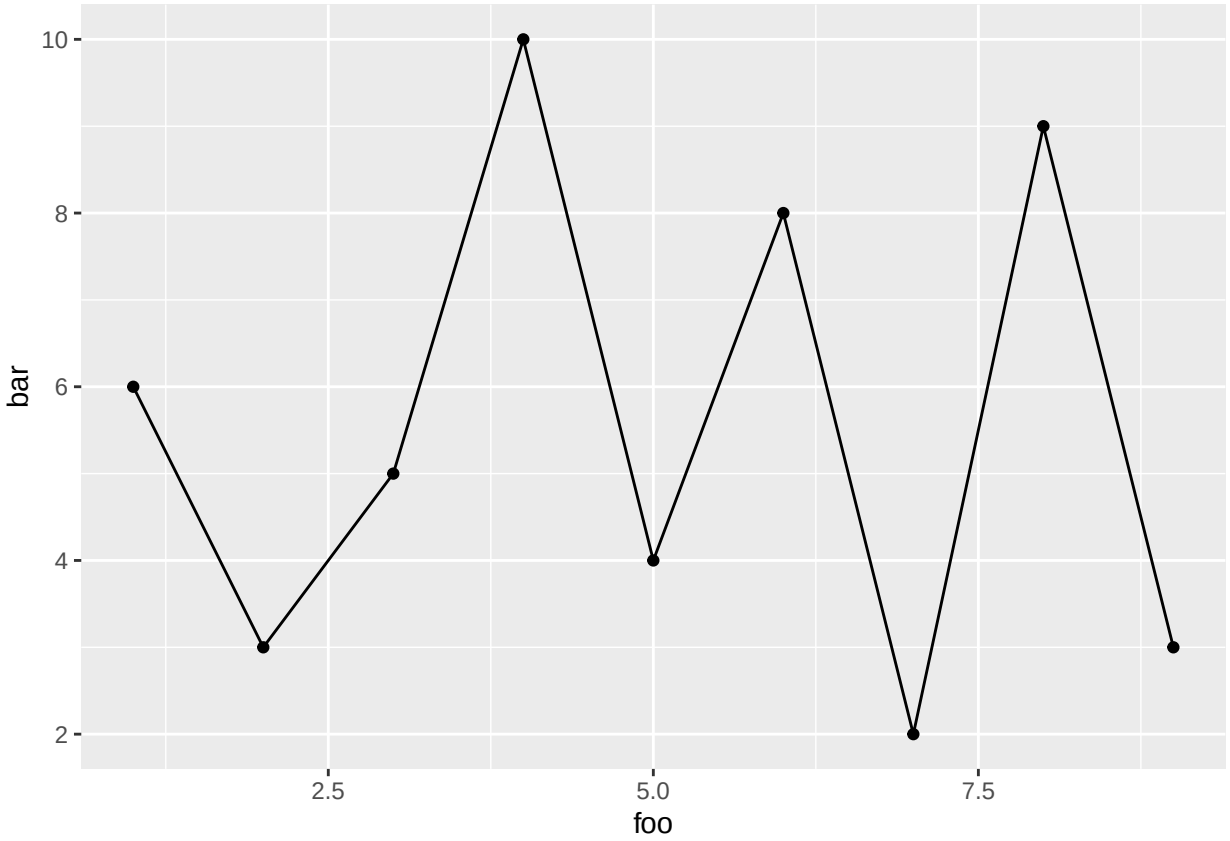
```
q <- qplot(foo, bar, main= "My qplot", xlab= "x-axis", ylab= "y axis")
q
```



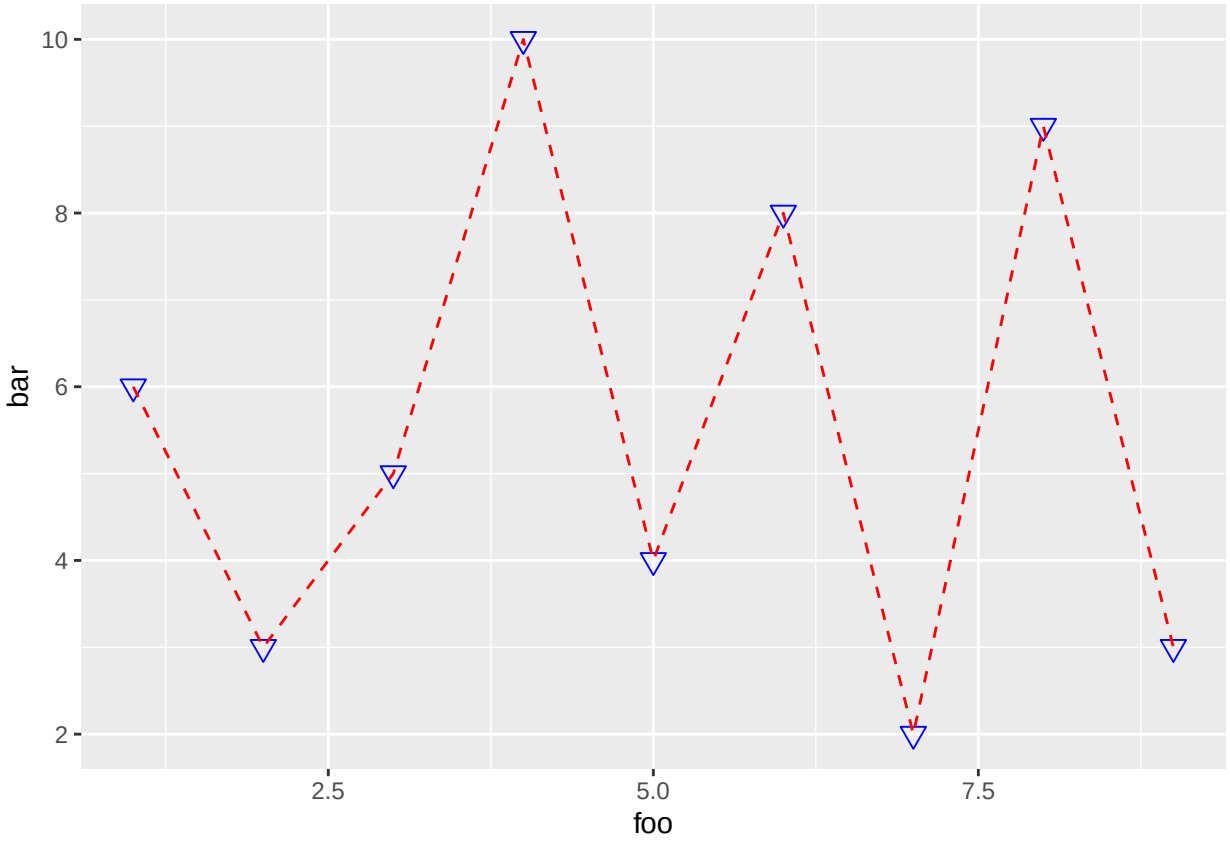
Just like the book said, the plot didn't automatically populate when I assigned it to 'q', I had to 'print' q to display it.

Now, let's spice it up with some features using the 'geom_point()' and 'geom_line()' fxns. The first chunck of code with print the default settings for both of these fxns. Then we'll specify different styles in a second code chunck.

```
qplot(foo, bar, geom= "blank") + geom_point() + geom_line()
```

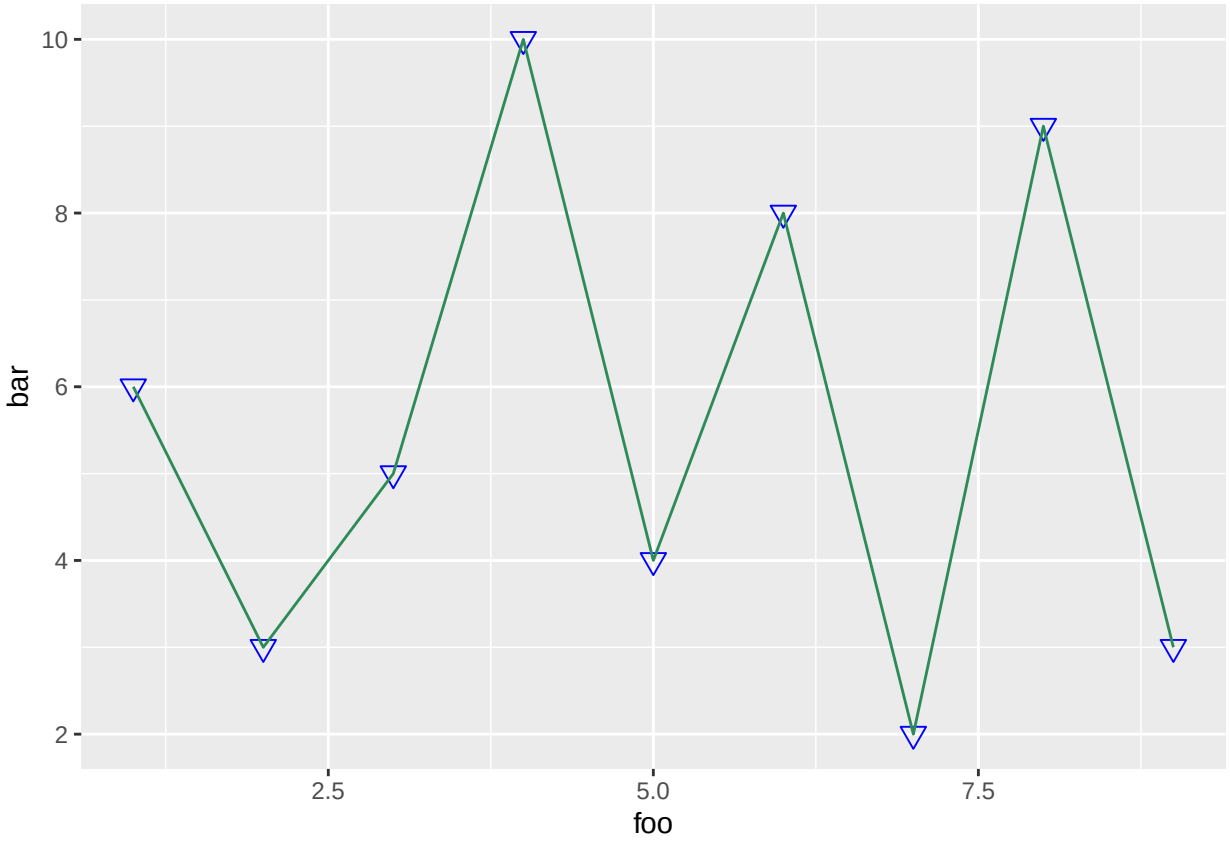



```
qplot(foo, bar, geom= "blank") + geom_point(size= 3, shape= 6, color= "blue") +
  geom_line(color= "red", linetype= 2)
```

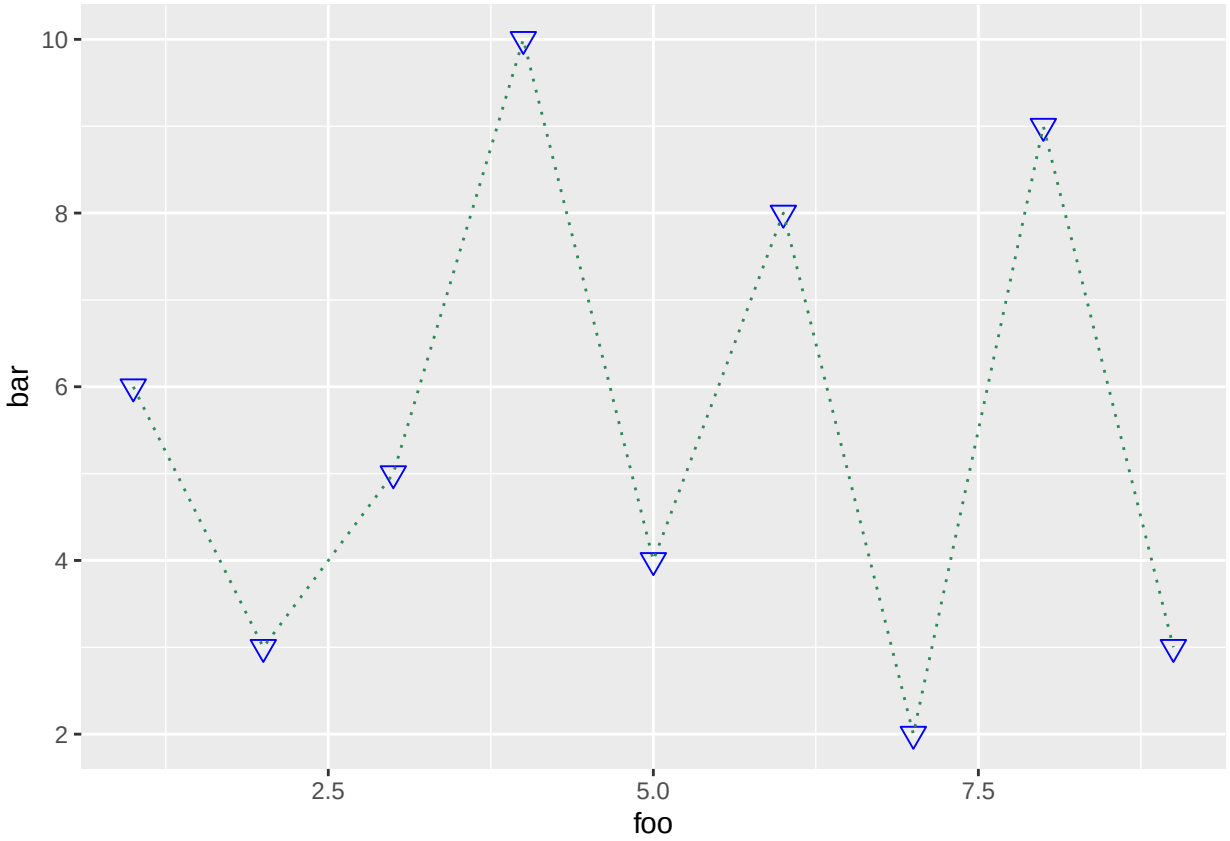


The object-oriented nature of ggplot2 allows you to store a plot and then tweak one the one section of code you want to play around with. For example, if we wanted to adjust the parameters of the line part of the above graph, we can store the main qplot fxn, along with the geom_point() parameters, then tweak the line component of code.

```
q2 <- qplot(foo, bar, geom= "blank") + geom_point(size= 3, shape= 6, color= "blue")
q2 + geom_line(color= "seagreen4", linetype= 1)
```




```
q2 + geom_line(color= "seagreen4", linetype= 3)
```



Complex Graphs

Okay, now we're going to get into more complex plots, by essentially mapping the aesthetic parameters to a dataset that is split or grouped, like a factor object. Let's enter in the same data we did for the last plotting practice to construct that elaborate plot.

```
x <- 1:20
y <- c(-1.49, 3.37, 2.59, -2.78, -3.94, -0.92, 6.43, 8.51, 3.41, -8.23, -12.01, -6.58,
       2.87, 14.12, 9.63, -4.58, -14.78, -11.67, 1.17, 15.62)
```

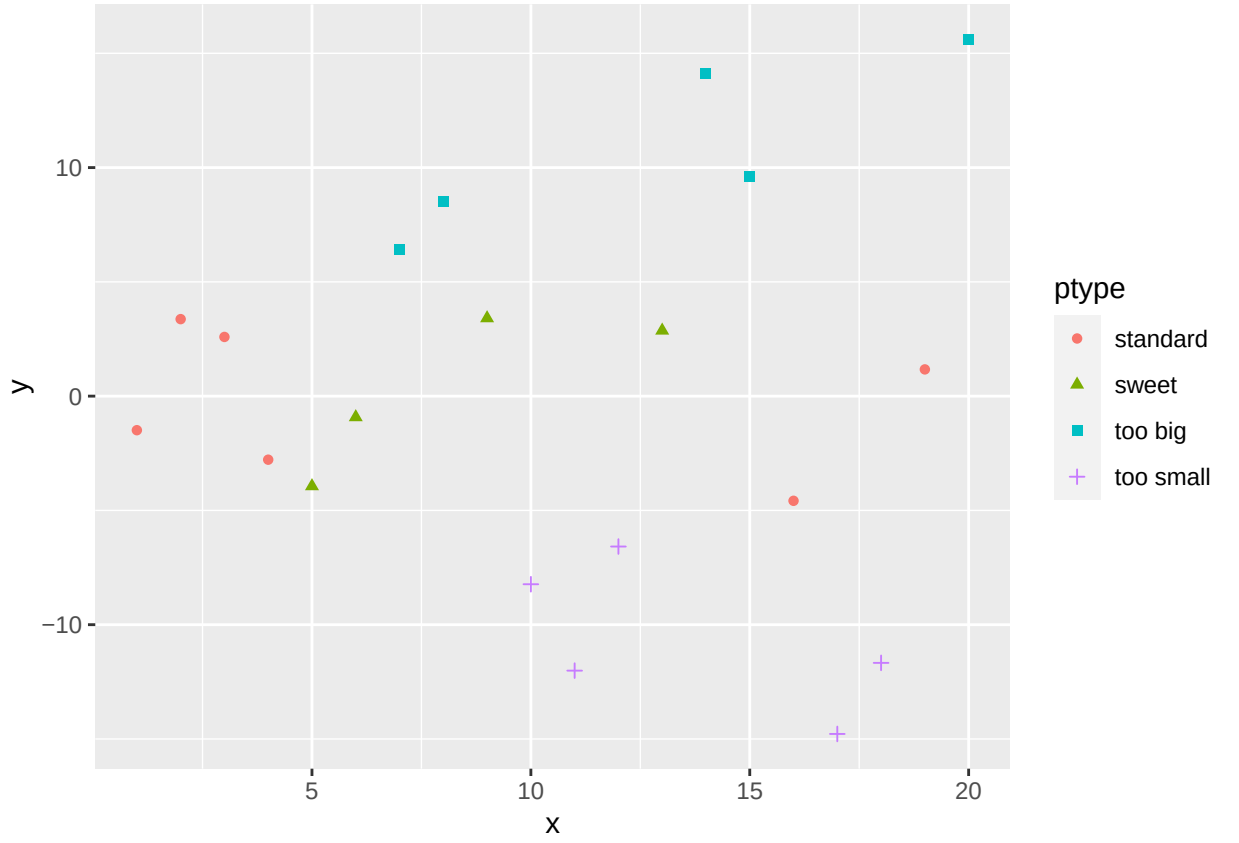
Now that we have our data entered, we can explicitly define the factor as follows:

```
ptype <- rep(NA, length(x=x))
ptype[y>=5] <- "too big"
ptype[y<=-5] <- "too small"
ptype[(x>=5 & x<=15) & (y>-5 & y<5)] <- "sweet"
ptype[(x<5 | x>15) & (y>-5 & y<5)] <- "standard"
ptype <- factor(x=ptype)
ptype
```

```
##  [1] standard  standard  standard  standard  sweet     sweet     too big
##  [8] too big   sweet     too small too small too small sweet     too big
## [15] too big   standard  too small too small standard  too big
## Levels: standard sweet too big too small
```

Cool, we now have a factor of 20 observations, sorted into 4 levels, we can now plot a simple graphic with qplot.

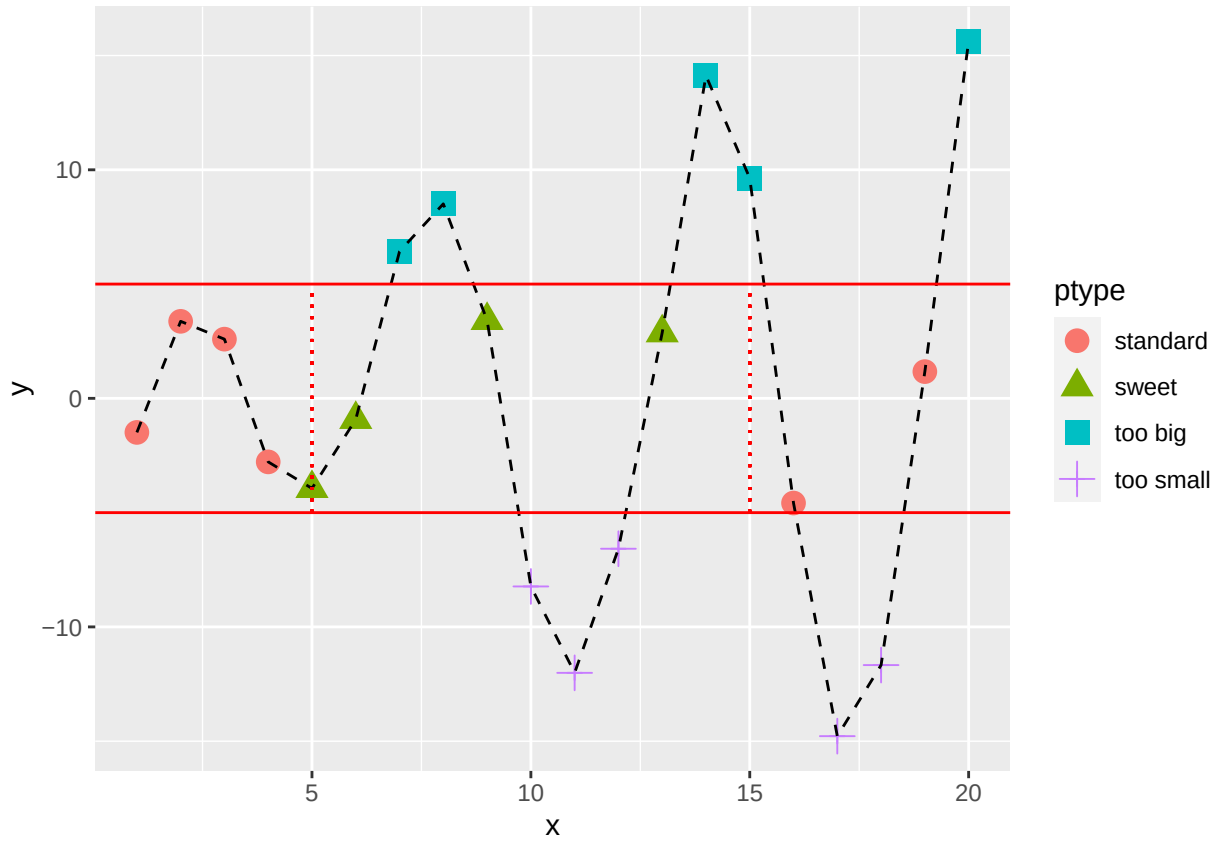
```
qplot(x, y, color= ptype, shape= ptype)
```



That's pretty cool, and somewhat simpler than using the basic R 'plot()' fxn. Crafting the factor seemed involved, but it was a good exercise for subsetting.

Now, let's add to this graph to make it look like complex graph we created in the previous practice using 'plot()'.

```
qplot(x, y, color= ptype, shape= ptype) + geom_point(size=4) +
  geom_line(mapping= aes(group=1), color= "black", lty= 2) +
  geom_hline(mapping= aes(yintercept= c(-5, 5)), color= "red") +
  geom_segment(mapping= aes(x= 5, y= -5, xend= 5, yend= 5), color= "red", lty= 3) +
  geom_segment(mapping= aes(x= 15, y= -5, xend= 15, yend= 5), color= "red", lty= 3)
```



Wow, beautiful! I MUCH prefer this graph to the one manually crafted using the 'plot()' fxn.