ggplot2

Chris

07/01/2019

Overview

Ggplot is part of the tidyverse, and is one of the most popular and powerful packages in R, especially for large and complicated data sets. Some of the reasons to use ggplot (stolen from the old ggplot wiki):

- Automatic legends, colors, etc.
- The “default” output is much nicer than with base graphics, unless starting from the output of a statistical analysis, where people have written nice things in base.
- Easy two-colour gradients to distinguish positive and negative values in GAM surfaces.
- Easy access to ribbons with transparency, for confidence intervals.
- Combine multiple data sets into a single graph with a snap-together, building-block approach.
- Large variety of customizable smoothing overlays, including loess, [list the other R packages whose smoothing you support, and emphasize the 2D (kde?)].
- Handsome default settings.
- Approach your graph from a visual perspective rather than a programming perspective.
- Turn a Cartesian graph into a polar graph with a single statement.
- Store any ggplot2 object for modification or future recall.
- easy superposition (boxplot + points + lines + ...)
- easy facetting
- easy legend

.

Generate some data

So to go through some basic and advanced features of ggplot, we will need some data. We are going to use the following code togenerate some realistic(ish) time series data. We will also be using some of the in-built data sets in R later on.

```
require(dplyr)
```

```
## Loading required package: dplyr
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
require(ggplot2)
```

```
## Loading required package: ggplot2
```

```
##the following was stolen from an excellent blog post on generating data

#write our function extensions
rpois_vec_proto <- function(lambda) {return(rpois(1, lambda = lambda))}
rpois_vec <- Vectorize(rpois_vec_proto)
runif_vec_proto <- function(min, max) {return(runif(1, min = min, max = max))}
runif_vec <- Vectorize(runif_vec_proto)

#Parameters
week_length <- 7
days <- 10*week_length
day_numbers <- 0:(days-1)
weekly_average_range <- c(10, 10000)
ts_count = 5

#generate weekly averages
weekly_averages <- rep(runif(n = ts_count, weekly_average_range[1], weekly_average_range[2]), each
= days)

#Add some randomly selected proportions to add some unique properties
#to each time series. We'll use these in a moment
min_below_fraction <- rep(runif(n = ts_count, 0.25, 1), each = days)
max_above_fraction <- rep(runif(n = ts_count, 1, 2), each = days)
unif_maxadd_fraction <- rep(runif(n = ts_count, 0, 0.7), each = days)

#Add these into a data frame
time_series <- data.frame(
  ts_number = rep(1:ts_count, each = days),
  day_number = rep(day_numbers, times = ts_count),
  min_below_fraction = min_below_fraction, max_above_fraction = max_above_fraction,
  unif_maxadd_fraction = unif_maxadd_fraction,

  weekly_averages = weekly_averages
)

#Build up the time series
time_series.1 <-
  #Use the sin function to generate cyclic sinusoidal shape, according to the length of the week
  mutate(time_series, cycle_unstretched_shape = sin(2*pi*day_number/week_length)) %>%
  #Stretch this using our randomly generated attributes
  mutate(cycle_stretched_shape = ifelse(cycle_unstretched_shape <= 0,
                                        cycle_unstretched_shape*min_below_fraction,
                                        cycle_unstretched_shape*max_above_fraction)) %>%
  #Get the average per day, according to what day of the week it is
  mutate(cyclic_averages = weekly_averages*(1+cycle_stretched_shape)) %>%
  #Get the actual count via Poisson distribution
  mutate(poisson_count = rpois_vec(cyclic_averages)) %>%
  #Add a random uniformly generated count
  mutate(unif_add_count = poisson_count + floor(runif_vec(0, cyclic_averages*unif_maxadd_fractio
n)))

time_series_1 <- select(time_series.1, ts_number, day_number, count = unif_add_count)
##2 sexes
time_series_1$sex<-c("m","f")

time_series.2 <-
  #Use the sin function to generate cyclic sinusoidal shape, according to the length of the week
  mutate(time_series, cycle_unstretched_shape = sin(2*pi*day_number/week_length)) %>%
  #Stretch this using our randomly generated attributes
  mutate(cycle_stretched_shape = ifelse(cycle_unstretched_shape <= 0,
                                        cycle_unstretched_shape*min_below_fraction,
                                        cycle_unstretched_shape*max_above_fraction)) %>%
  #Get the average per day, according to what day of the week it is
  mutate(cyclic_averages = weekly_averages*(1+cycle_stretched_shape)) %>%
  #Get the actual count via Poisson distribution
  mutate(poisson_count = rpois_vec(cyclic_averages)) %>%
```

```
  #Add a random uniformly generated count
  mutate(unif_add_count = poisson_count + floor(runif_vec(0, cyclic_averages*unif_maxadd_fractio
n)))

time_series_2 <- select(time_series.2, ts_number, day_number, count = unif_add_count)
##2 sexes
time_series_2$sex<-c("f","m")

## 2 sites
time_series_2<-rbind(cbind(time_series_1, site=1),
cbind(time_series_2, site=2))
```

Then have a quick look at the structure of the time series generated:

```
head(time_series_2)
```

```
##   ts_number day_number count sex site
## 1         1          0  9107   m    1
## 2         1          1 14325   f    1
## 3         1          2 23742   m    1
## 4         1          3 10387   f    1
## 5         1          4  6890   m    1
## 6         1          5  2861   f    1
```

```
str(time_series_2)
```

```
## 'data.frame':    700 obs. of  5 variables:
##  $ ts_number : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day_number: int  0 1 2 3 4 5 6 7 8 9 ...
##  $ count     : num  9107 14325 23742 10387 6890 ...
##  $ sex       : chr  "m" "f" "m" "f" ...
##  $ site      : num  1 1 1 1 1 1 1 1 1 1 ...
```

Basic GGPLOT2

The basic arguments for ggplot are a little different to the ones you will be used to from plot(). Lets demonstate this on a single time series first:

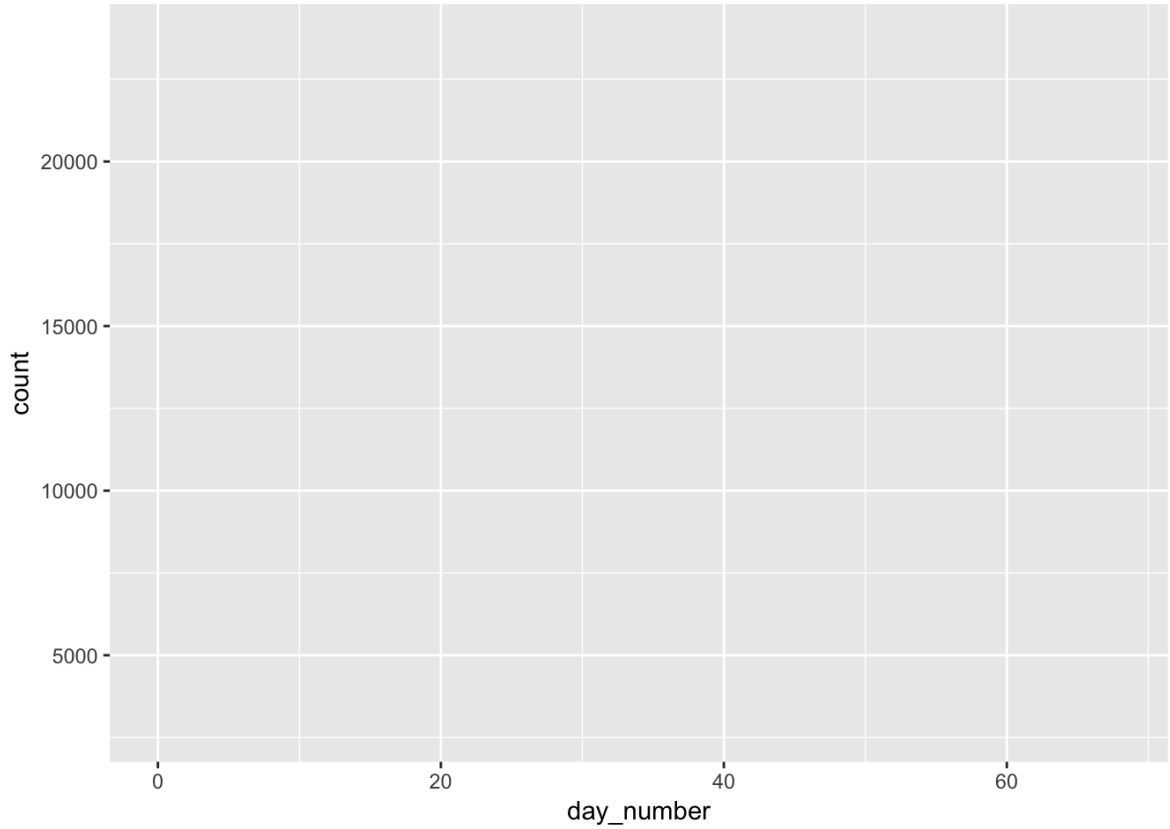
```
ts1<-subset(time_series_2, ts_number==1 & site==1 & sex=="m")
```

Now plot the time series. ggplot arguments always starts with ggplot() to make the ggplot object. Within that the first argument is your data, and then you can specify the aesthetics of the plot using aes(). aes() basically says “look for these things in the data I have specified”.We want to specify x and y aesthetics because we want to plot the abundances (count) through time.

This is a good time to highlight that ggplot() likes long format data, with x and y variables in seperate columns. When you have more complex data you will be able to use additional columns to specify how ggplot should subset and plot your data so that all of the data management is carried out by ggplot, and not by you.

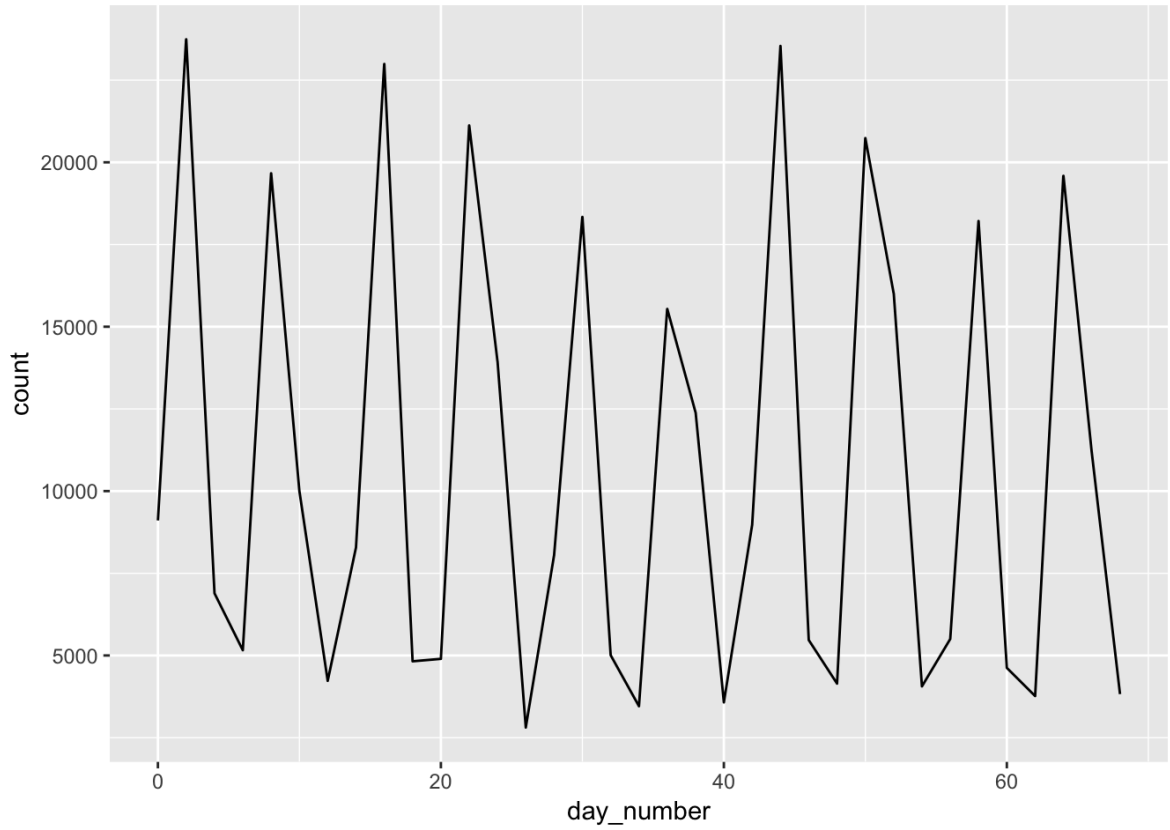
But for the moment, we are just looking at a single time series:

```
ggplot(data=ts1, aes(x=day_number, y=count))
```



So, you'll see there is nothing there! That's because we haven't told ggplot what we actually want it to plot. Points? Lines? Bars? In this case we want lines:

```
ggplot(data=ts1, aes(x=day_number, y=count))+geom_line()
```



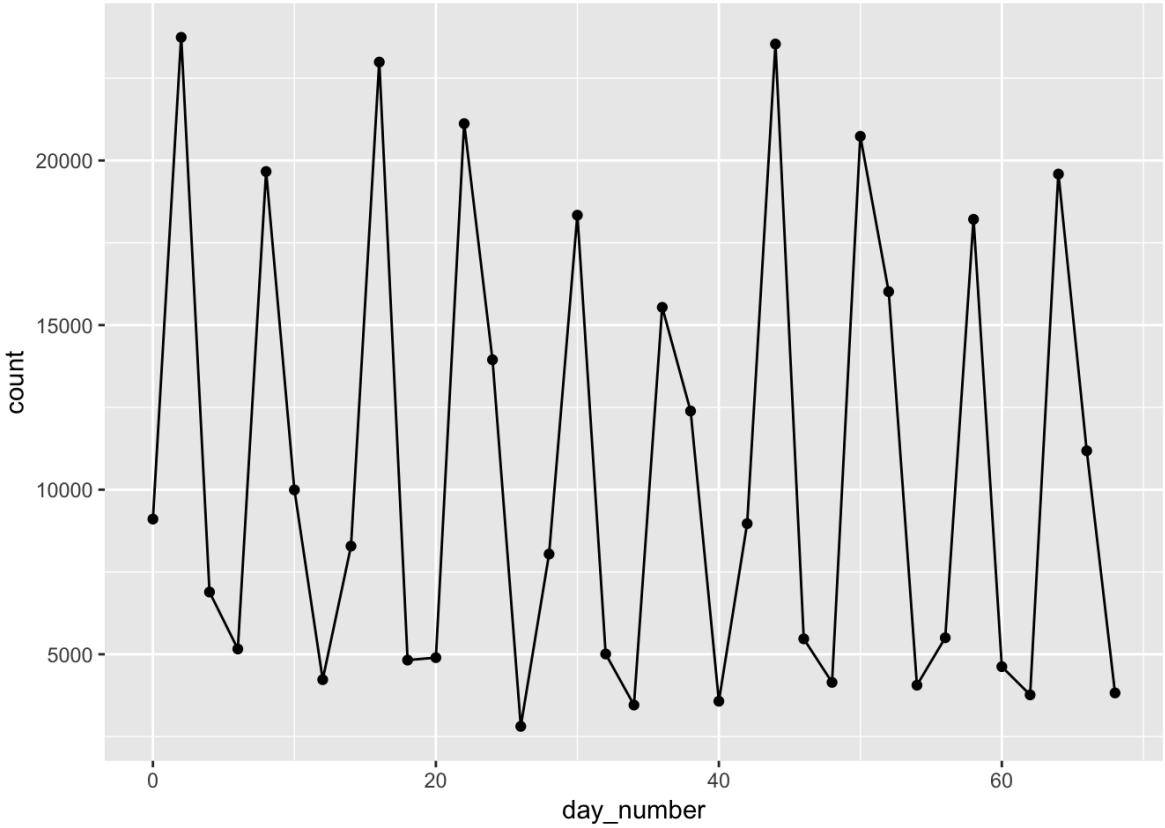
All of the ggplot graphics (lines, points, segments, etc) start with geom_… (geom_line, geom_point etc) and typically have the same arguments to specify what they look like (colours, size, etc).

Instead of specifying the data and aesthetics in the ggplot() function, you can also do it in the geom_… function. This is useful when you are plotting multiple different types of data on the same plot, or subsetting the data in different ways.

```
##this:
ggplot(data=ts1, aes(x=day_number, y=count))+geom_line()
##is equivelant to this:
ggplot()+geom_line(data=ts1, aes(x=day_number, y=count))
```

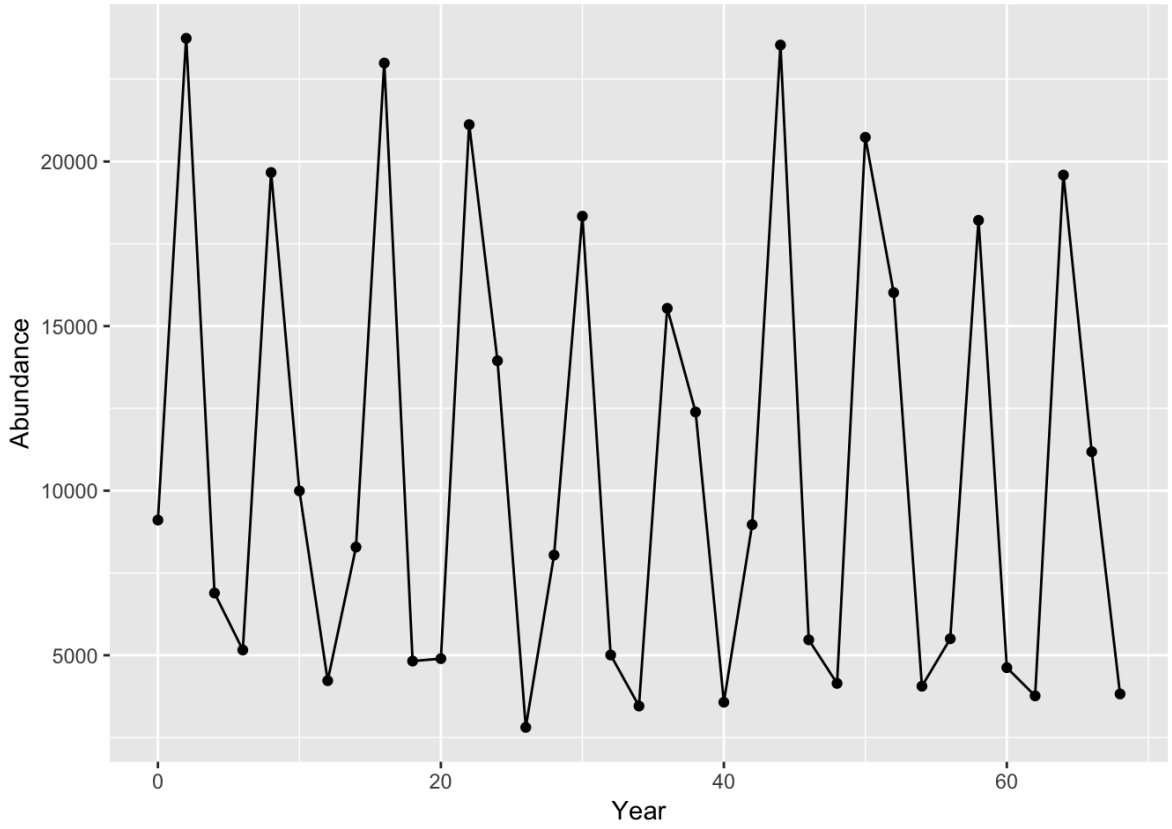
You can also stack multiple plot types really easily. you dont even have to re-specify what the data re, ggplot will just take it from the ggplot() function:

```
ggplot(data=ts1, aes(x=day_number, y=count))+geom_line()+geom_point()
```



You'll notice that the x and y axis titles are being taken from the data we supplied (ts1). This is easy to change:

```
ggplot(data=ts1, aes(x=day_number, y=count))+geom_line()+geom_point()+xlab("Year")+ylab("Abundanc
e")
```

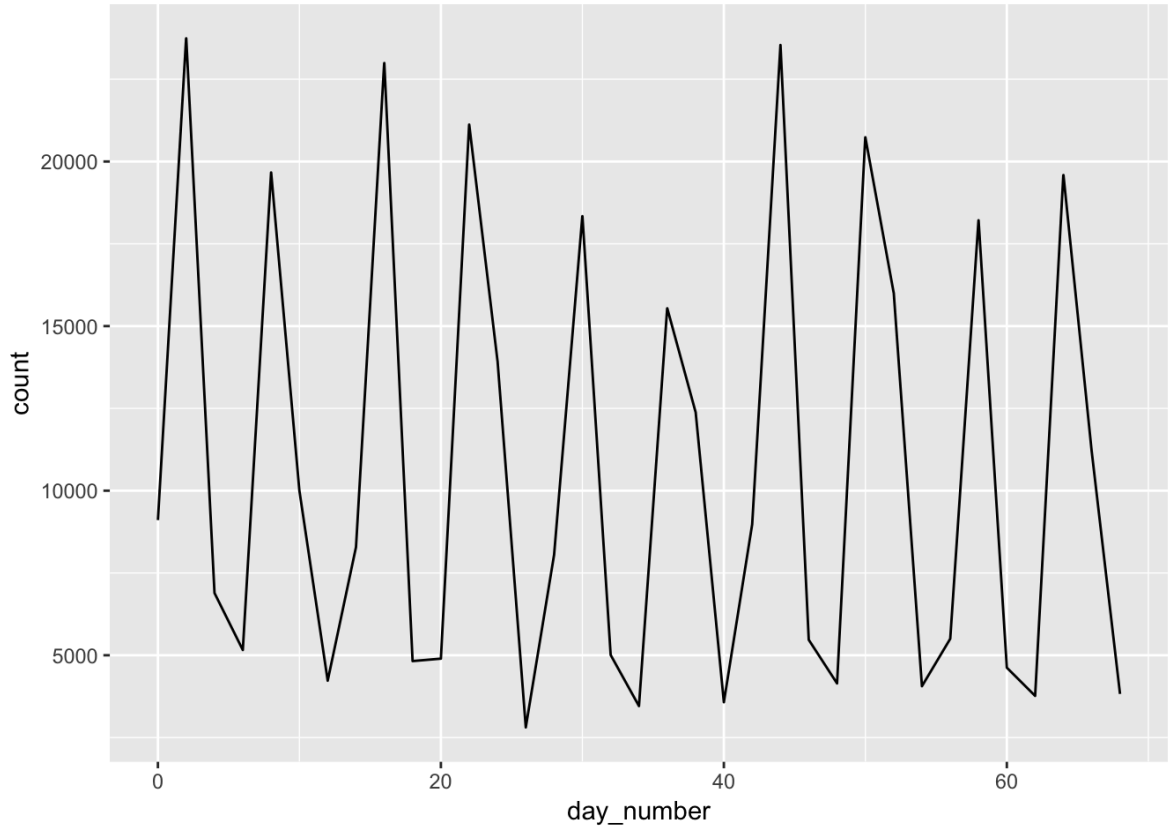


Done! A very basic plot in ggplot.

Splitting up your ggplot arguments

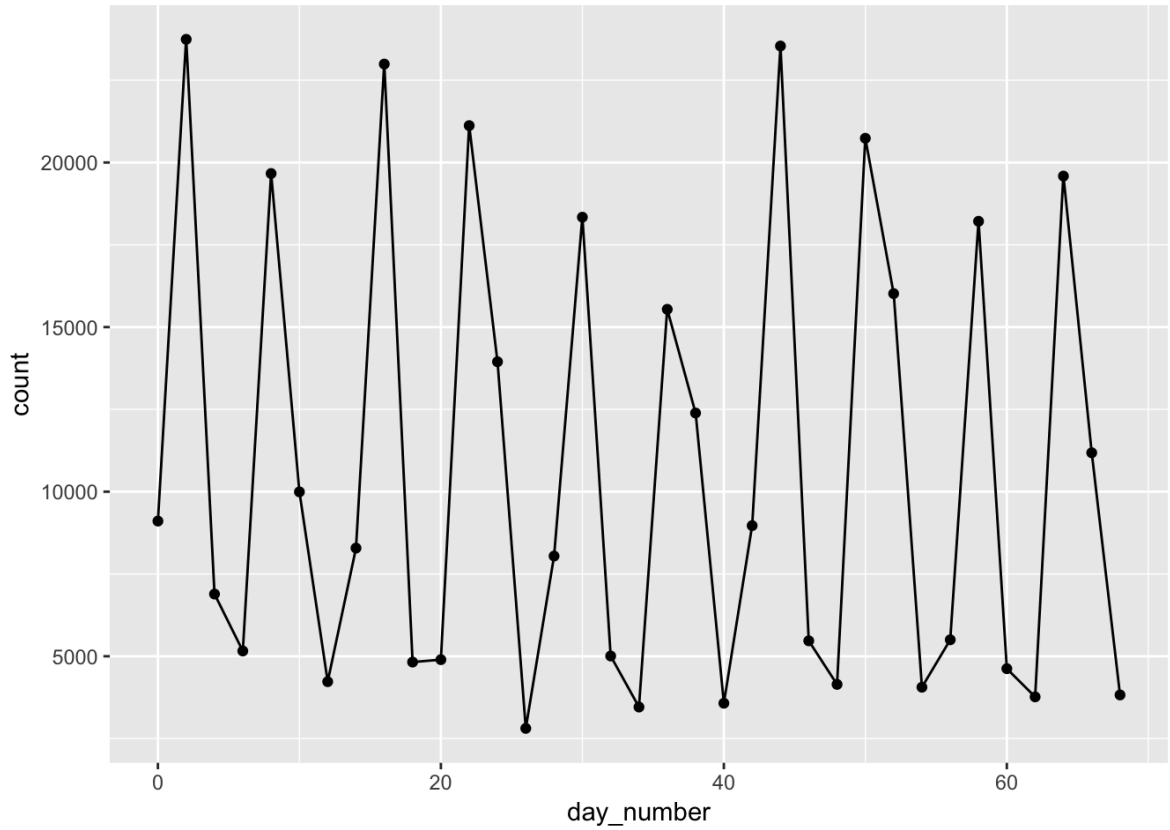
A really useful function of ggplot is the ability to split the arguments across multiple stages, this is especially important when plots become really complex (see later on). You can do this by saving the ggplot objects as objects in R:

```
##make the ggplot object
p1<-ggplot(data=ts1, aes(x=day_number, y=count))
## add the graphic (in this case lines)
p1<-p1+geom_line()
##plot it
p1
```

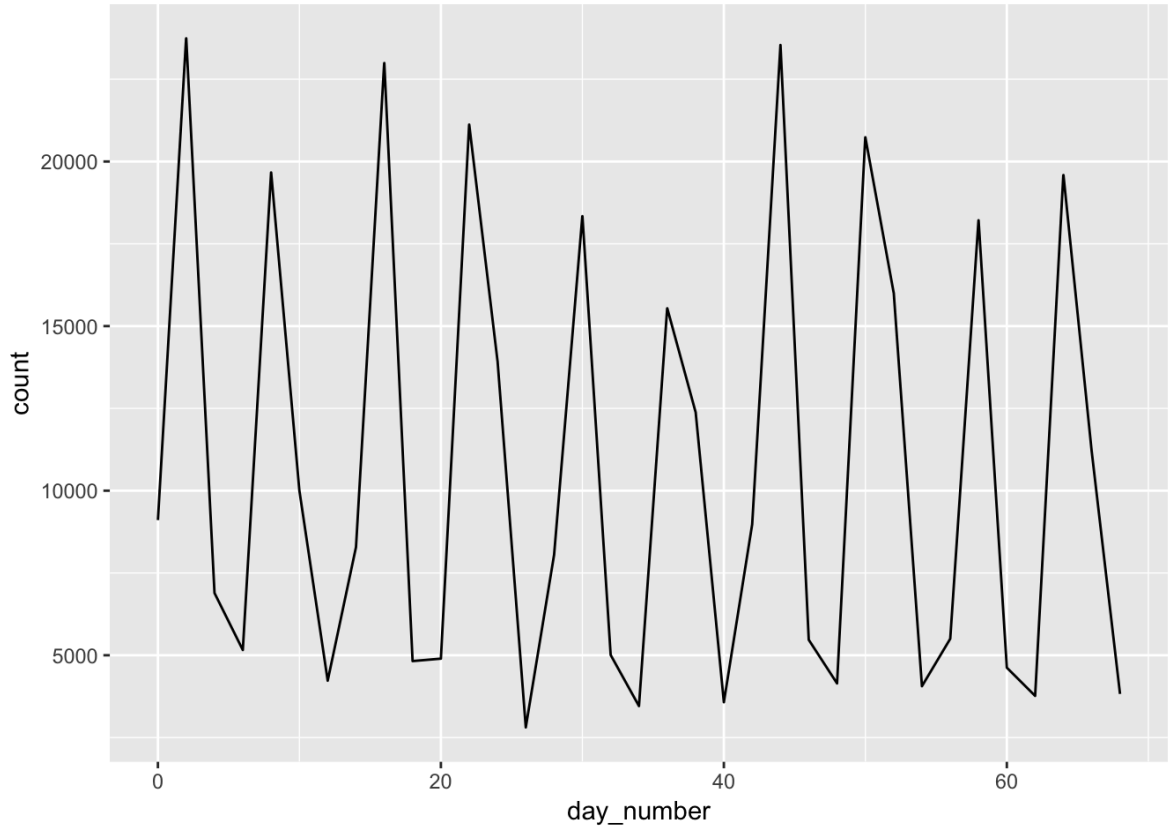


And this means you can test how a graphic might look, but not overwrite it:

```
##try p1 with points too
p1+geom_point()
```

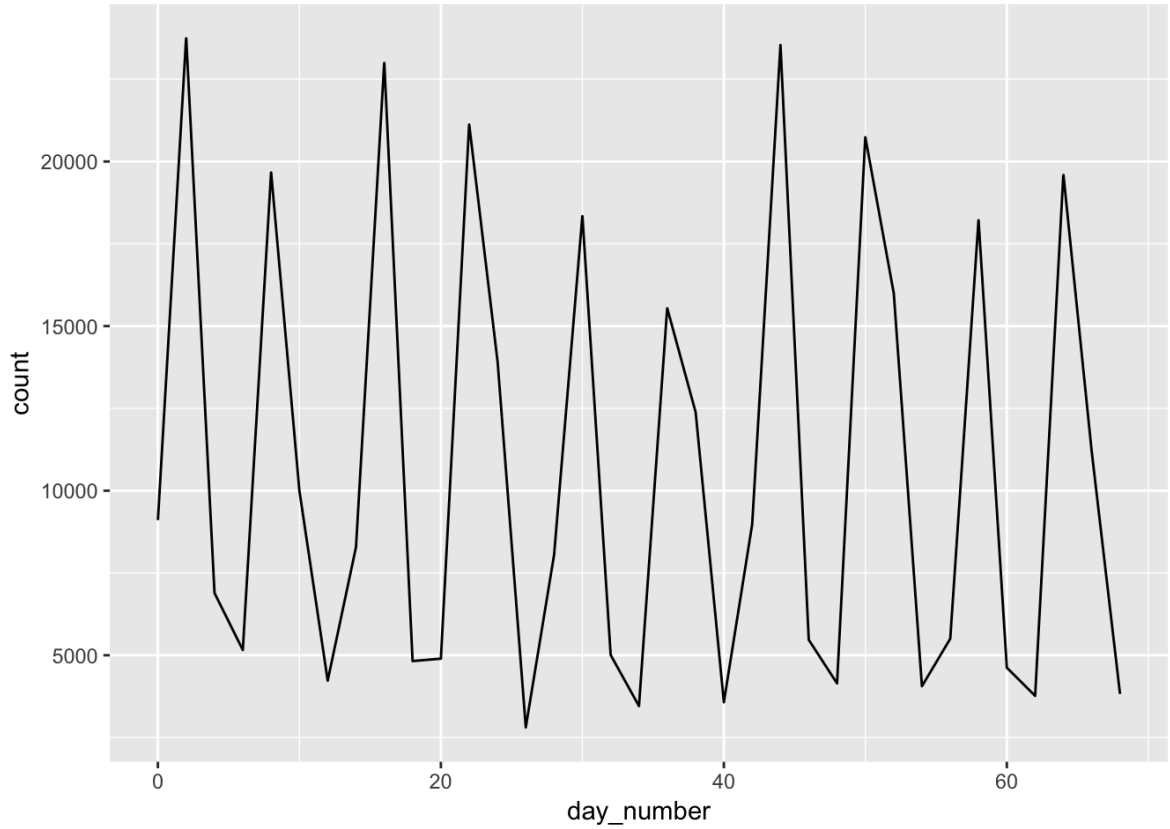



```
##but see that p1 hasn't been altered
p1
```

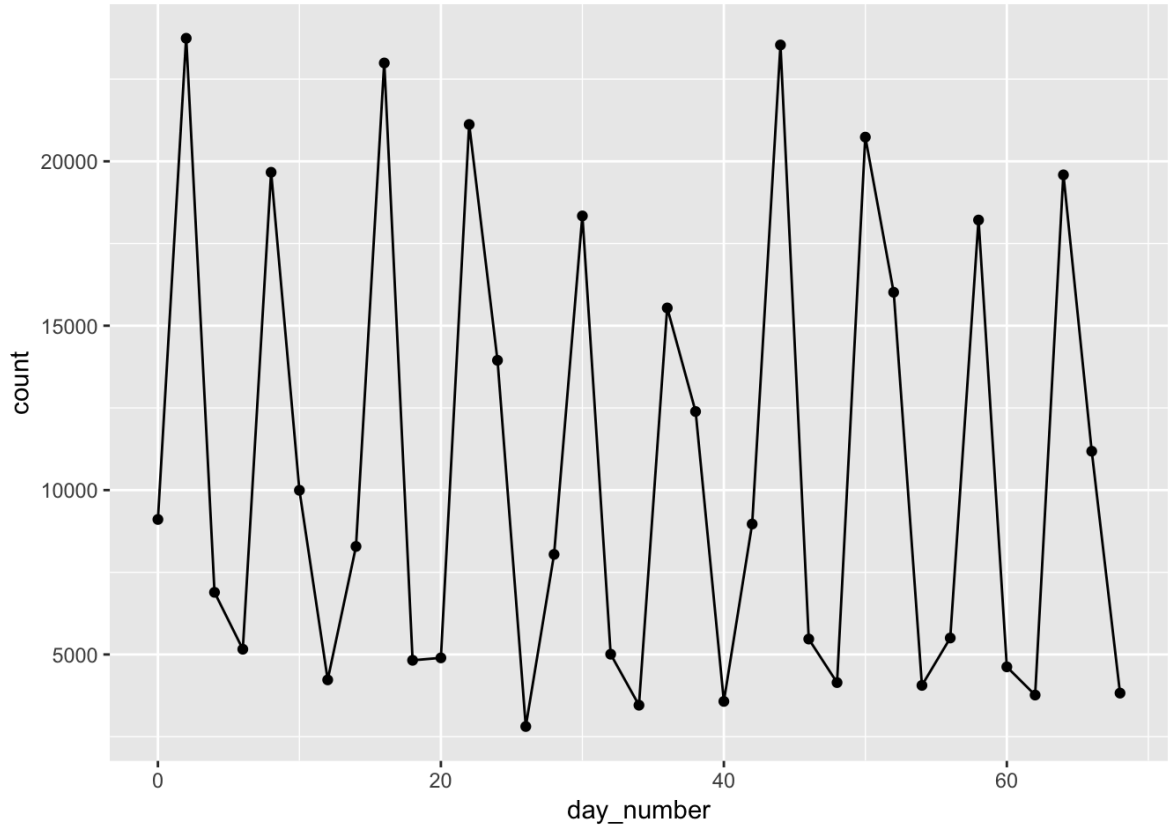


Alternatively you can easily make 2 versions of the same graphic which are a little different, and compare how they look:

```
##make the ggplot object
p1<-ggplot(data=ts1, aes(x=day_number, y=count))
## add the graphic (in this case lines)
p1<-p1+geom_line()
##make a second plot where you have points too, saved to a new object so you dont overwrite p1:
p2<-p1+geom_point()
##compare the two
p1
```



```
p2
```



Big (ish) data

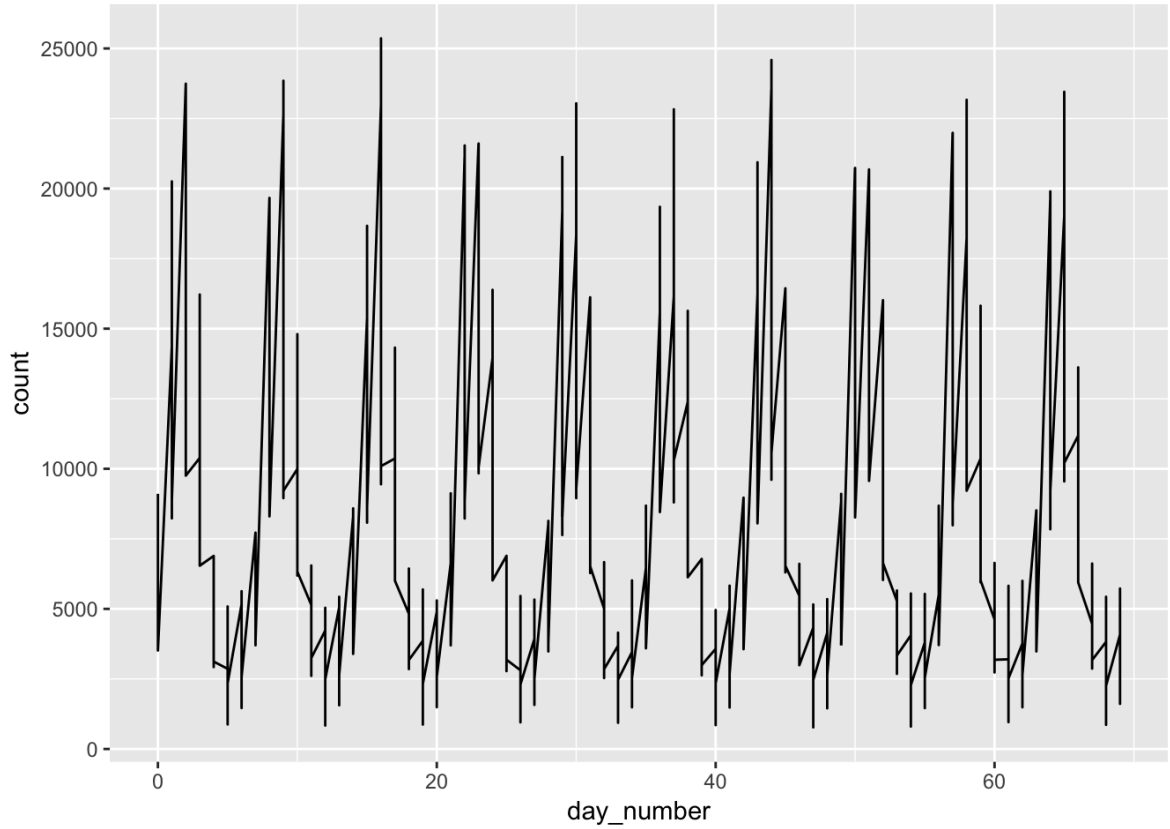
So far we have looekd at really simple data, but where ggplot really comes into its own is when your data gets complicated, with multiple levels and factors. Lets take a look at our full data set:

```
##the structure of the data frame
str(time_series_2)
```

```
## 'data.frame':    700 obs. of  5 variables:
##  $ ts_number : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day_number: int  0 1 2 3 4 5 6 7 8 9 ...
##  $ count     : num  9107 14325 23742 10387 6890 ...
##  $ sex       : chr  "m" "f" "m" "f" ...
##  $ site      : num  1 1 1 1 1 1 1 1 1 1 ...
```

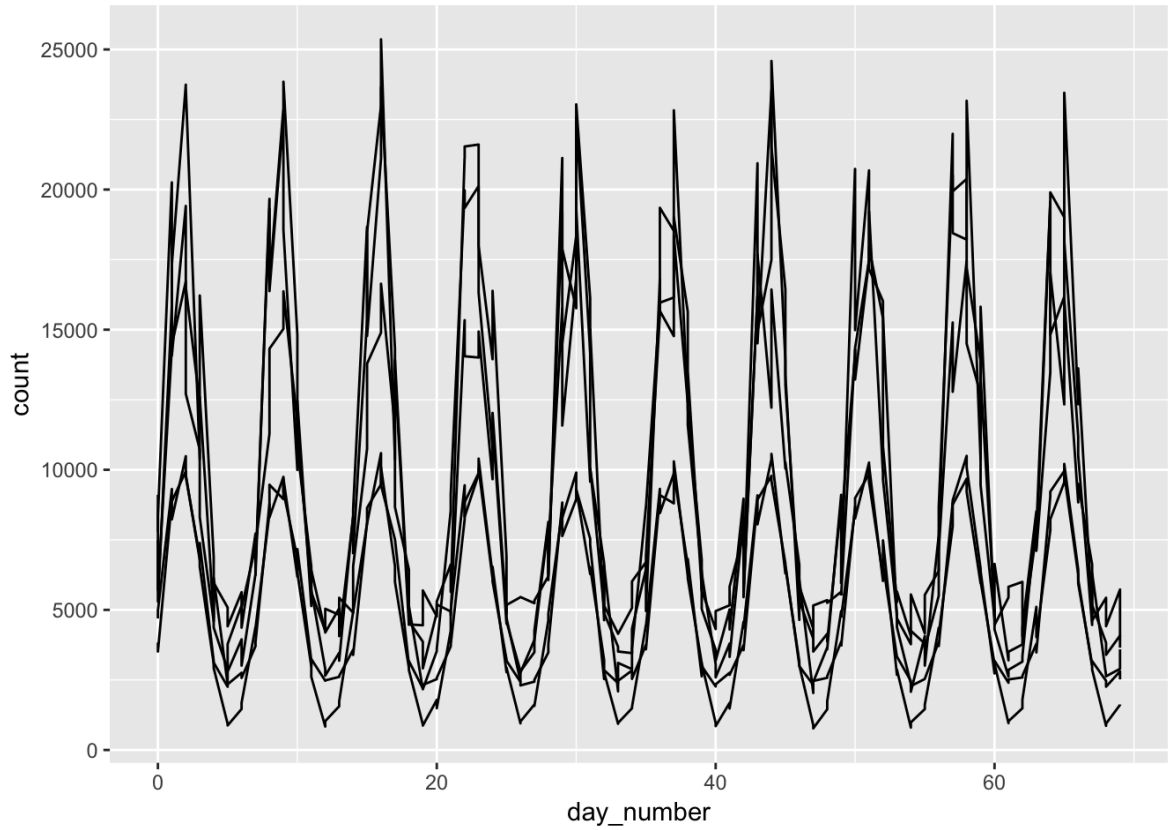
Let's try and plot all of the time series data, using the code we had above:

```
ggplot(time_series_2, aes(x=day_number, y=count))+geom_line()
```



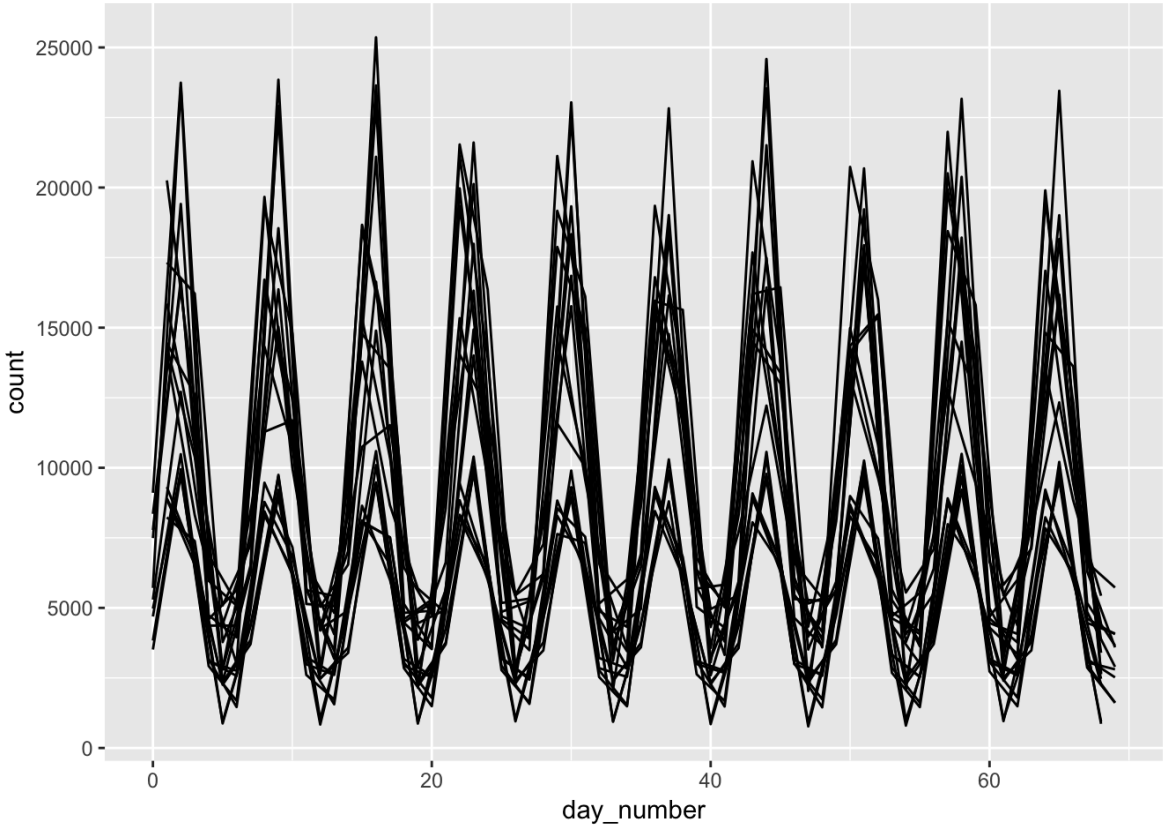
Something clearly isn't right here, we know there are 20 time series, and there is only 1 line. This is a common issue you'll run into, and its because you havent specified that there are groups within the data (i.e. the different time series). We can solve this by telling ggplot to treat each time series (ts_number) as a different data set, and thus plot a seperate line for each one. This is done using the "group=" argument:

```
ggplot(time_series_2, aes(x=day_number, y=count, group=ts_number))+geom_line()
```



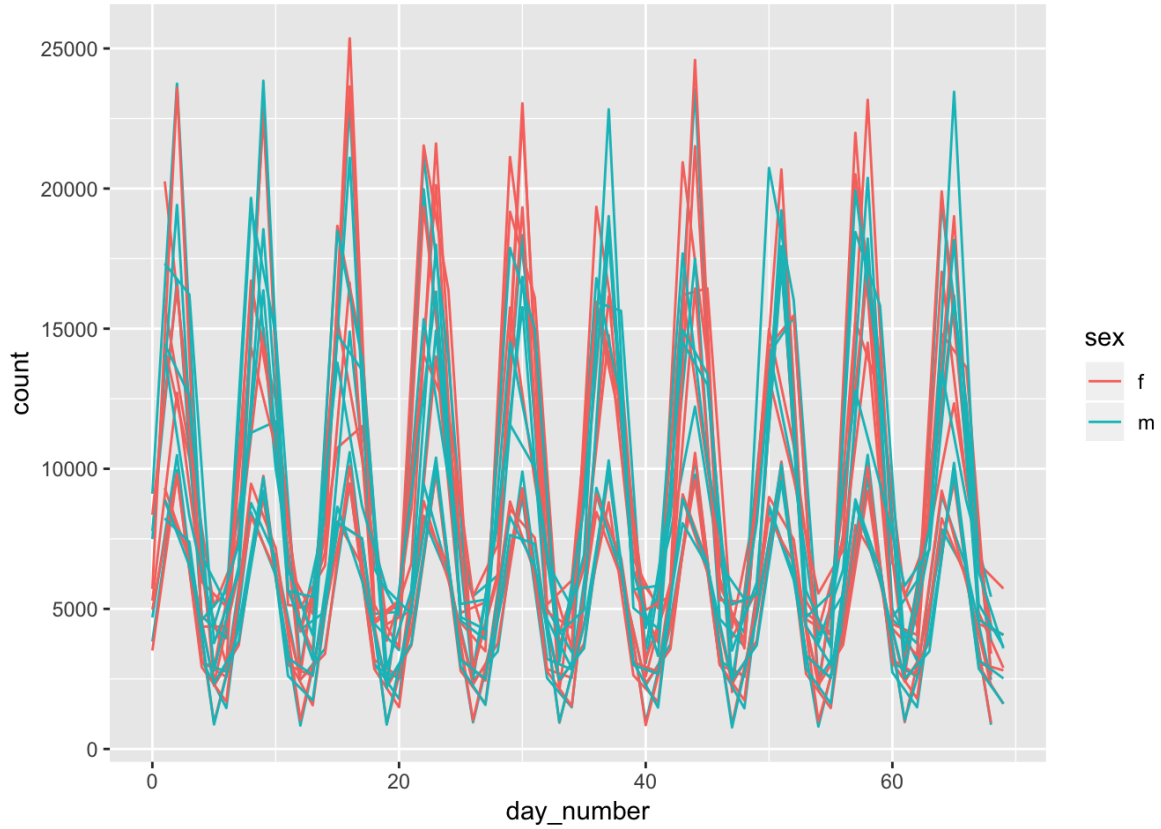
This still doens't look right. Let's think carefully about our data structure: we have 5 time series, at each of 2 sites, with data on males and females. So we need to take all of these into account. To tell ggplot that there are multiple columns defining the different data sets, we use interaction():

```
ggplot(time_series_2, aes(x=day_number, y=count, group=interaction(ts_number, site, sex)))+geom_li
ne()
```



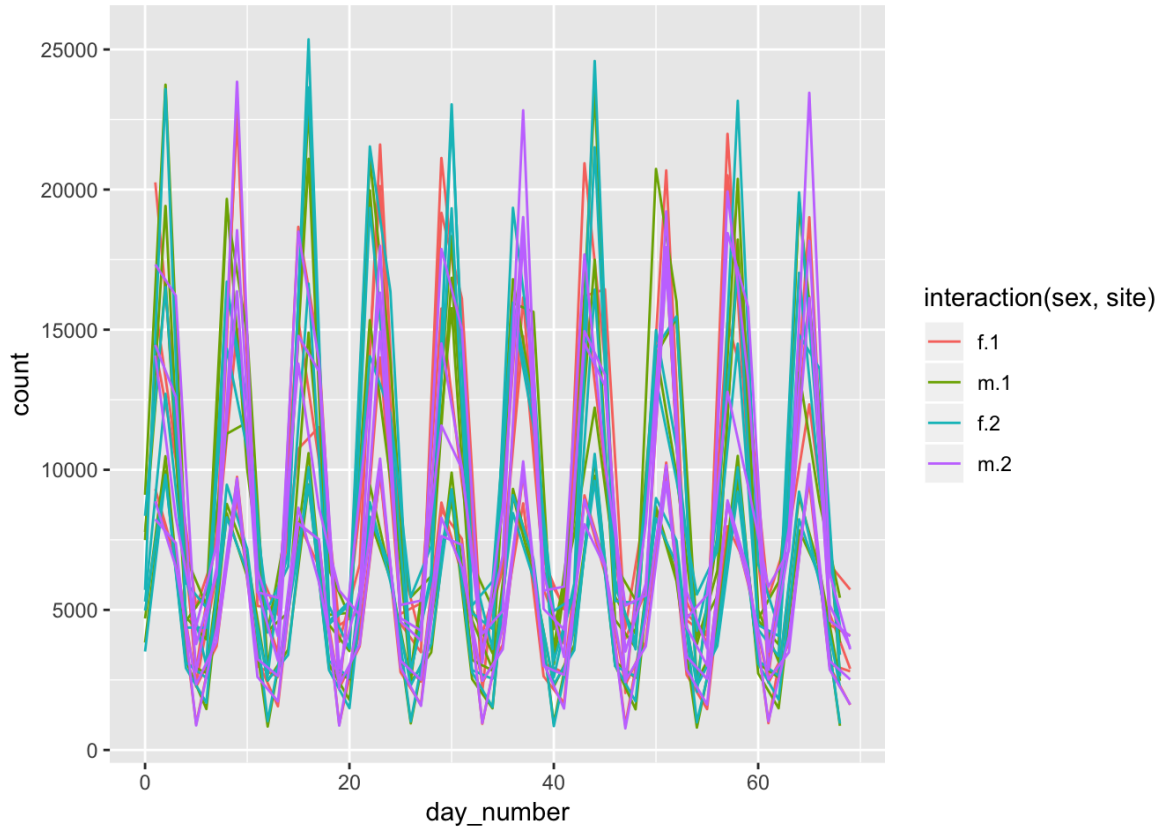
This is a bit messy (!) but should now be plotting all of our data. We can add some colours to try and make sense of whats going on. ggplot will do this automatically for us, even with lots of different levels in your data. Here we only have 2 as we are colouring the data by sex:

```
p2<-ggplot(time_series_2, aes(x=day_number, y=count, group=interaction(ts_number, sex, site), col=
sex))+geom_line()
p2
```



This hasnt really helped very much. We can easily make ggplot automatically make more colour levels for other factors, e.g. colours for each sex at each site again using that interaction() command:

```
p2<-ggplot(time_series_2, aes(x=day_number, y=count, group=interaction(ts_number, sex, site), col=
interaction(sex,site)))+geom_line()
p2
```



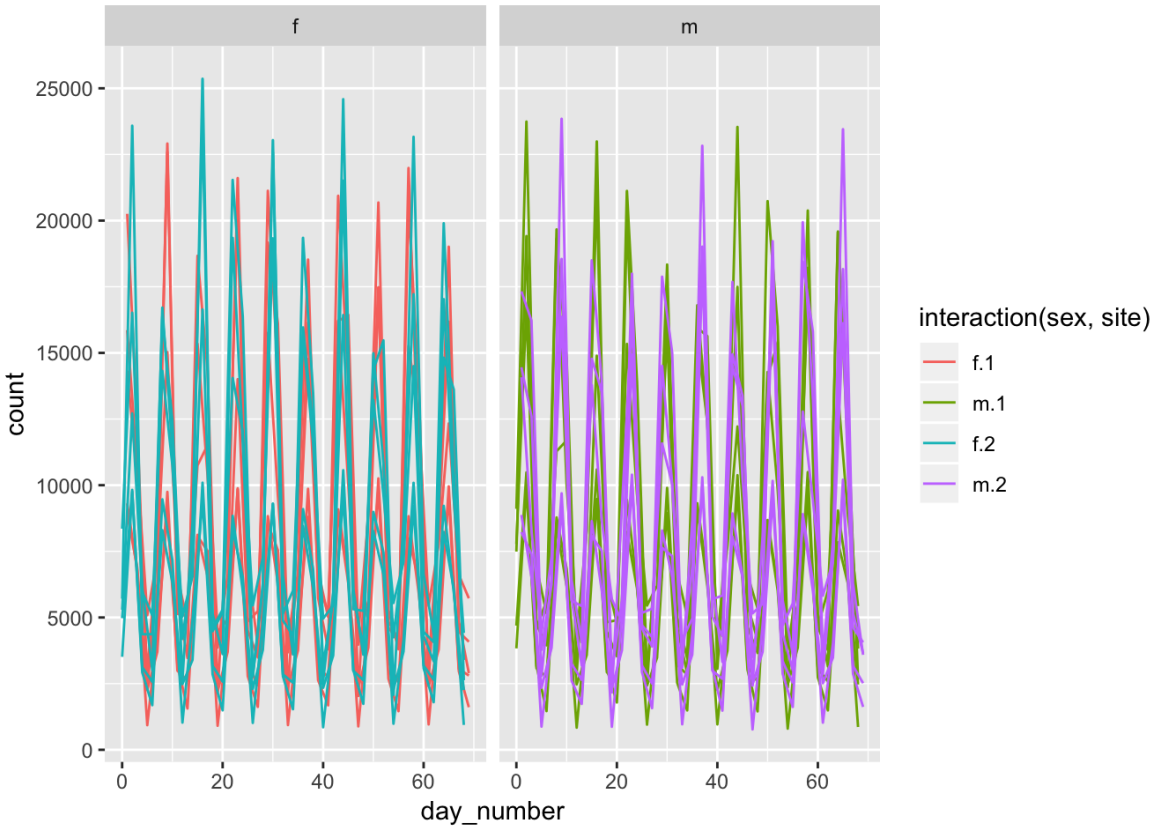
Frankly this looks horrendous and you cant see whats going on. But its useful to note that ggplot automatically makes legends which correspond to the colours (and shapes) we have in the plot, which stops you making mistakes with this. We can see that its easy to specific different levels to group by, but its almost impossible to see whats going on. ggplot has

plenty of powerful tools to help us solve this issue.

Facets

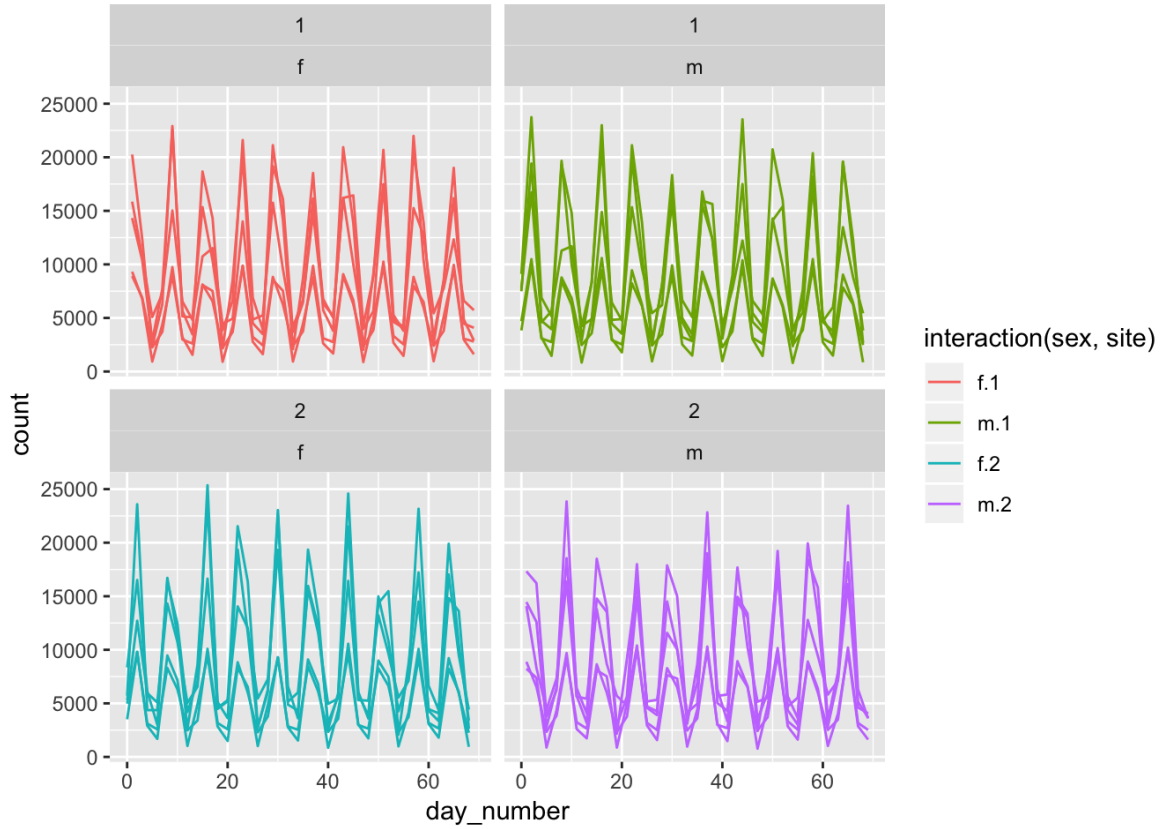
Facets in ggplot mean pannels in a plot. You can specify them either through facet_wrap or facet_grid. They do very similar things. We will use facet_wrap to make sense of our data:

```
##we will use p2 we specified above, but add an additional argument:
p2+facet_wrap(~sex)
```



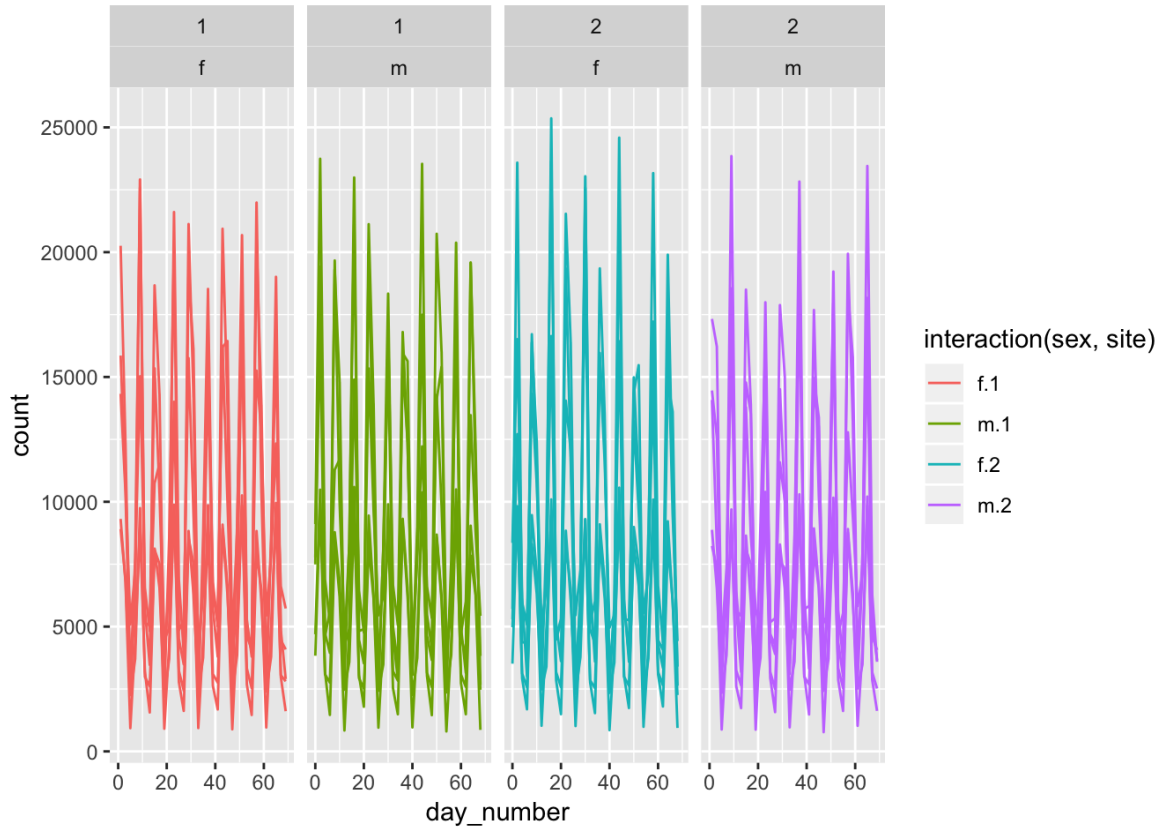
You can see this has now made 2 panel plots, one for each sex. We can see a little more clearly the dynamics, but it is still pretty messy. An easy way to make it clearer would be to split it by site too:

```
p2+facet_wrap(site~sex)
```

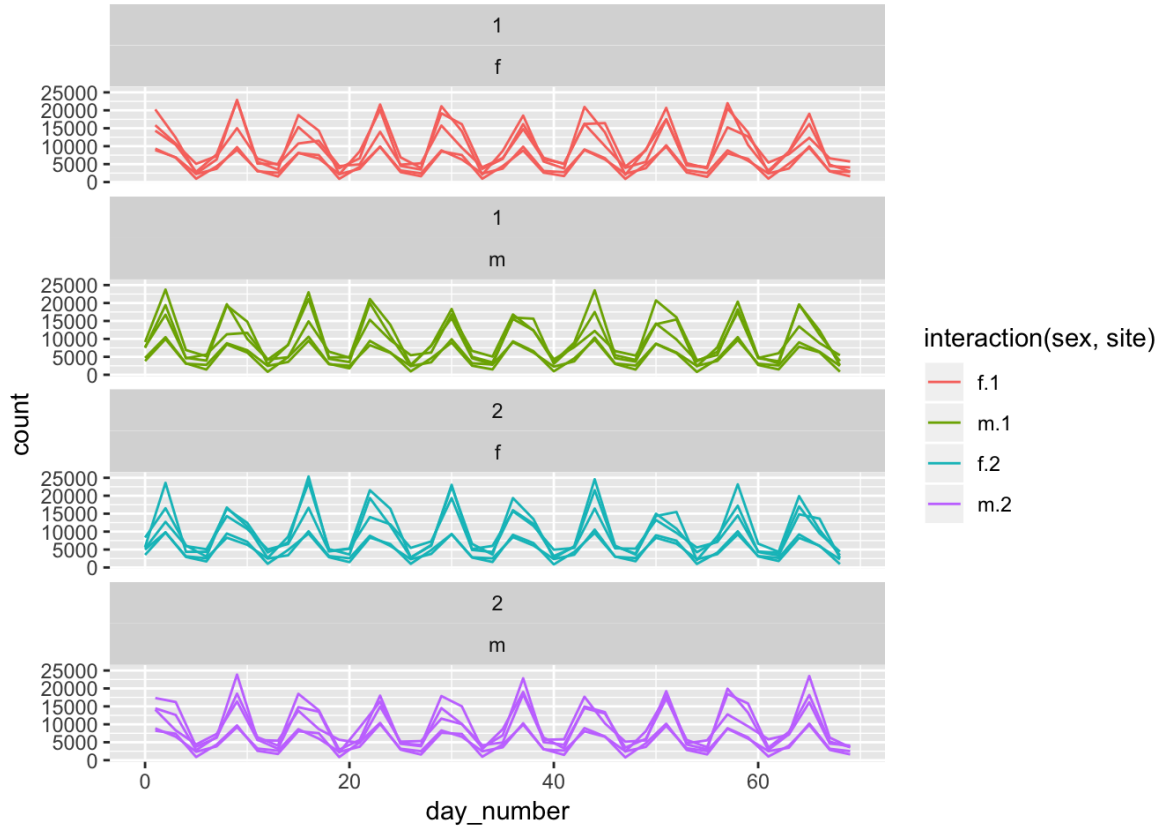


You can change the arragnement of these fairly easily:

```
##all on one row
p2+facet_wrap(site~sex, nrow=1)
```

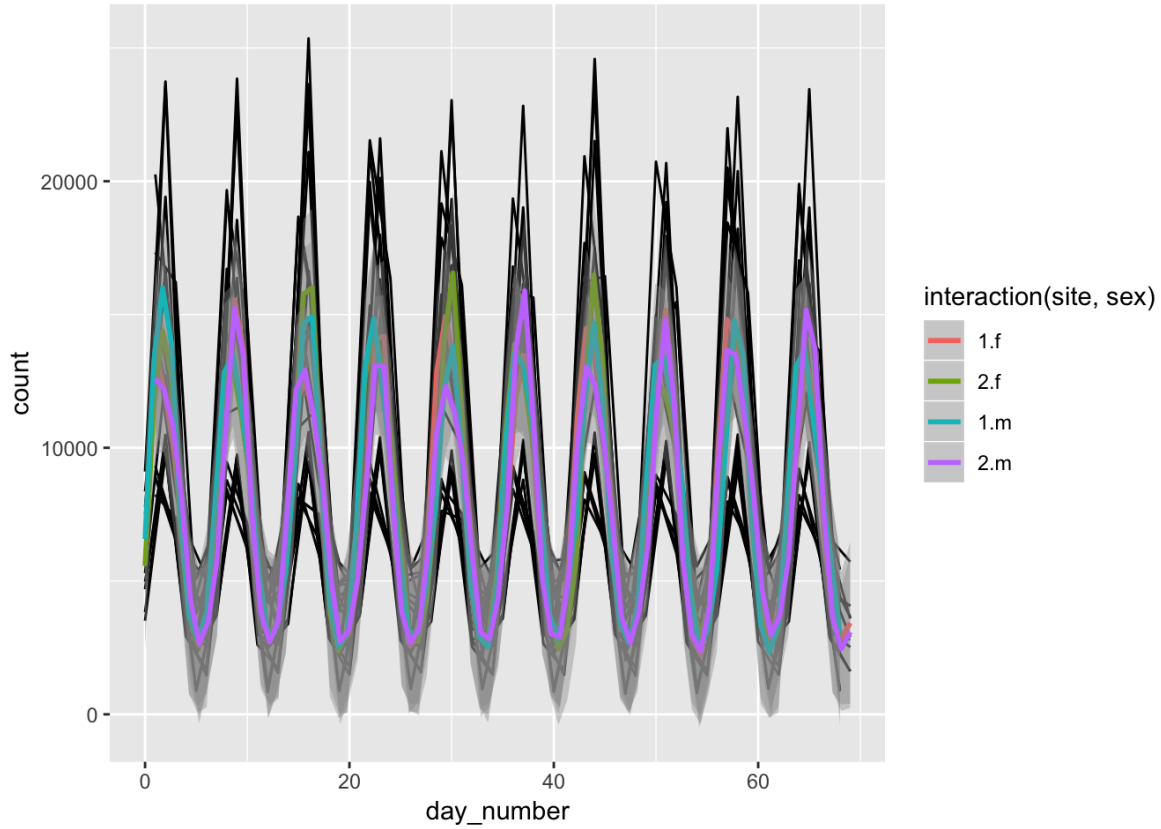



```
## all in one column
p2+facet_wrap(site~sex, ncol=1)
```

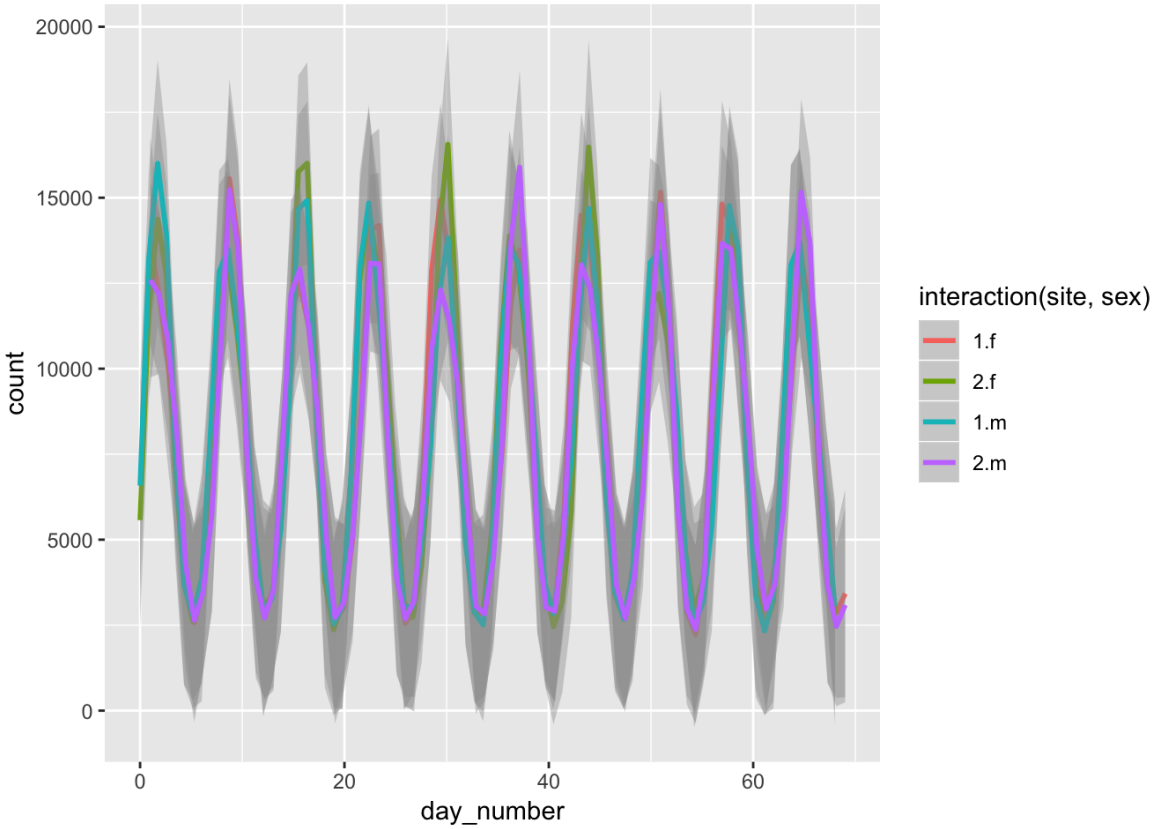


This hasn't helped hugely, as there are so many time series in each facet that its hard to tell whats going on. One alternative option could be to fit curves to this data and just show the fit of these curves, rather than all 20 populations. We can do this really easily by using geom_smooth():

```
ggplot(data=time_series_2, aes(x=day_number, y=count)) +
  geom_line(aes(group=interaction(ts_number, sex, site))) +
  geom_smooth(aes(col=interaction(site, sex)), method="loess", span = 0.1)
```

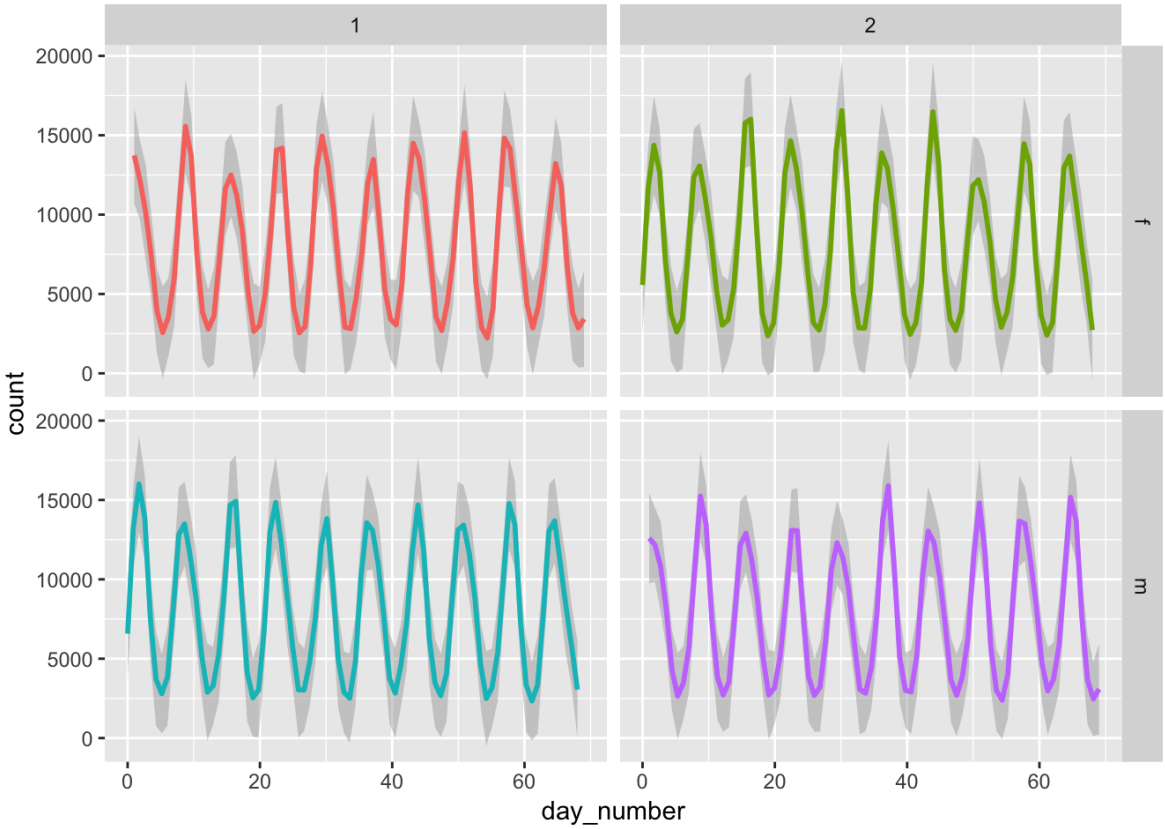


```
## or just plot the smooths by themselves:
ggplot(data=time_series_2, aes(x=day_number, y=count)) +
  geom_smooth(aes(col=interaction(site, sex)), method="loess", span = 0.1)
```



And then if we put these into a facet plot we can see whats going on (here using facet_grid). Ggplot will also automatically provide 95% CI's around the smoothed lines (the grey shading):

```
ggplot(data=time_series_2, aes(x=day_number, y=count)) +
  geom_smooth(aes(col=interaction(site, sex)), method="loess", span = 0.1) +
  facet_grid(sex~site) +
  theme(legend.position = "none")
```



Its really useful to note that ggplot doesnt just fit smoothed lines, but can also fit linear regressions etc. Lets take a look at a new data set to do that. This is the iris data set included in R:

```
library(datasets)
data(iris)
summary(iris)
```

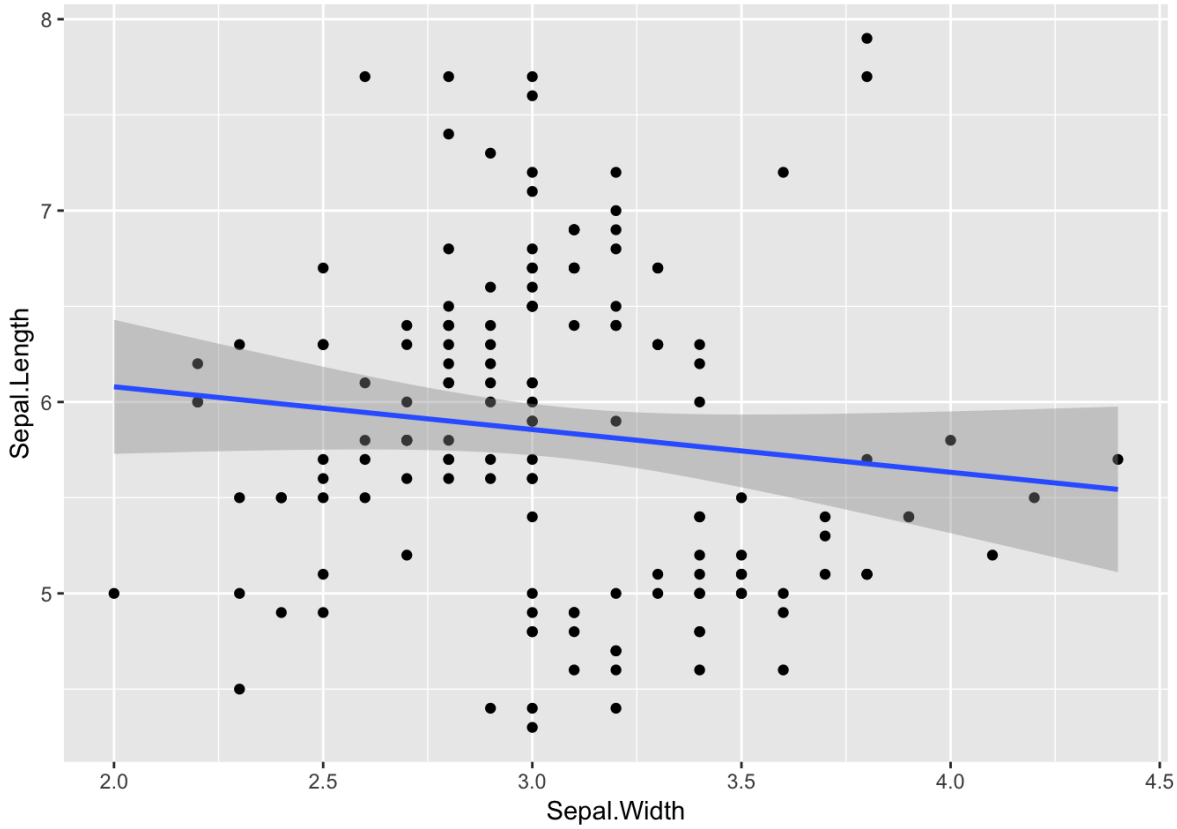
```
##    Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

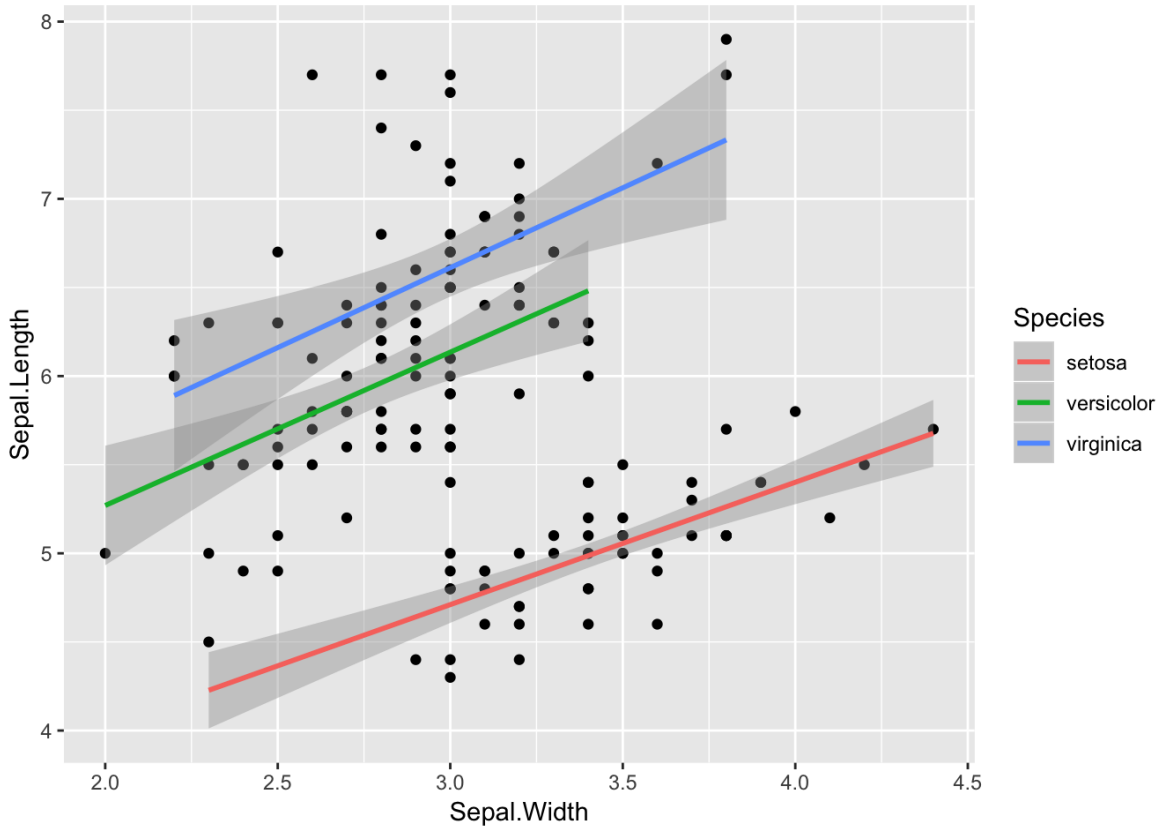
Let say we want to plot sepal width against sepal length, and fit a linear regression to that. We can do that by doing:

```
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length)) + geom_point() + geom_smooth(method=lm)
```



And easily split that up so that we treat each of the three species seperately:

```
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length)) + geom_point() + geom_smooth(aes(group=Species, c
ol=Species), method=lm)
```



Alternative plots without reshaping my data

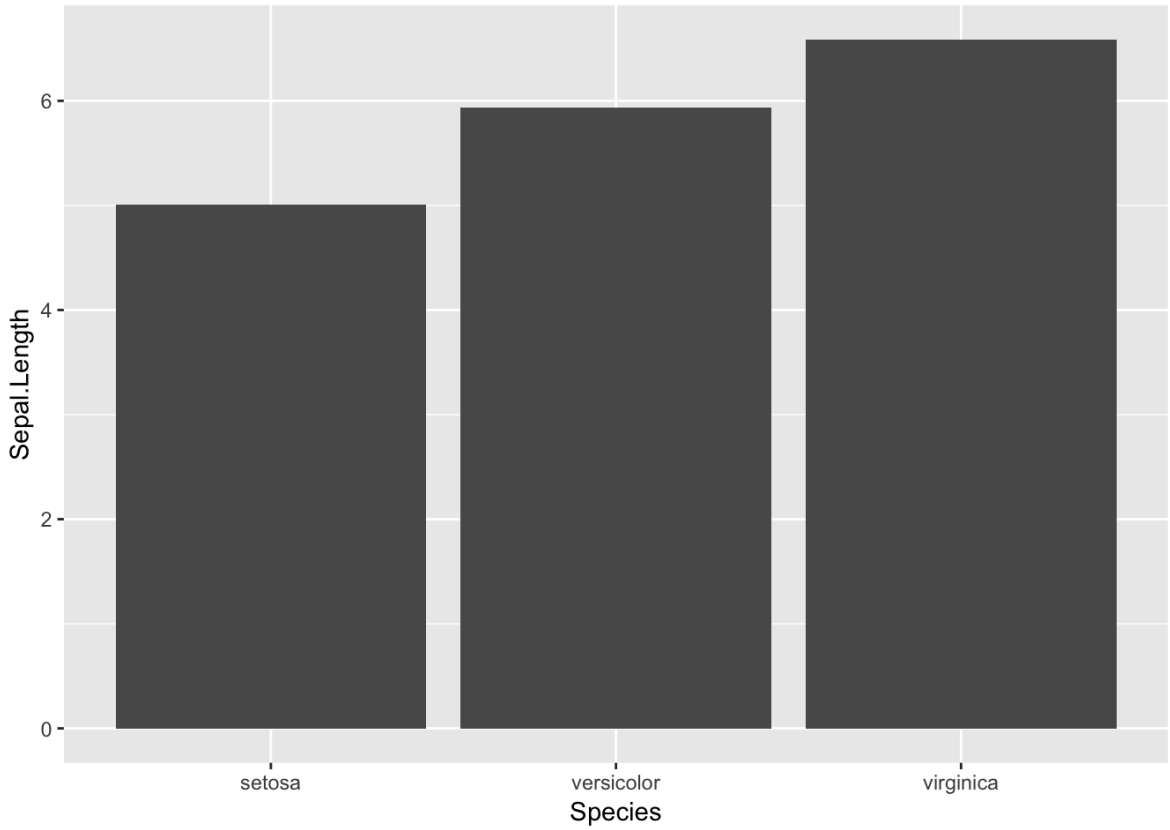
One great thing about ggplot is its ability to produce a new plot without you having to do all the data reshaping. Lets take a look at the iris data set to demonstrate this.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Now, lets say we want to plot the mean sepal length for each species. We don't have to calculate this and then plot it, we can do it all in ggplot:

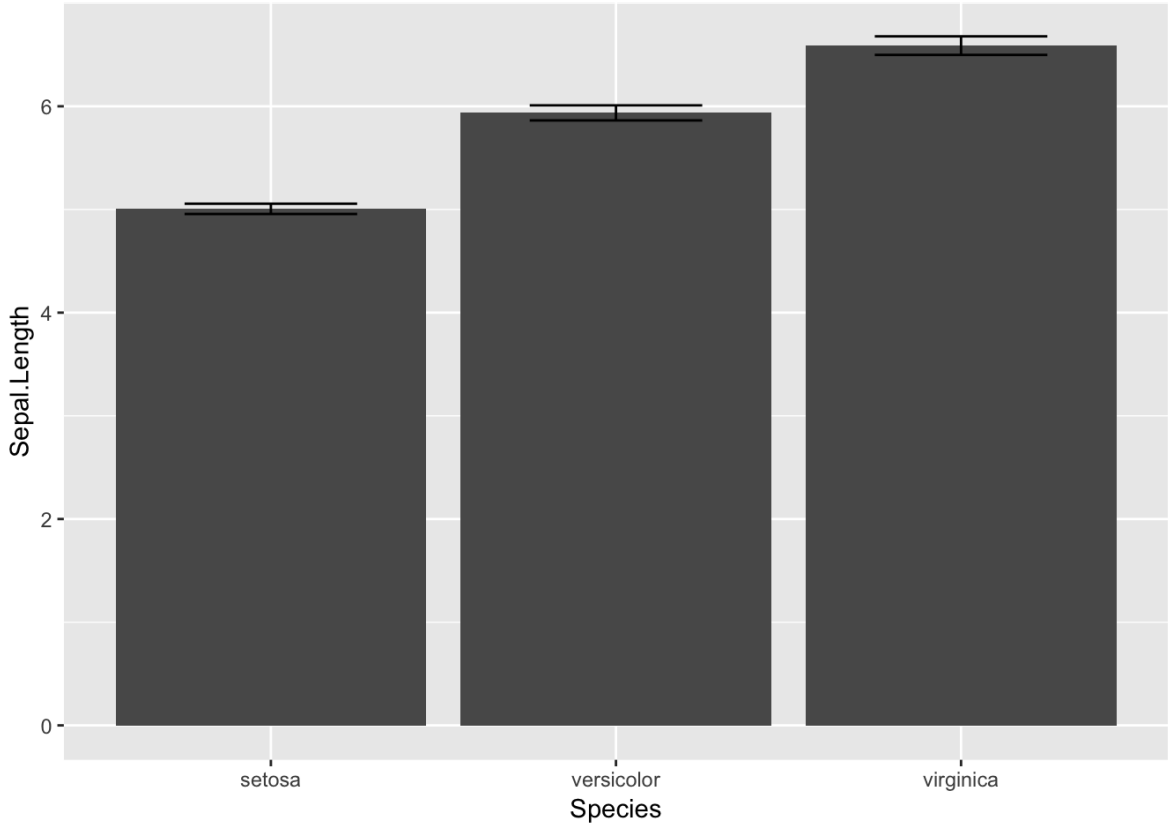
```
ggplot(iris, aes(x=Species, y=Sepal.Length)) +
    stat_summary(fun.y = mean, geom = "bar")
```




```
##note, you can also replace the fun.y =  with lots of basic stat functions in R (max, min, media
n, etc)
ggplot(iris, aes(x=Species, y=Sepal.Length)) +
    stat_summary(fun.y = max, geom = "bar")
```

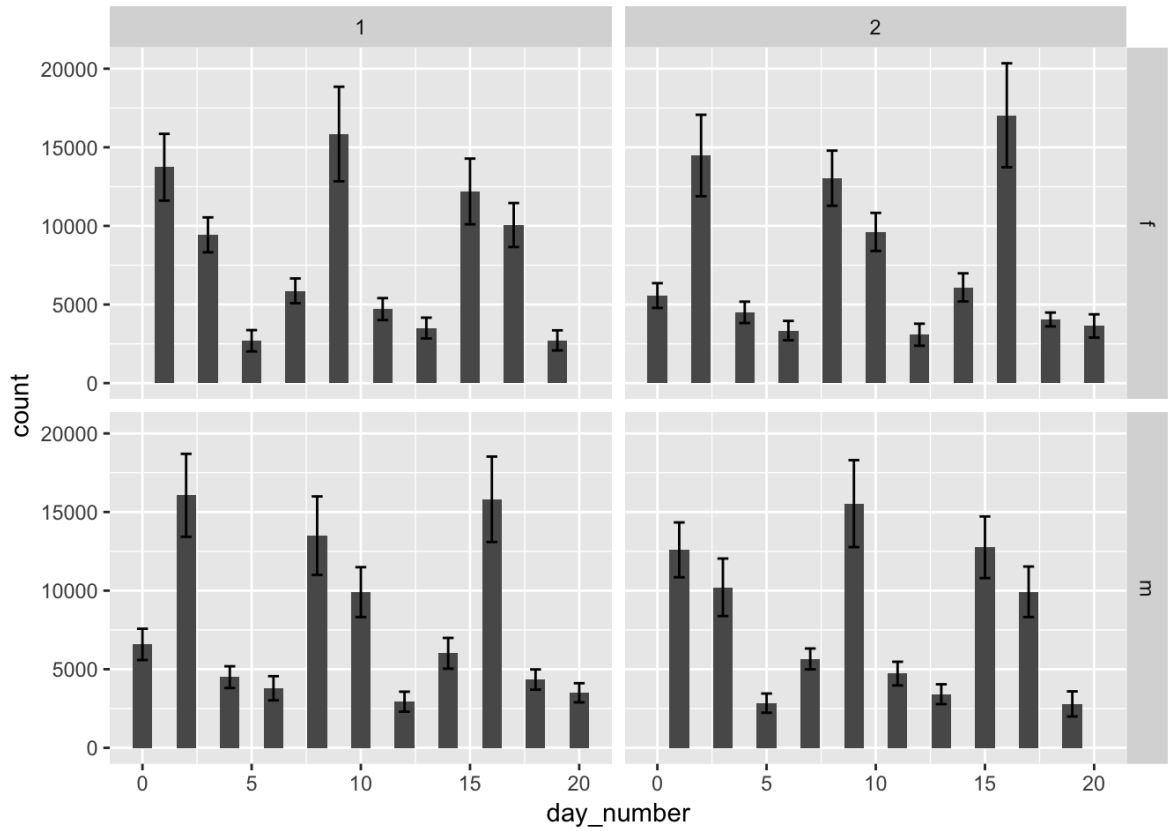
We can also really easily add error bars to this:

```
p5 <- ggplot(iris, aes(x=Species, y=Sepal.Length)) +
      stat_summary(fun.y = mean, geom = "bar") +
      stat_summary(fun.data = mean_se, geom = "errorbar", width=0.5)

p5
```



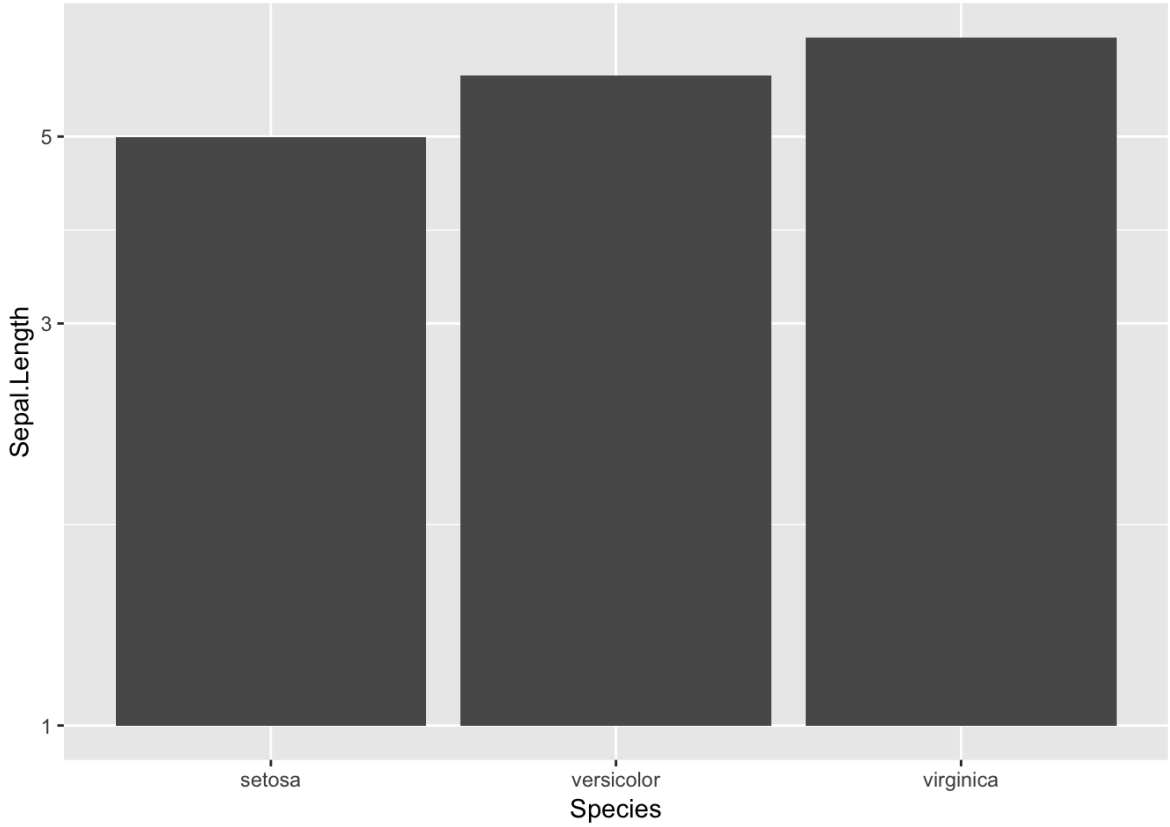
As you can imagine, this gets more and more useful the larger your data set gets. ggplot will audomatically do all these calculations and add in your error bars across different facets. As a silly example:

```
small_example<-subset(time_series_2, day_number<=20)
psmall<- ggplot(small_example, aes(x=day_number, y=count)) +
        stat_summary(fun.y = mean, geom = "bar") +
        stat_summary(fun.data = mean_se, geom = "errorbar", width=0.5) +
        facet_grid(sex~site)
psmall
```



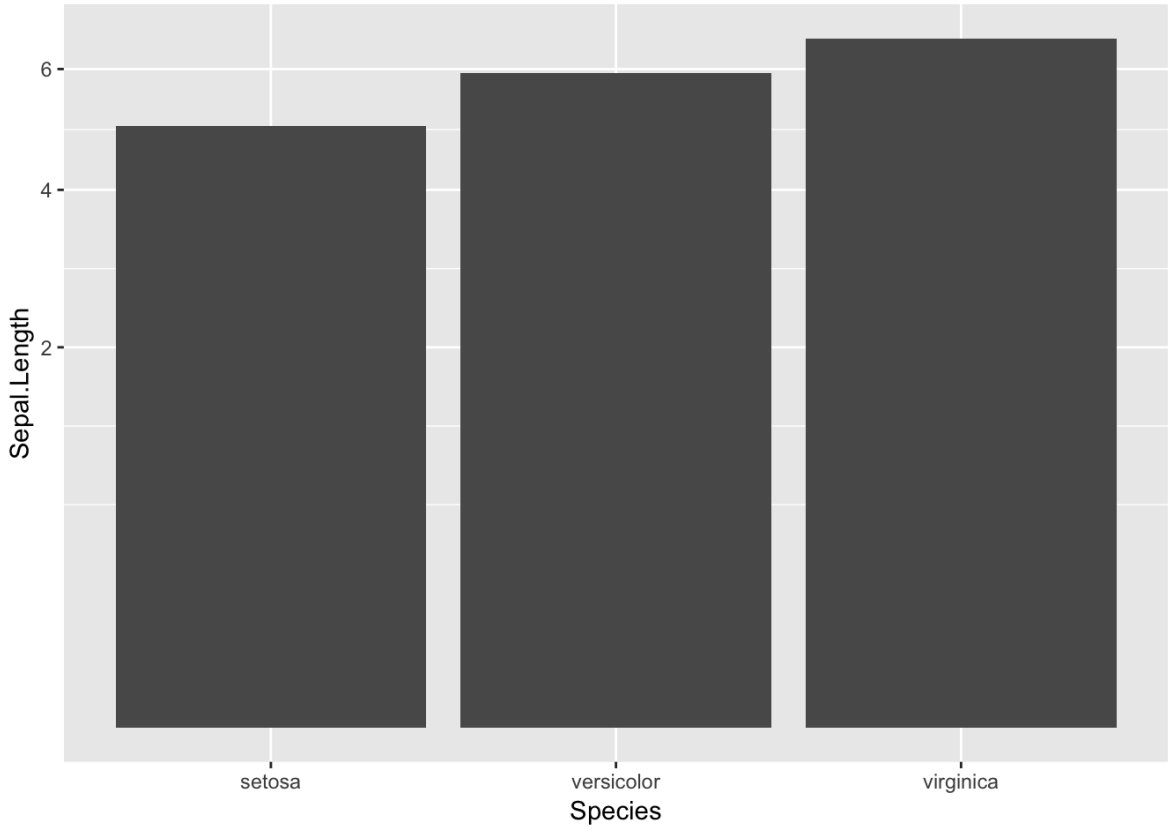
Related to this is ggplot's ability to transform your data on the fly, without making any changes to your original dataframe. For example, to logging the count data from our earlier plot. The bonus is this doesnt alter the original data in any way:

```
p10 <- ggplot(iris, aes(x=Species, y=Sepal.Length)) +
       stat_summary(fun.y = mean, geom = "bar")

p10 + scale_y_continuous(trans='log10') ##or scale_x_continuous(trans='log10')
```



Or square root:

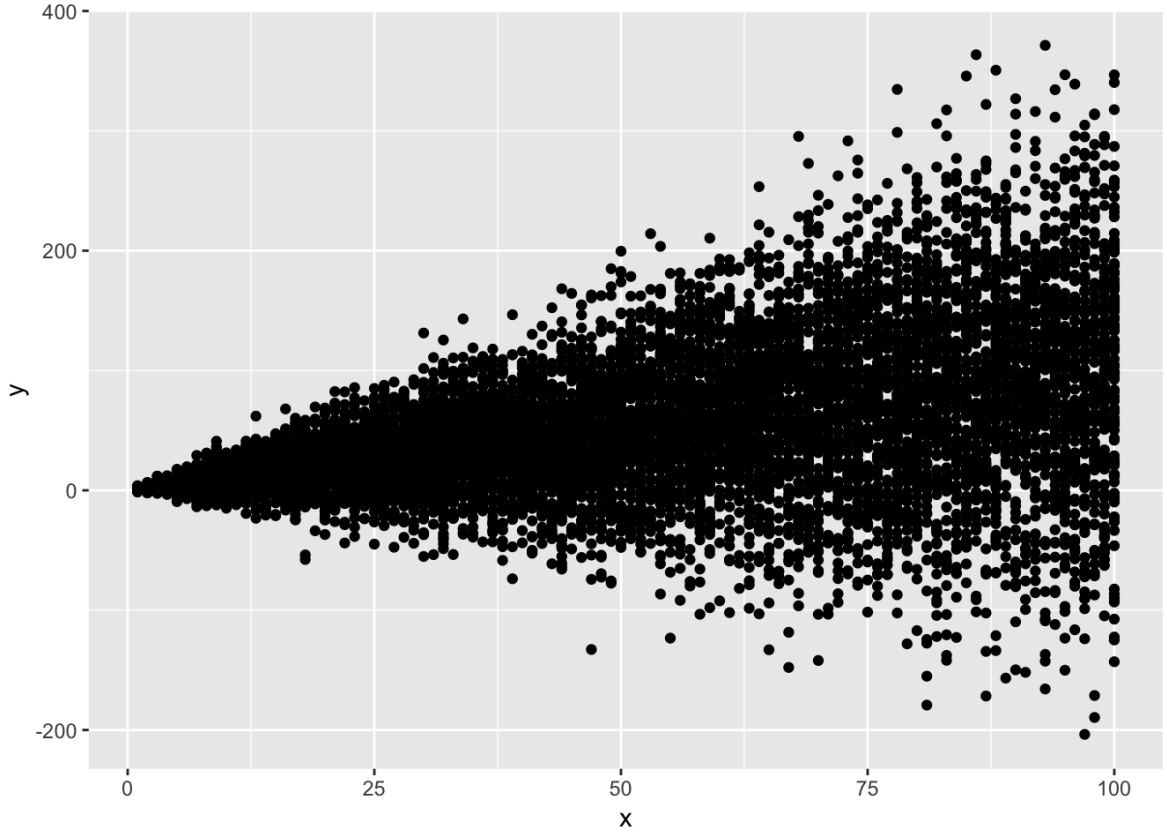
```
p10 + scale_y_continuous(trans='sqrt')
```



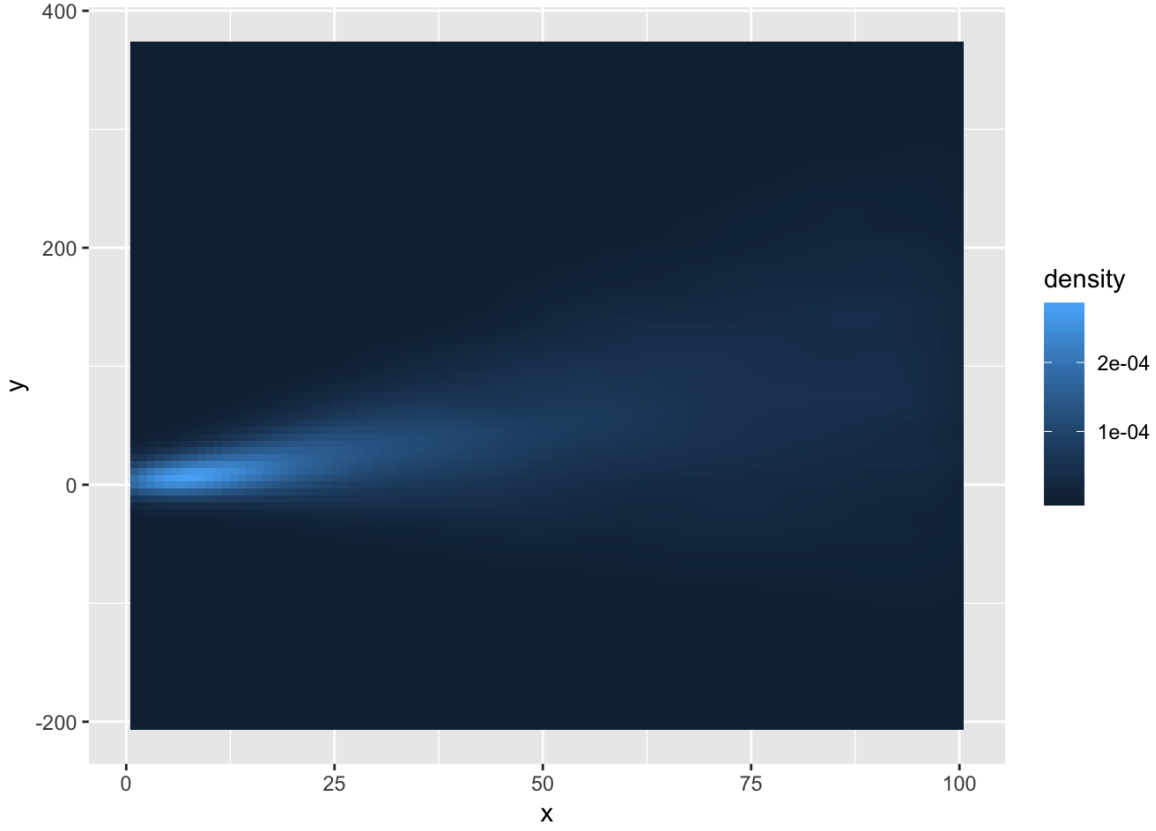
Heat maps

Because ggplot uses the same basic arguments to form graphics, it is really easy to swap between different plot types. For example, for some plots the number of points being plotted is just too high, and you cant easily visalise whats going on. Heat maps are a nifty way of visualising this, achieved through a few easy changes in ggplot. We will simulate some data to visalise this:

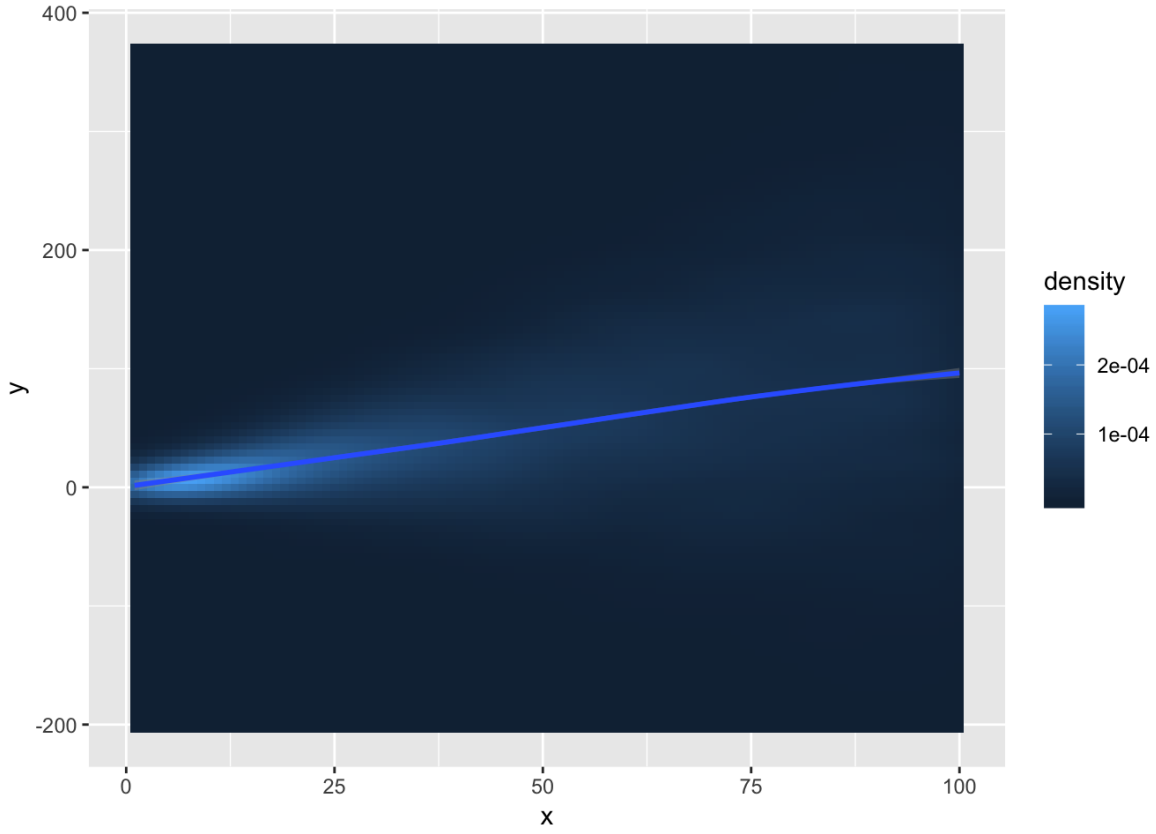
```
x=rep(c(1:100), 100)
y=x+rnorm(length(x), 0, x)
dd<-data.frame(x,
               y)

##a normal scatter plot
ggplot(dd, aes(x=x, y=y))+geom_point()
```




```
##and a density plot
p2<-ggplot()+stat_density2d(data=dd,  aes(x=x, y=y, fill=..density..),contour=F, geom="tile", n=10
0)
p2
```



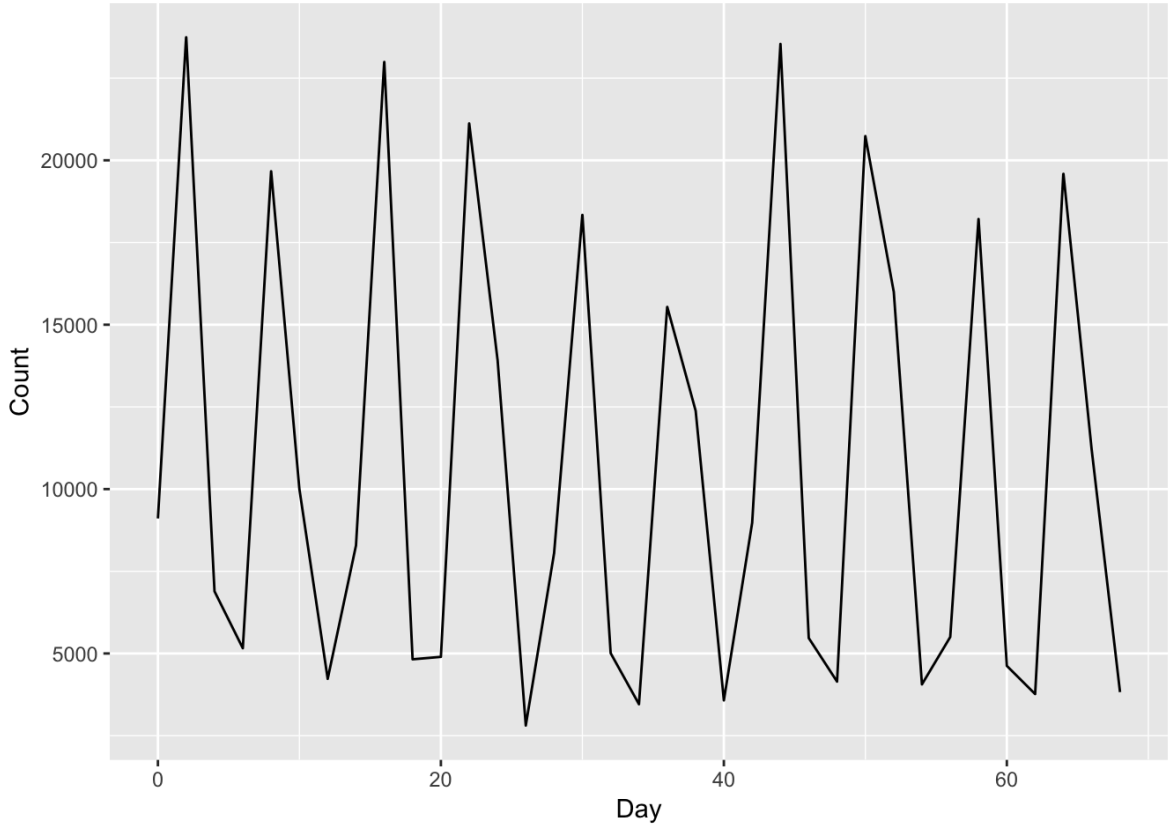

```
##and add a trend line:

p2+stat_smooth(data=dd,  aes(x=x, y=y), method="loess")
```



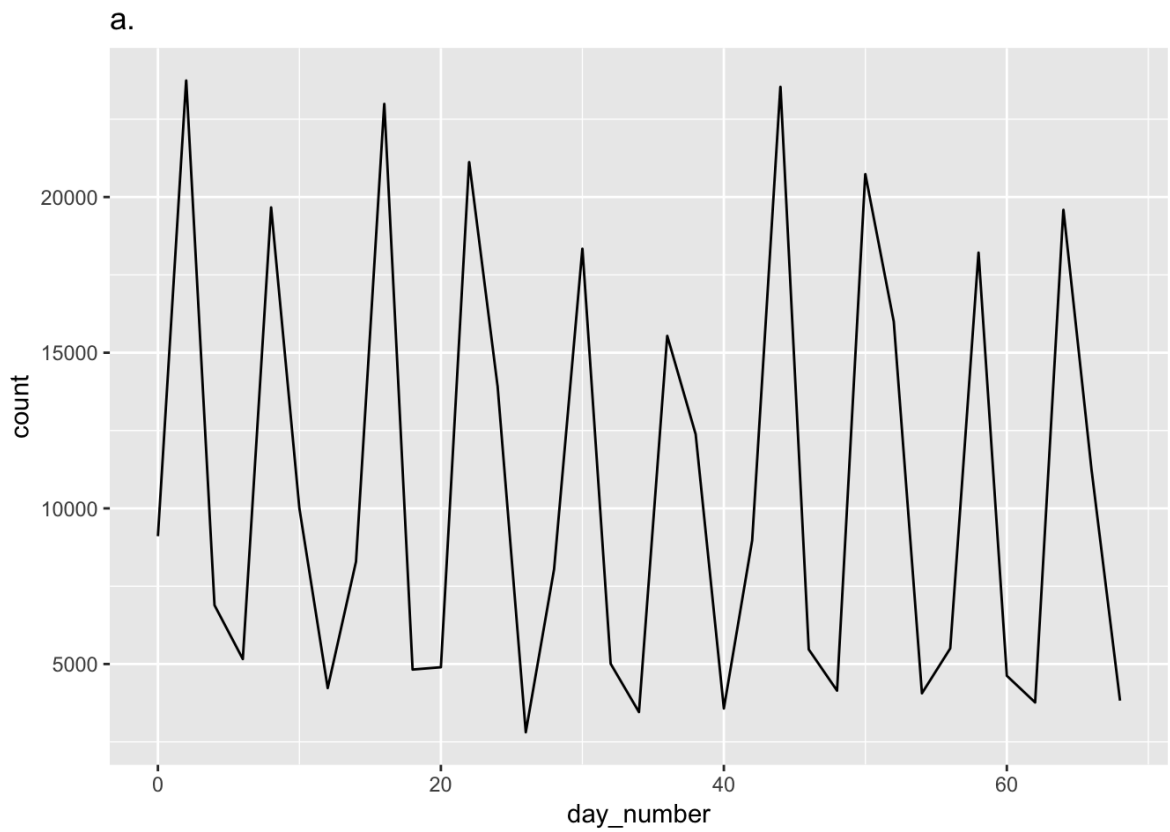
Make my plot look pretty

Below are some quick tips for changing the basic aesthetics of a plot.

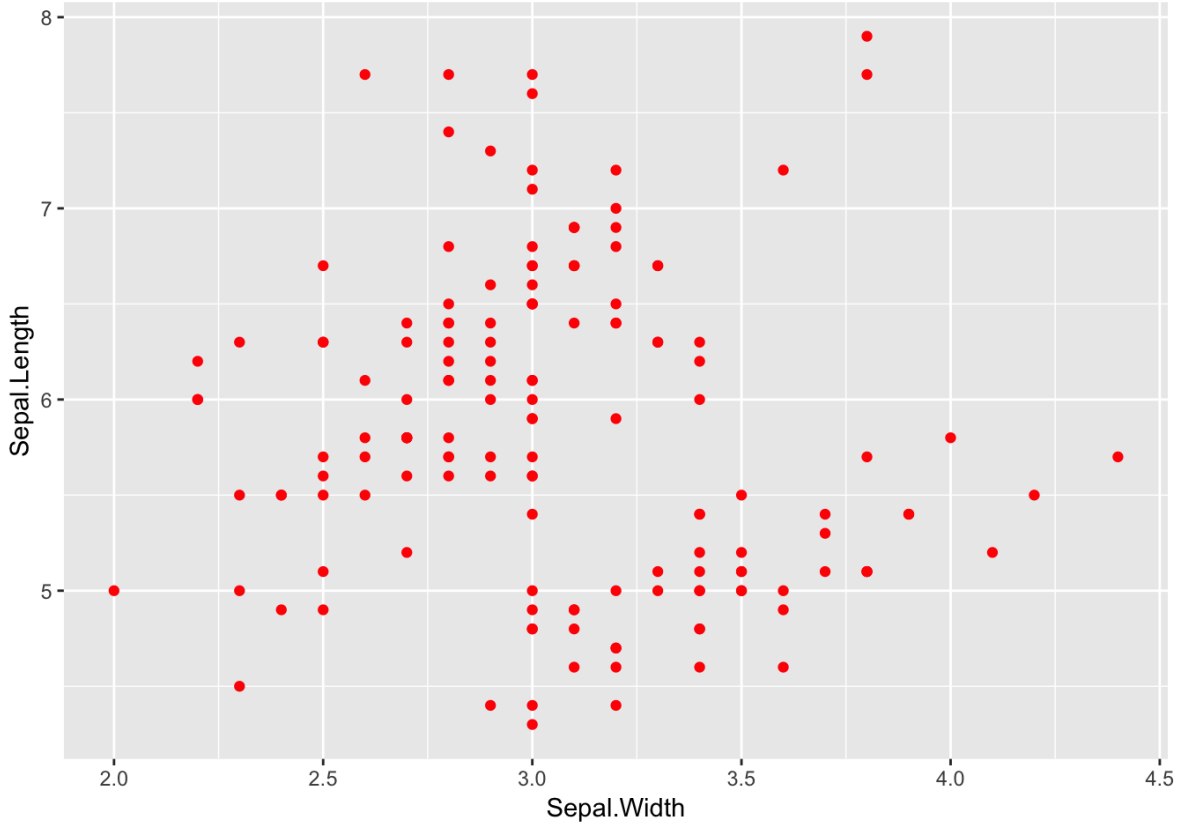
```
##setting x and y labels
p1+ylab("Count")+xlab("Day")
```



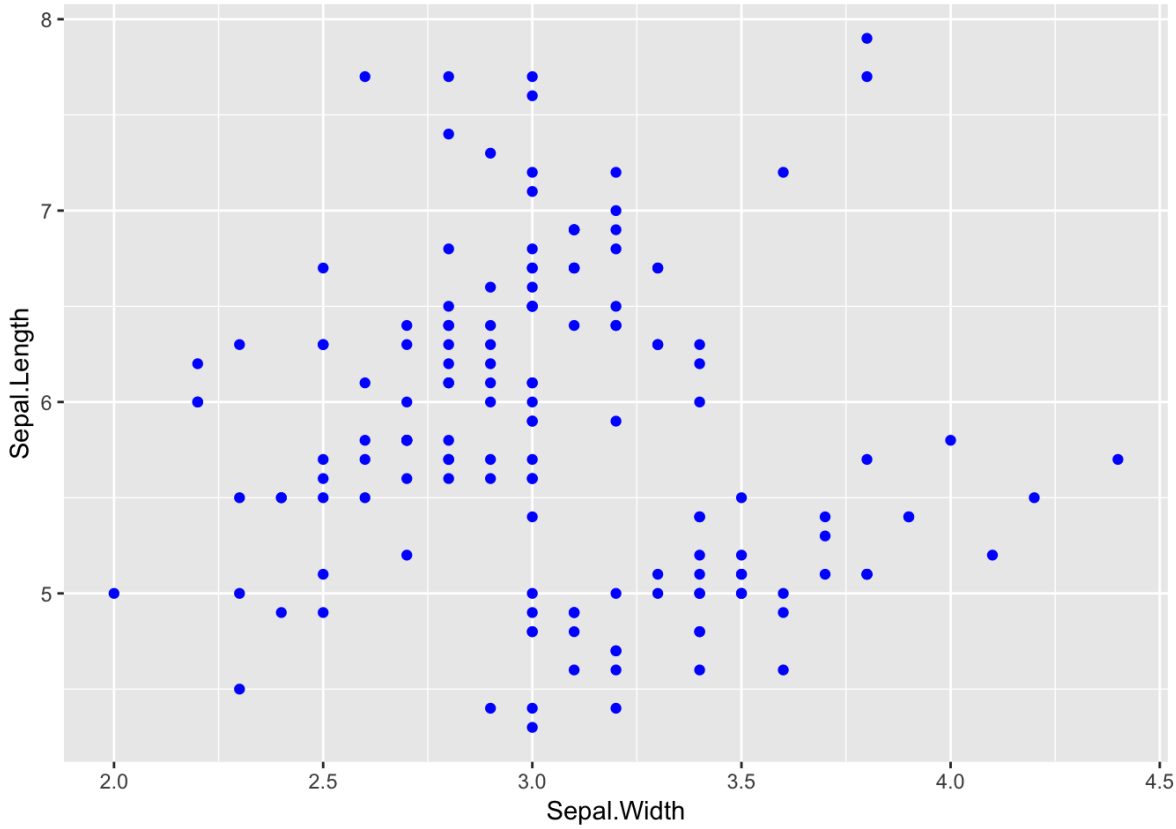

```
##add a title
p1+ggtitle("a.")
```




```
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length)) + geom_point(col="red")
```

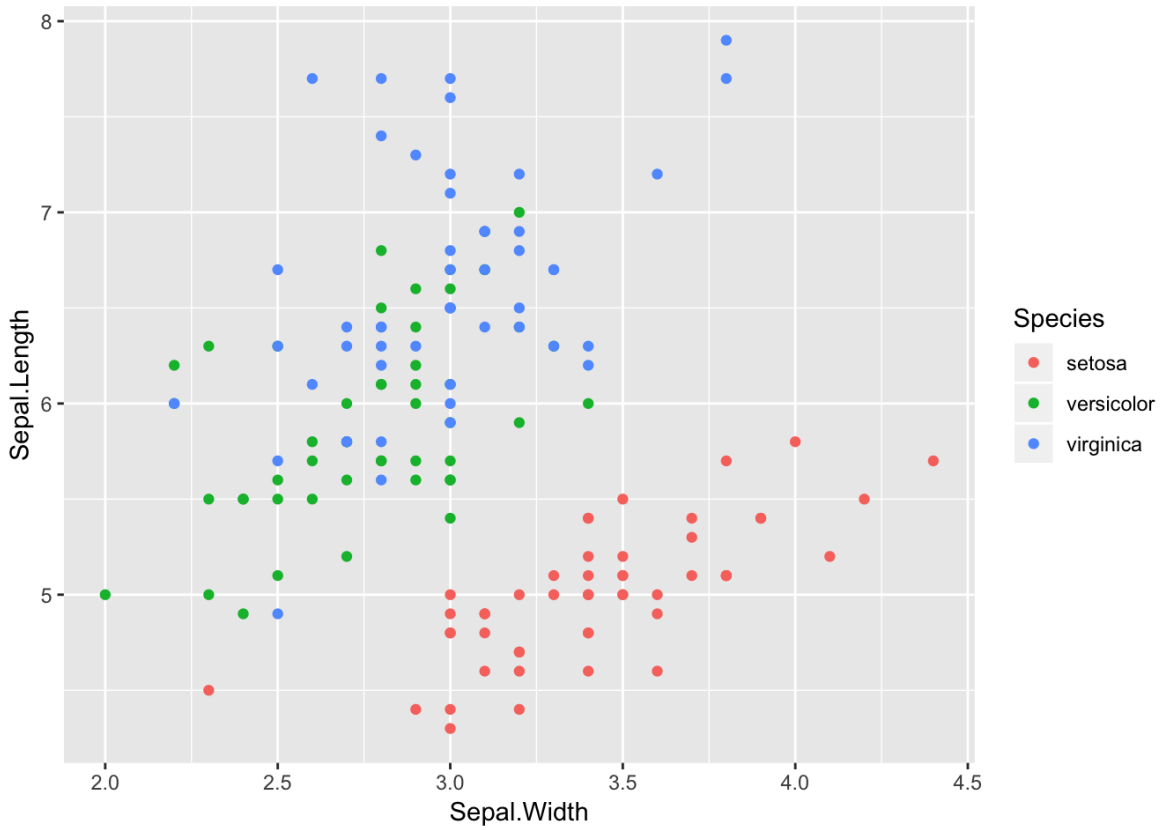



```
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length)) + geom_point(col="blue")
```



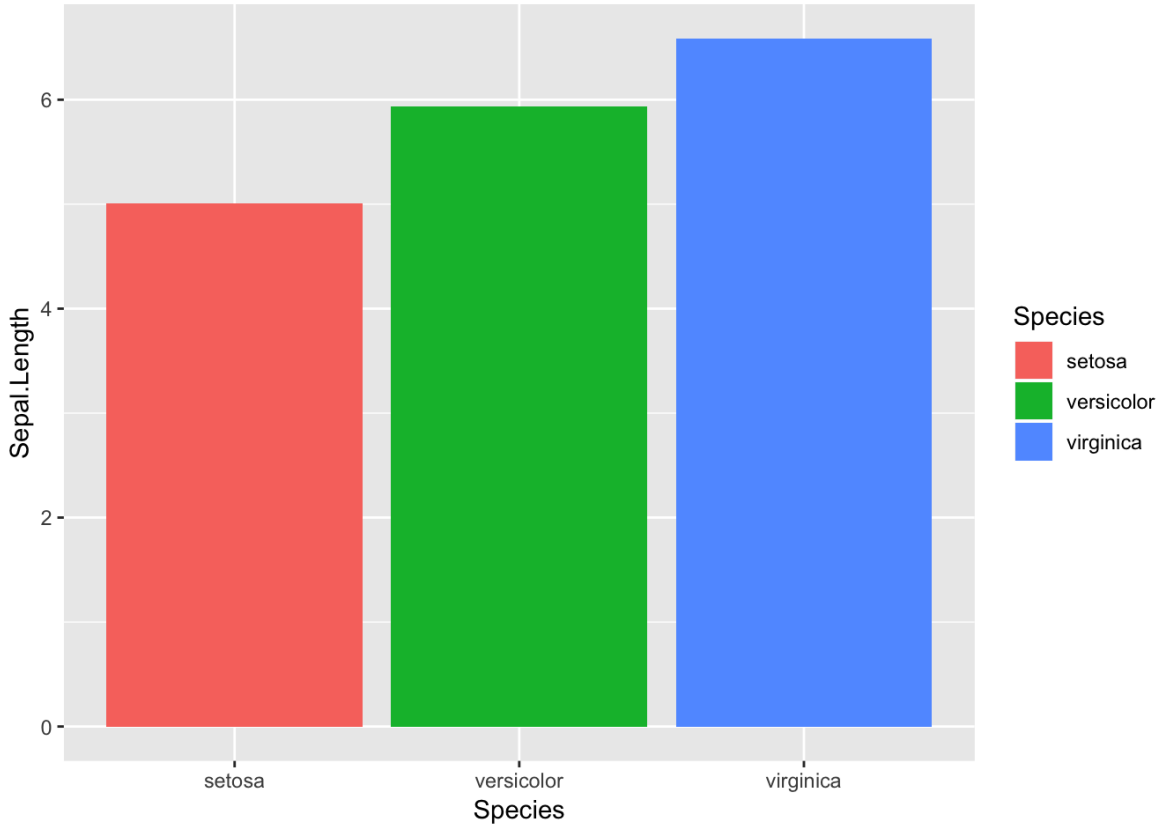
Rather than specifying a single colour, we can allow ggplot to colour the points automatically, say be species:

```
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length)) + geom_point(aes(col=Species))
```



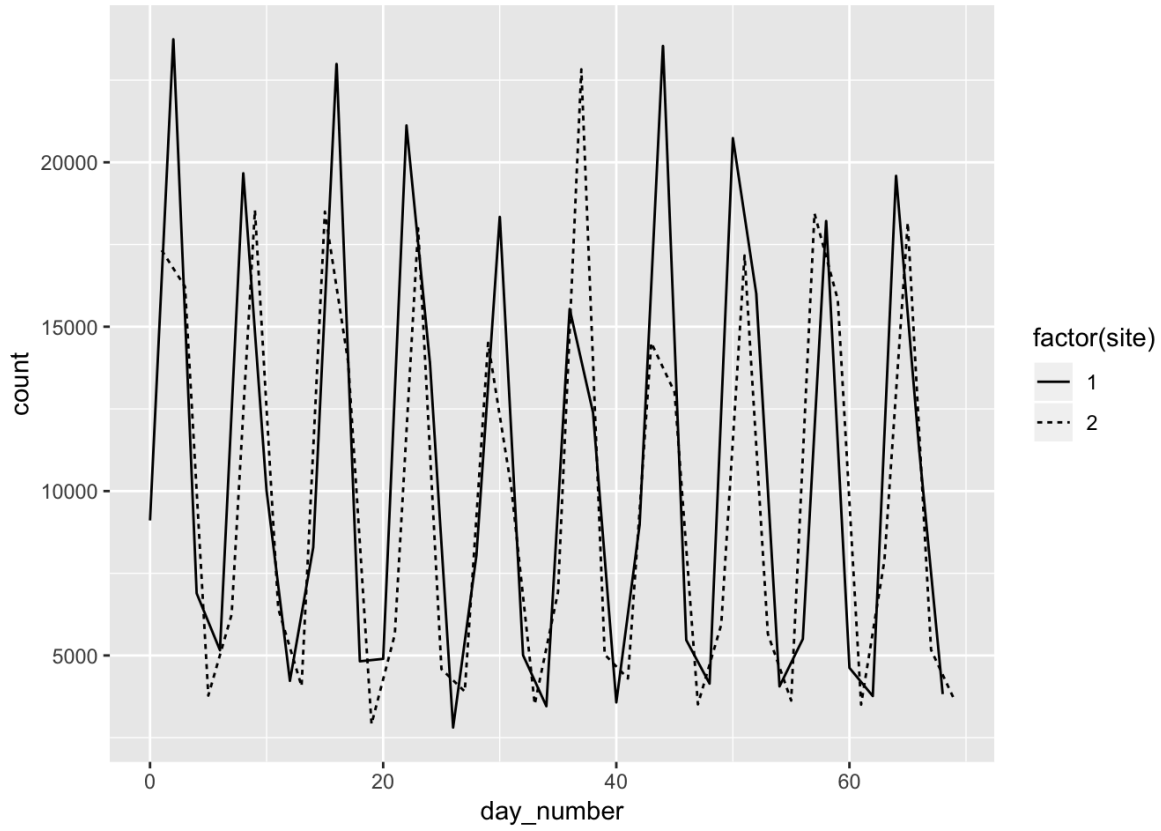
Or in a bar plot we can change the fill. Remember when we are using the data to set the colours/shapes/line types we need to specify where ggplot shoud look using aes():

```
p5 <- ggplot(iris, aes(x=Species, y=Sepal.Length)) +
      stat_summary(aes(fill=Species), fun.y = mean, geom = "bar")

p5
```



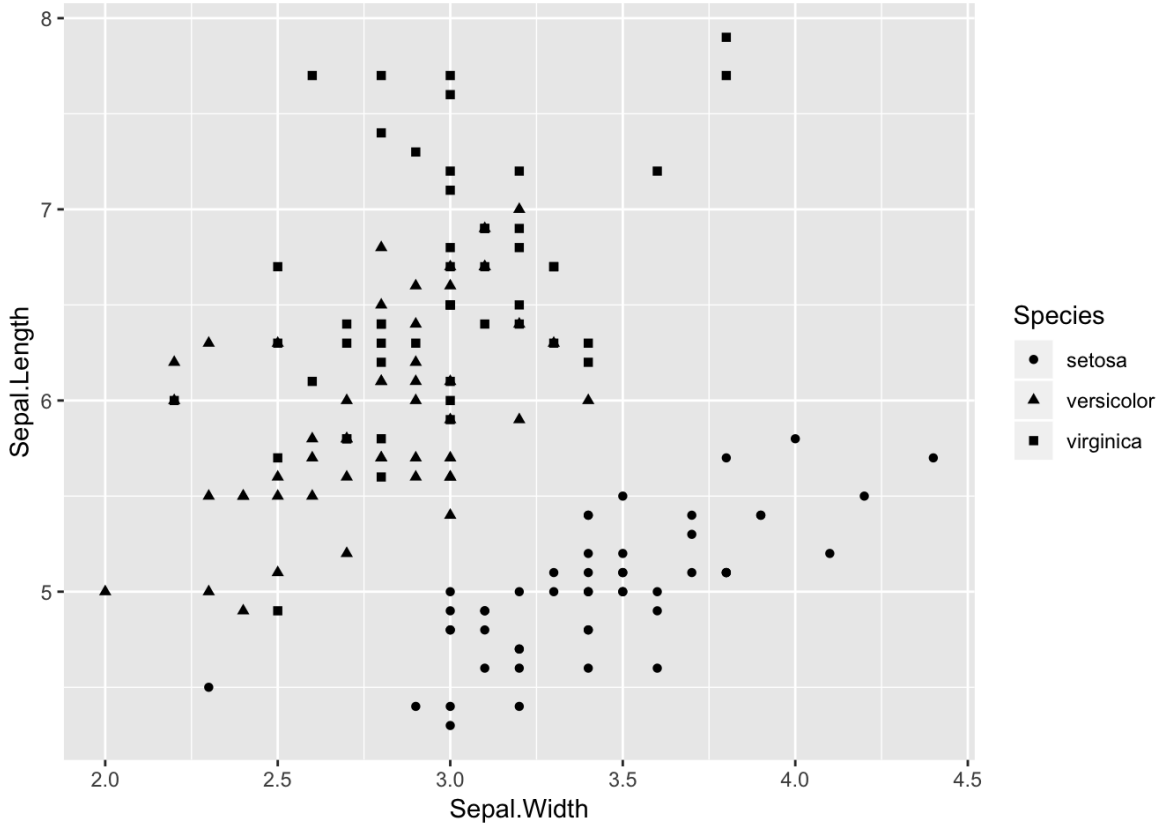
We can tell ggplot to automatically differentiate our plots my line type too:

```
ggplot(subset(time_series_2, ts_number==1 & sex=="m"), aes(x=day_number, y=count, group=interactio
n(ts_number, site), linetype=factor(site)))+geom_line()
```



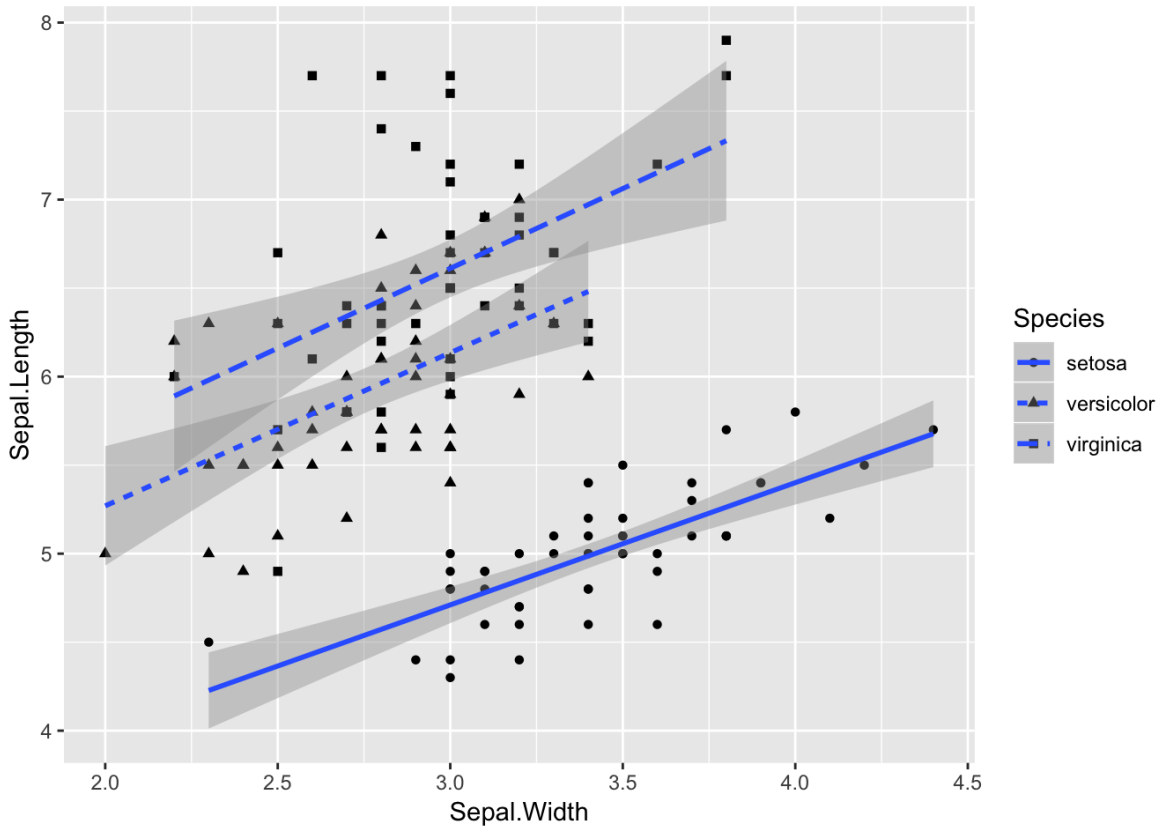
and point types:

```
ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length)) + geom_point(aes(shape=Species))
```



Or both:

```
p6.5 <- ggplot(iris, aes(x=Sepal.Width, y=Sepal.Length)) + geom_point(aes(shape=Species)) + geom_s
mooth(aes(group=Species, linetype=Species), method=lm)
p6.5
```



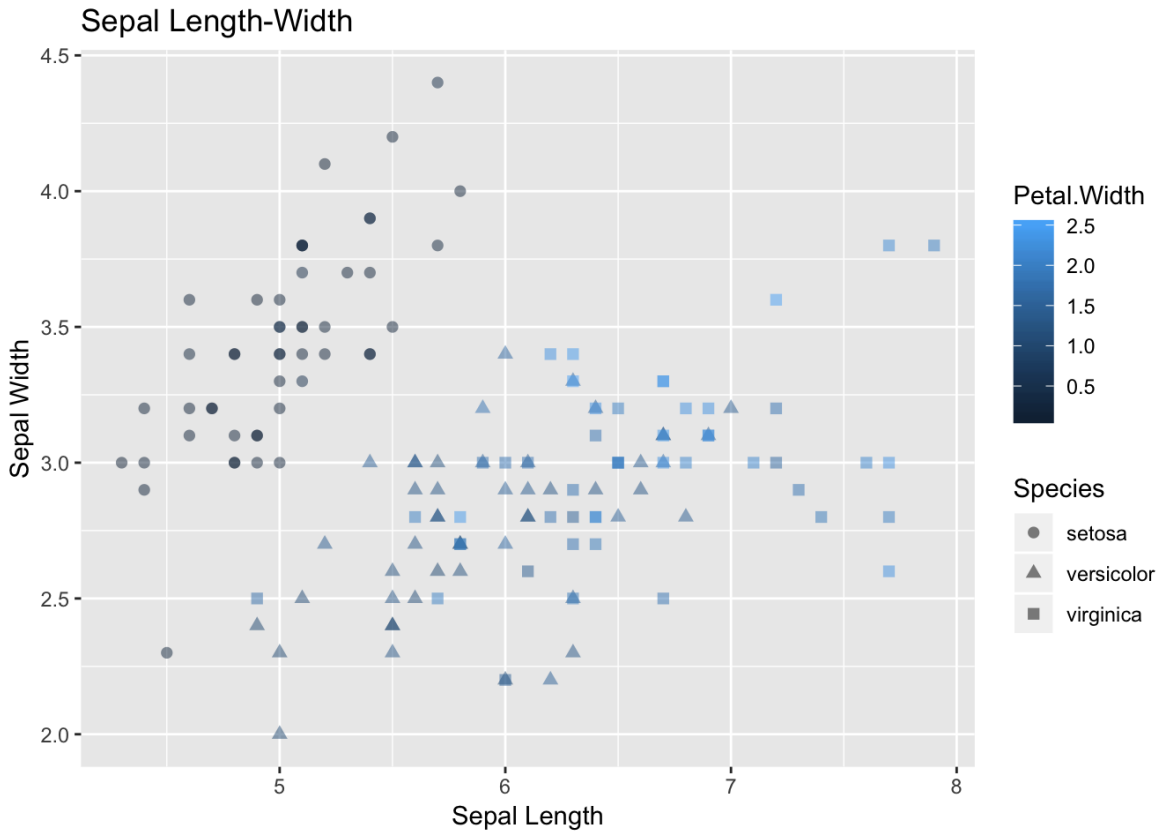
And the great thing is that ggplot will update this all automatically, even when you have lots of factors (e.g. if we had 10 sites for the above examples).

The ability to automatically change colours, points, and lines really comes into its own when the plots become more complex. Here is a quick example (modified) from Jihui Lee:

```
library(datasets)
data(iris)
summary(iris)
```

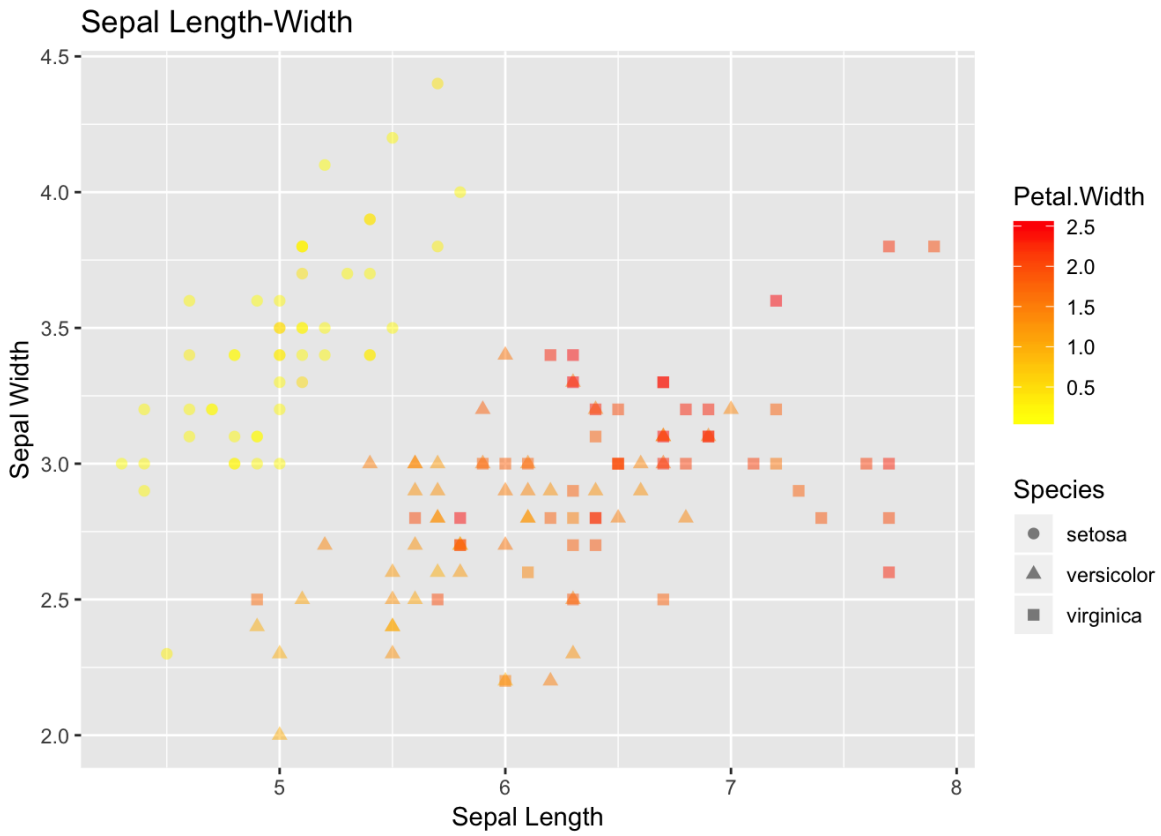
```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

```
scatter <- ggplot(data=iris, aes(x = Sepal.Length, y = Sepal.Width)) + xlab("Sepal Length") + ylab
("Sepal Width") + ggtitle("Sepal Length-Width")

scatter <- scatter + geom_point(aes(color = Petal.Width, shape = Species), size = 2, alpha = I(1/2
))
scatter
```



You can also easily specify your own colours (rather than the base ones that ggplot produces). So for the example above we could:

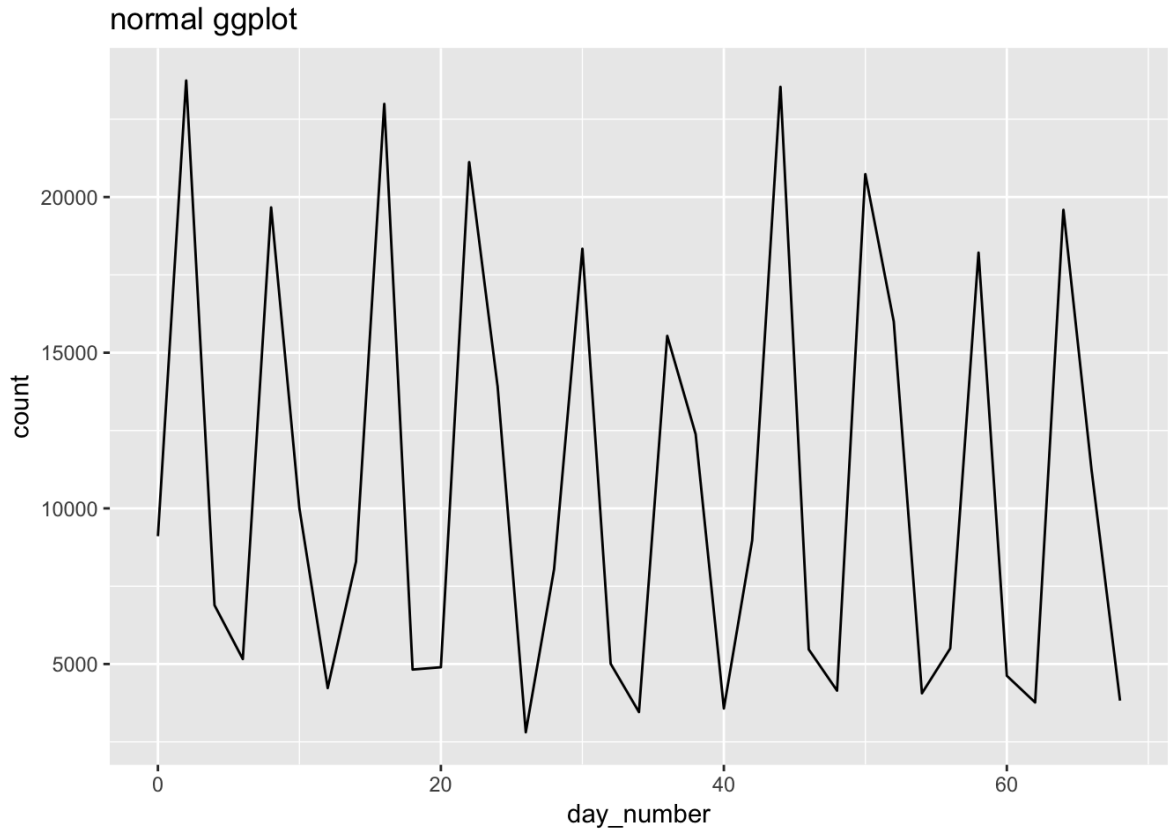
```
scatter <- scatter + scale_color_gradient(low = "yellow", high = "red")
scatter
```



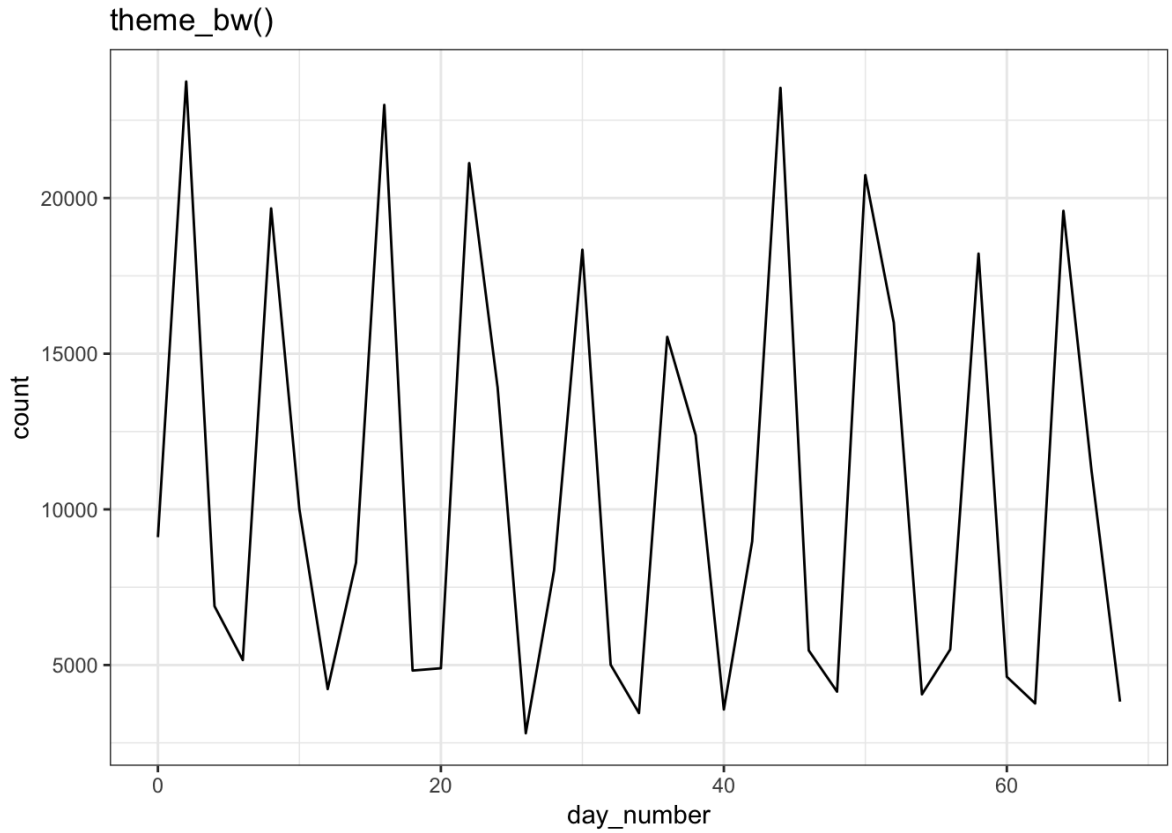
And ggplot will helpfully update the legend and everyting in line with your new colour selection. Nice!

One of the simplest wasy of sprucing up your plot is changing the theme. Themes control the look of the plot background and axis, not the points or lines you are plotting. There are a bunch of preset ones (and you can heavily modify them yourself). I tend to use theme_bw() or theme_classic():

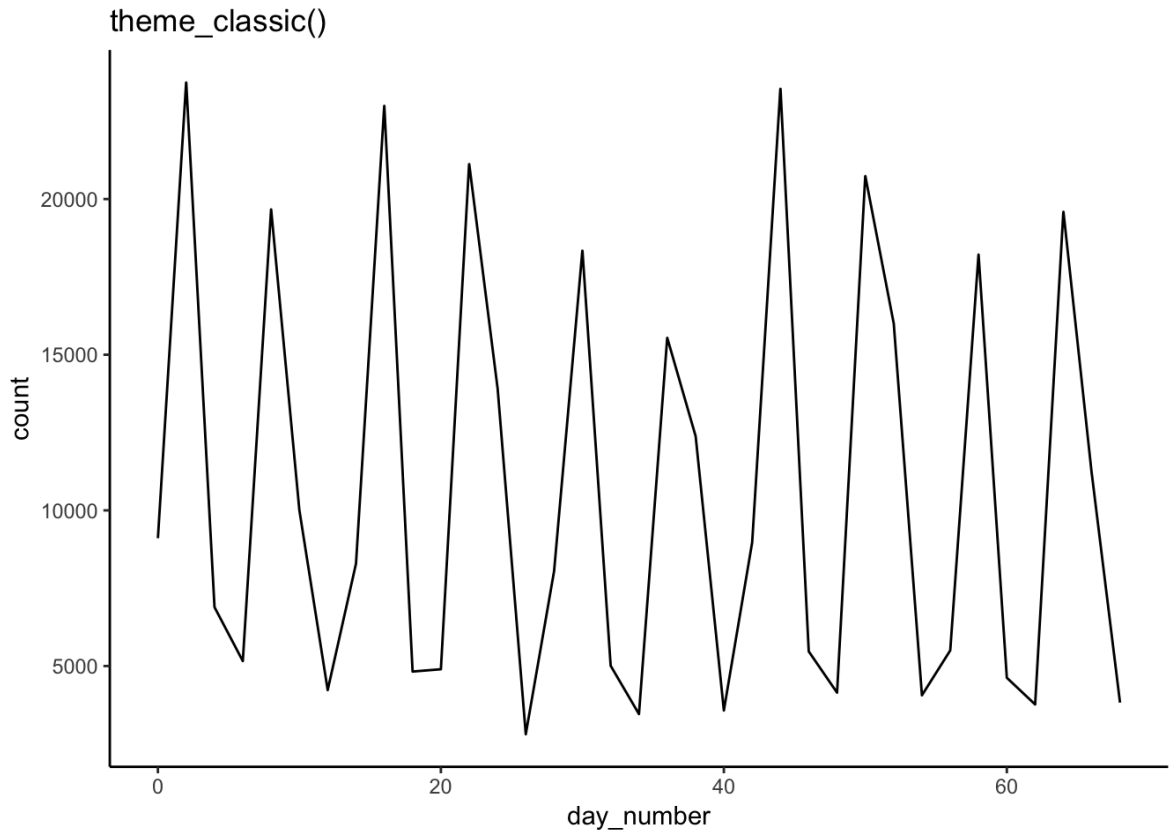
```
##"normal" ggplot output:
p1 + ggtitle("normal ggplot")
```



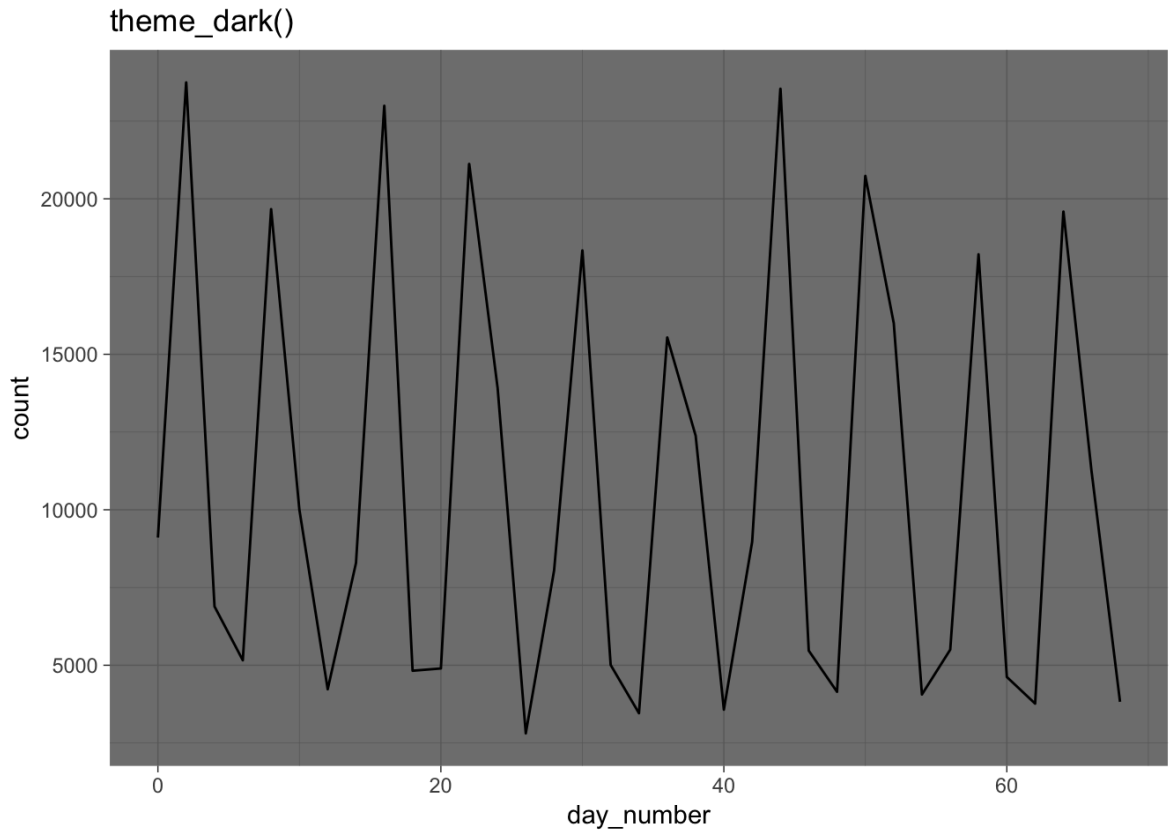

```
##alternative themes
p1 + theme_bw() + ggtitle("theme_bw()")
```



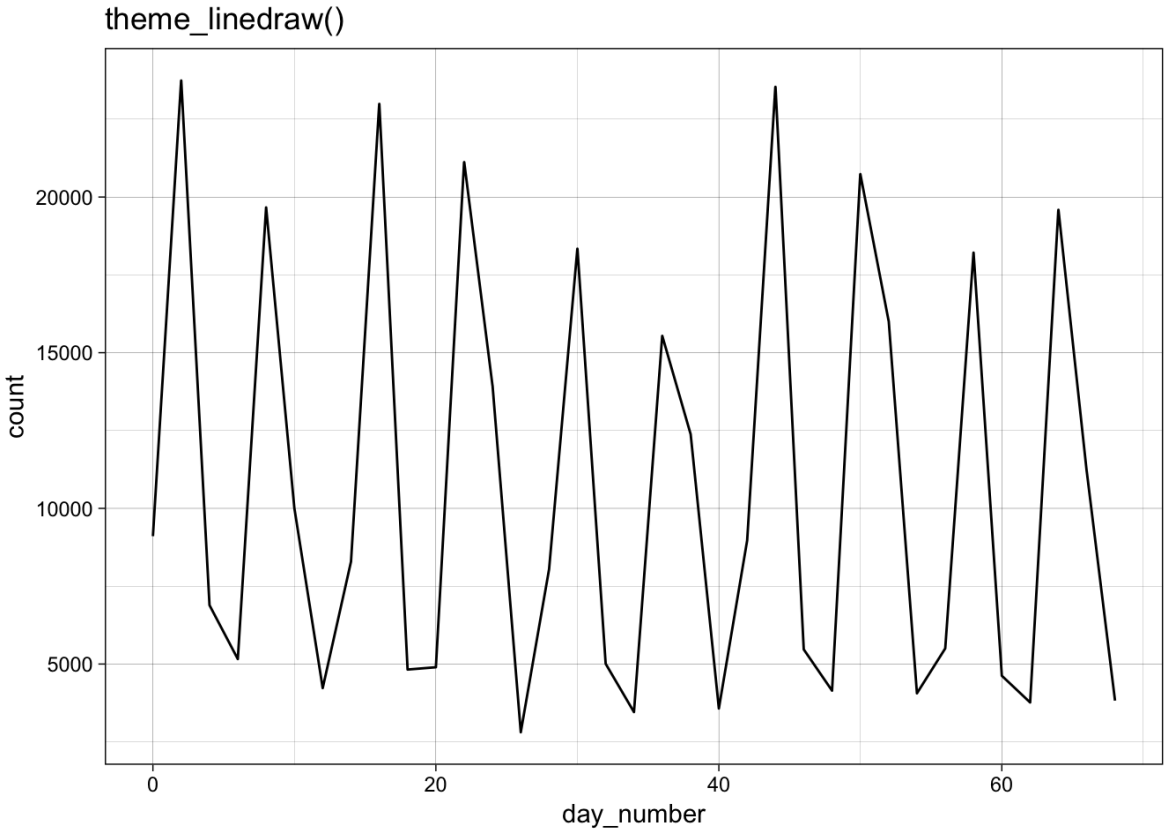

```
p1 + theme_classic() + ggtitle("theme_classic()")
```




```
p1 + theme_dark() + ggtitle("theme_dark()")
```

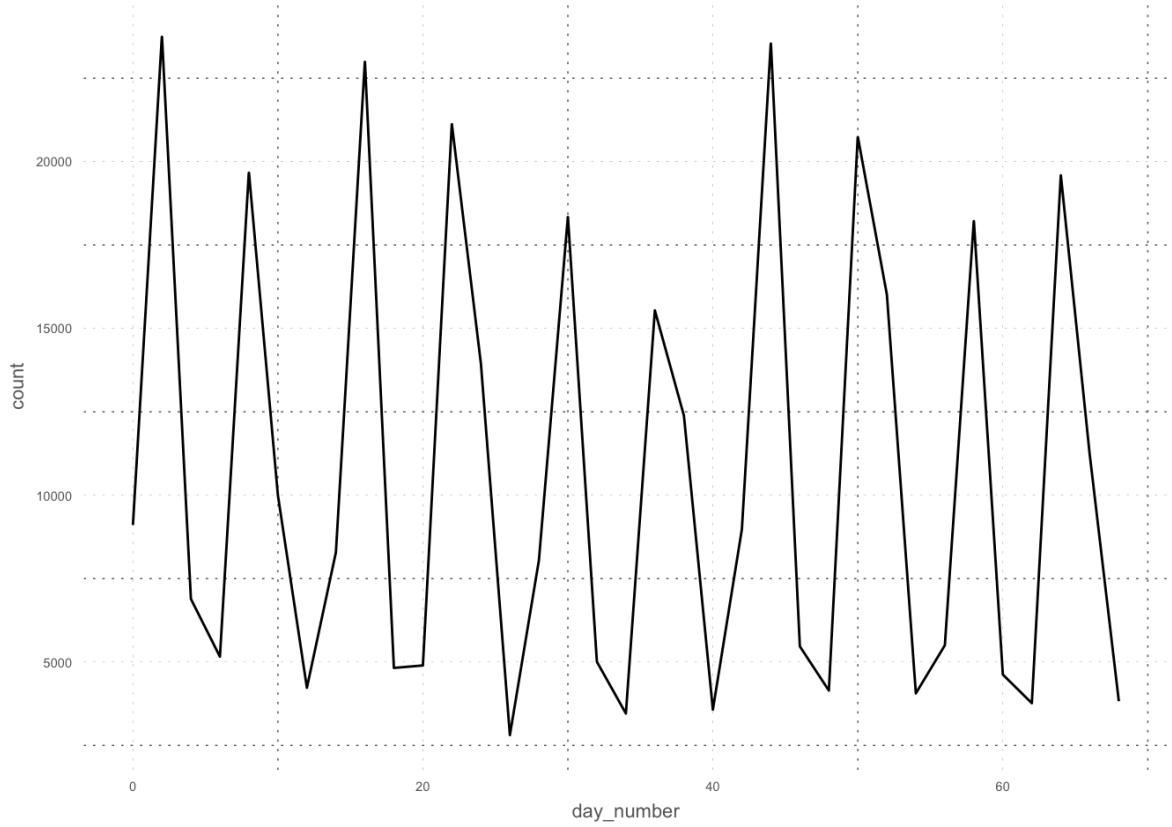



```
p1 + theme_linedraw() + ggtitle("theme_linedraw()")
```



And here is a custom one I pulled from R bloggers (https://www.r-bloggers.com/custom-themes-in-ggplot2/ (https://www.r-bloggers.com/custom-themes-in-ggplot2/)):

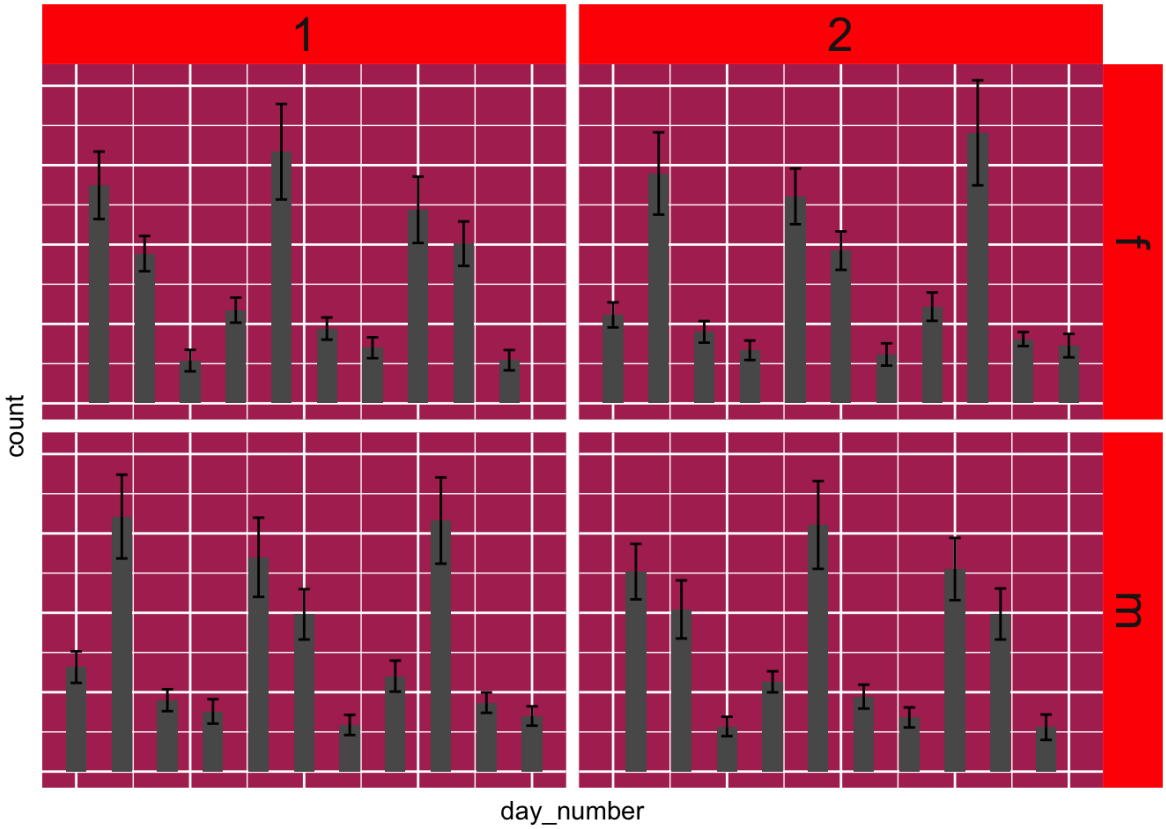
```
theme_new <- function(base_size = 11,
                      base_family = "",
                      base_line_size = base_size / 170,
                      base_rect_size = base_size / 170){
  theme_minimal(base_size = base_size,
                base_family = base_family,
                base_line_size = base_line_size) %+replace%
    theme(
      plot.title = element_text(
        color = rgb(25, 43, 65, maxColorValue = 255),
        face = "bold",
        hjust = 0),
      axis.title = element_text(
        color = rgb(105, 105, 105, maxColorValue = 255),
        size = rel(0.75)),
      axis.text = element_text(
        color = rgb(105, 105, 105, maxColorValue = 255),
        size = rel(0.5)),
      panel.grid.major = element_line(
        rgb(105, 105, 105, maxColorValue = 255),
        linetype = "dotted"),
      panel.grid.minor = element_line(
        rgb(105, 105, 105, maxColorValue = 255),
        linetype = "dotted",
        size = rel(4)),

      complete = TRUE
    )
}
p1+theme_new()
```



There are a bunch of fancy themes you can play with, and whole packages dedicated to providing new themes. Check out the ggthemr package (https://www.shanelynn.ie/themes-and-colours-for-r-ggplots-with-ggthemr/ (https://www.shanelynn.ie/themes-and-colours-for-r-ggplots-with-ggthemr/)).

Using the theme() argument allows you really fine control over what is being plotted. It's more hard work, but you can change things like the presence or absence of tick marks, background lines, the fills of facet title boxes, etc. And, if you define a custom theme (as below) you can then simply tag it on to any ggplot, so all your graphs in a publication have the same style. Have a go at hashing out some of the below options and see how they affect the overall look of a plot. There are a huge number of options in theme() so I have only included a few here (see: https://ggplot2.tidyverse.org/reference/theme.html (https://ggplot2.tidyverse.org/reference/theme.html)).

```
red_disaster_theme<-function(){
            theme(
                    axis.text.y=element_blank(),
                axis.ticks.y=element_blank(),
                axis.text.x=element_blank(),
                axis.ticks.x=element_blank(),
                strip.background=element_rect(fill="red"),
                    panel.background=element_rect(fill="maroon"),
                    strip.text=element_text(size=20)
              )
}

psmall + red_disaster_theme()
```



Grid extra

Combining multiple graphics is really useful, especially when your data is complicated. For instance, you might want to combine a plot of the abundances through time with some summary statistic so you dont use up too much space in your publication. The package gridExtra makes this easy.

```
require(gridExtra)
```

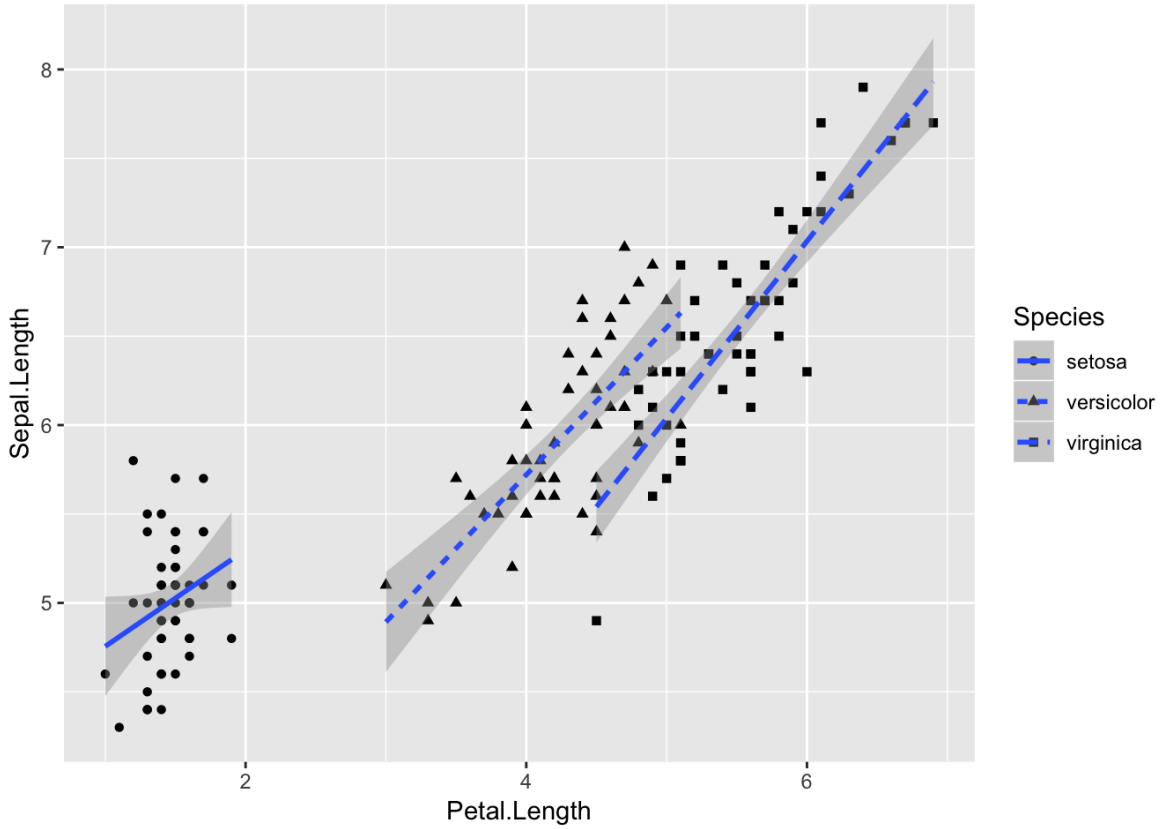
```
## Loading required package: gridExtra
```

```
##
## Attaching package: 'gridExtra'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

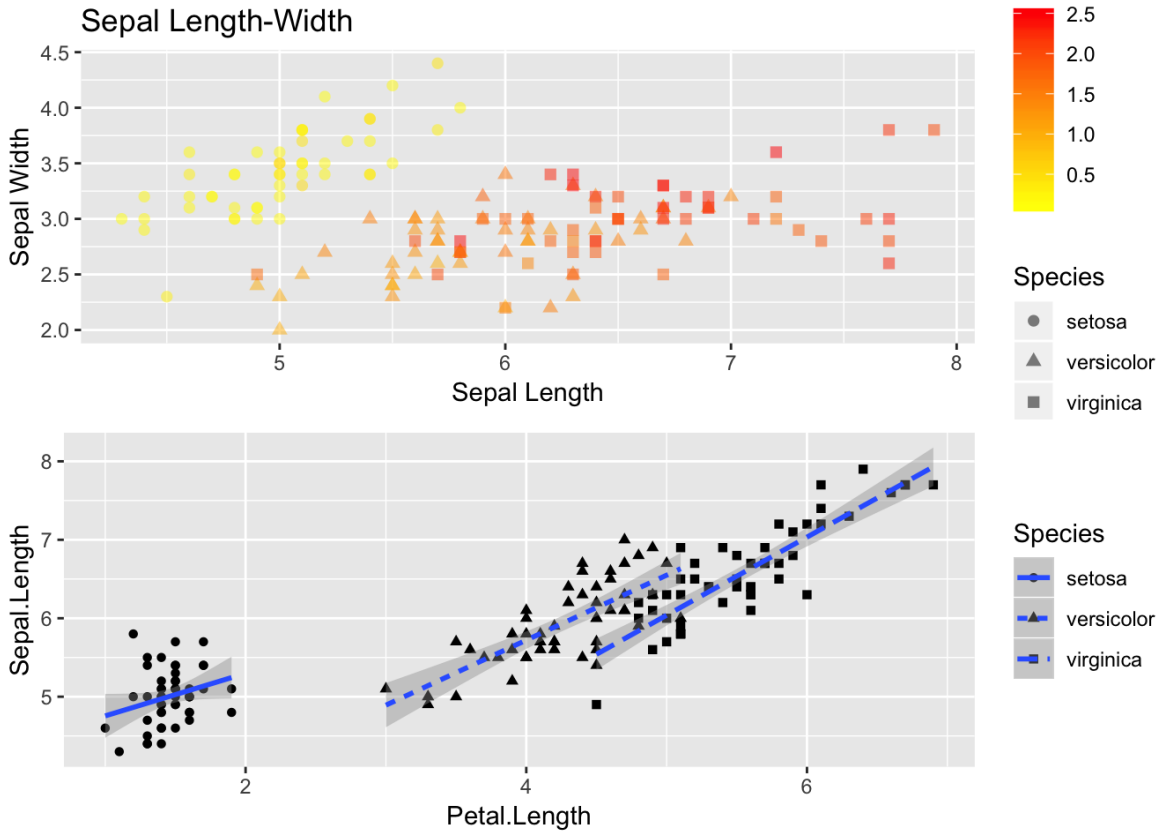
```
##pull out the scatter plot from earlier, and make a new one of petla length agains sepal length:
scatter
```


```
pet.sep <- ggplot(iris, aes(x=Petal.Length, y=Sepal.Length)) + geom_point(aes(shape=Species)) + ge
om_smooth(aes(group=Species, linetype=Species), method=lm)
pet.sep
```



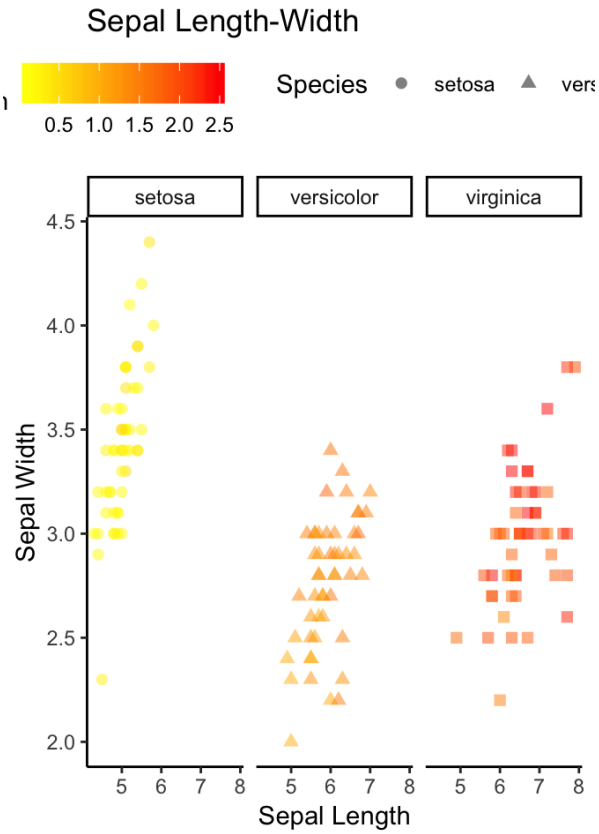
Combining them is as easy as:

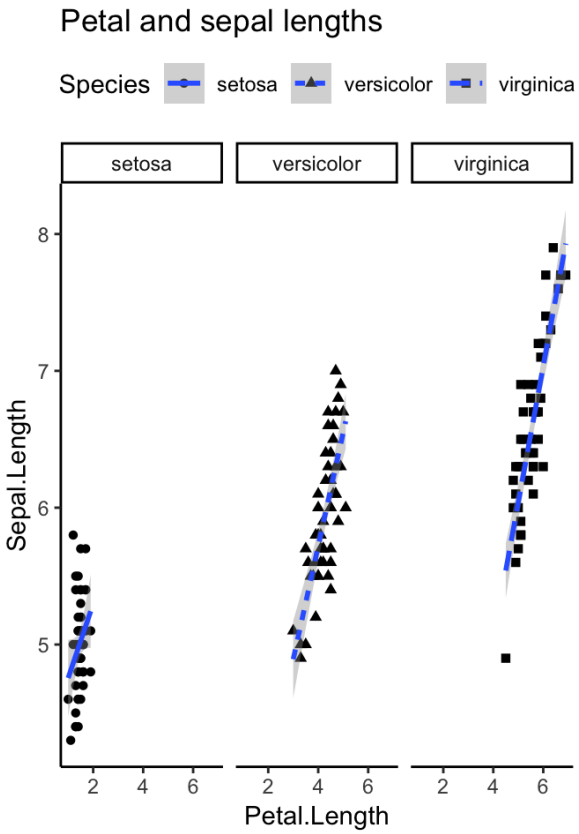
```
grid.arrange(scatter, pet.sep)
```



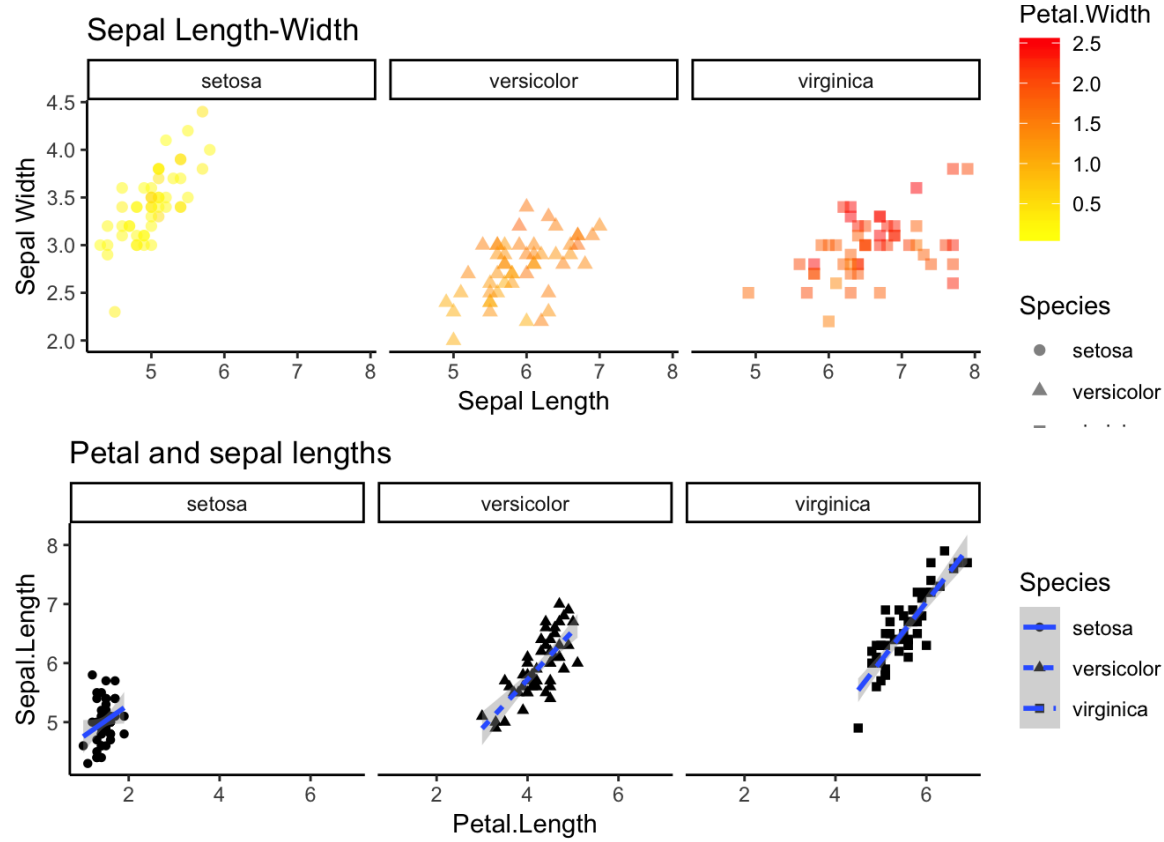
This looks ok, but we can make it look better:

```
scatter <- scatter + theme_classic() + theme(legend.position = "top") + facet_wrap(~Species)

##note that the colours in the second plot are redundant (the information is included in the x axi
s so we can get rid of the legend:
pet.sep <- pet.sep + theme_classic() + theme(legend.position = "top") + facet_wrap(~Species) + ggt
itle("Petal and sepal lengths")

##we could put them side by side instead
grid.arrange(scatter, pet.sep, ncol=2)
```



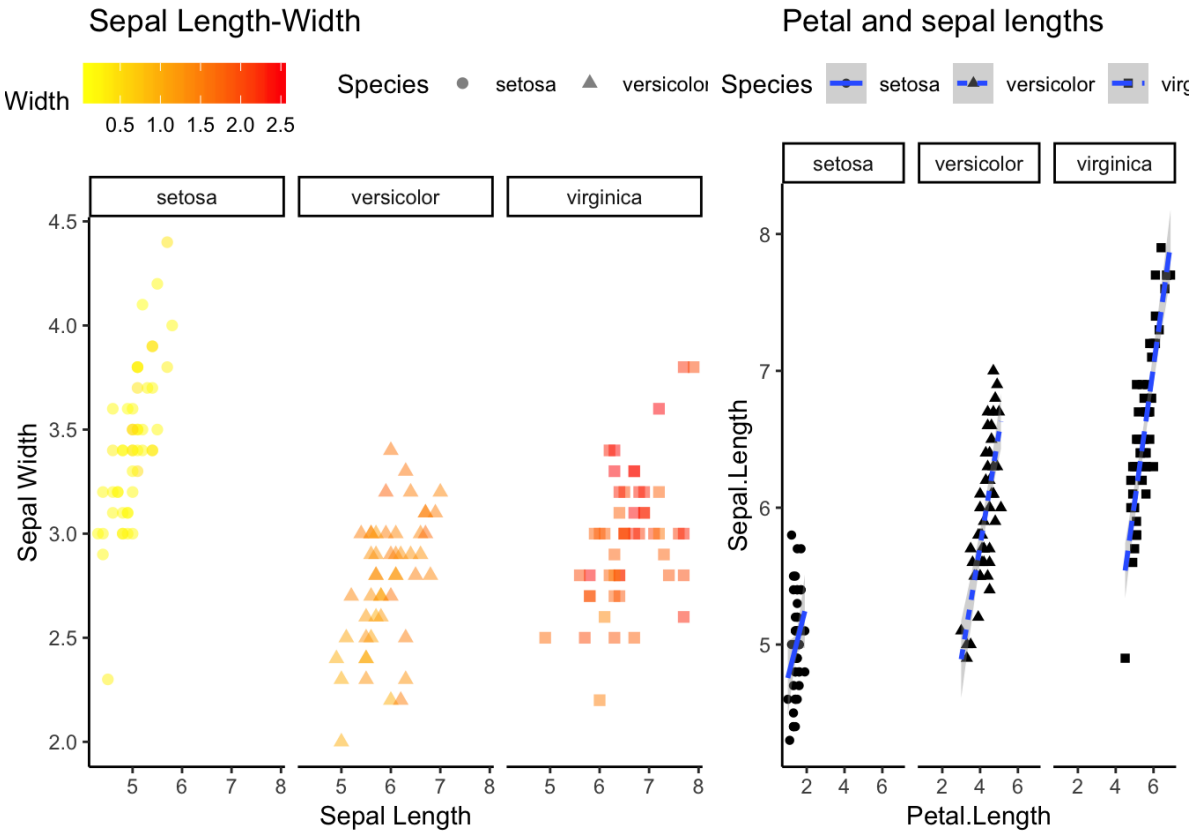



```
##but actually vertically works better, with their legends at the side:
grid.arrange(scatter + theme(legend.position = "right") ,
             pet.sep + theme(legend.position = "right"),
             ncol=1)
```



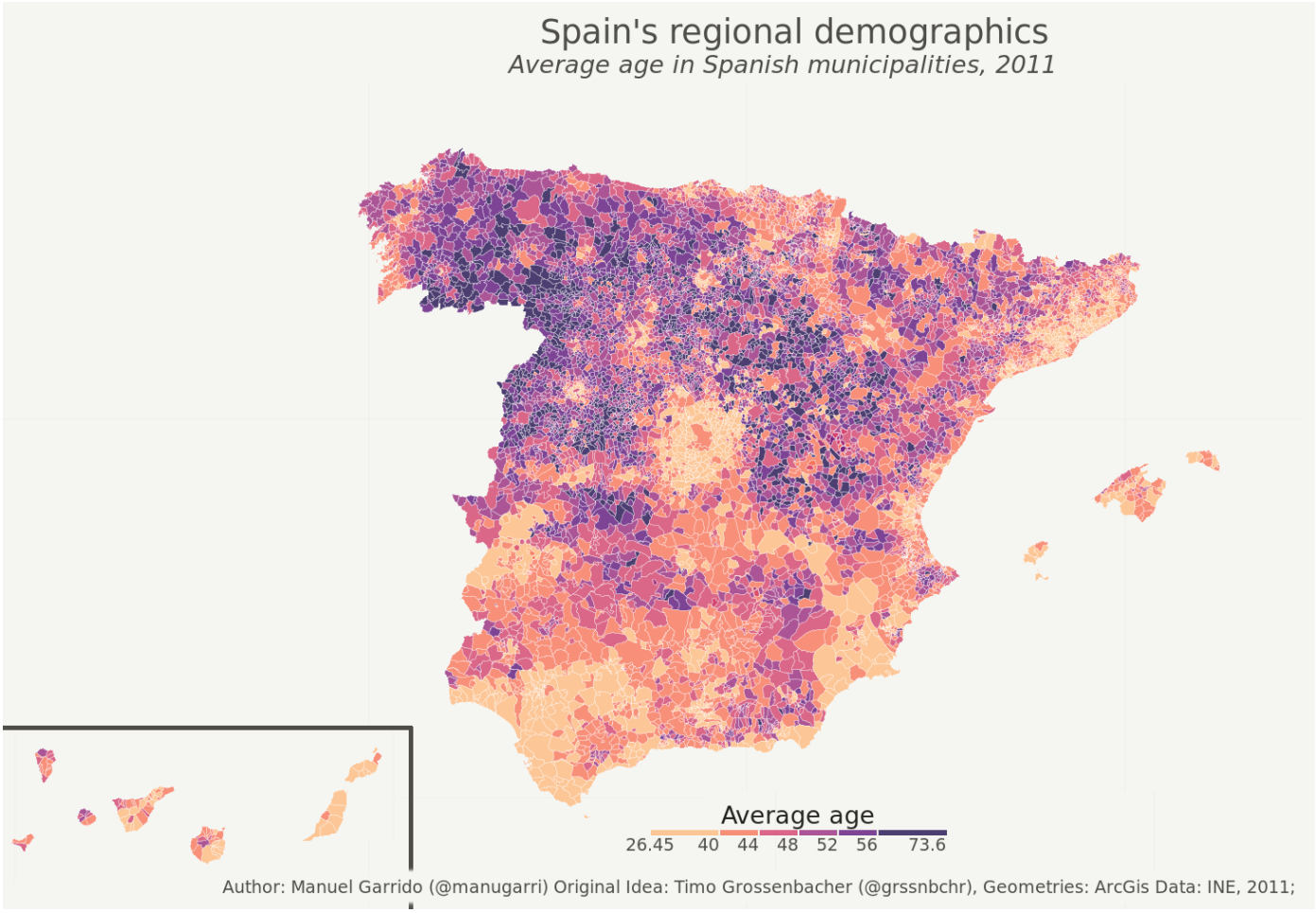
You can also change the relative widths of the combined plots:

```
grid.arrange(scatter, pet.sep, widths=c(6, 4))
```



Plotting spatial data

There are lots of ways to manitulate and plot spatial data in ggplot, and you can make some amazing maps:



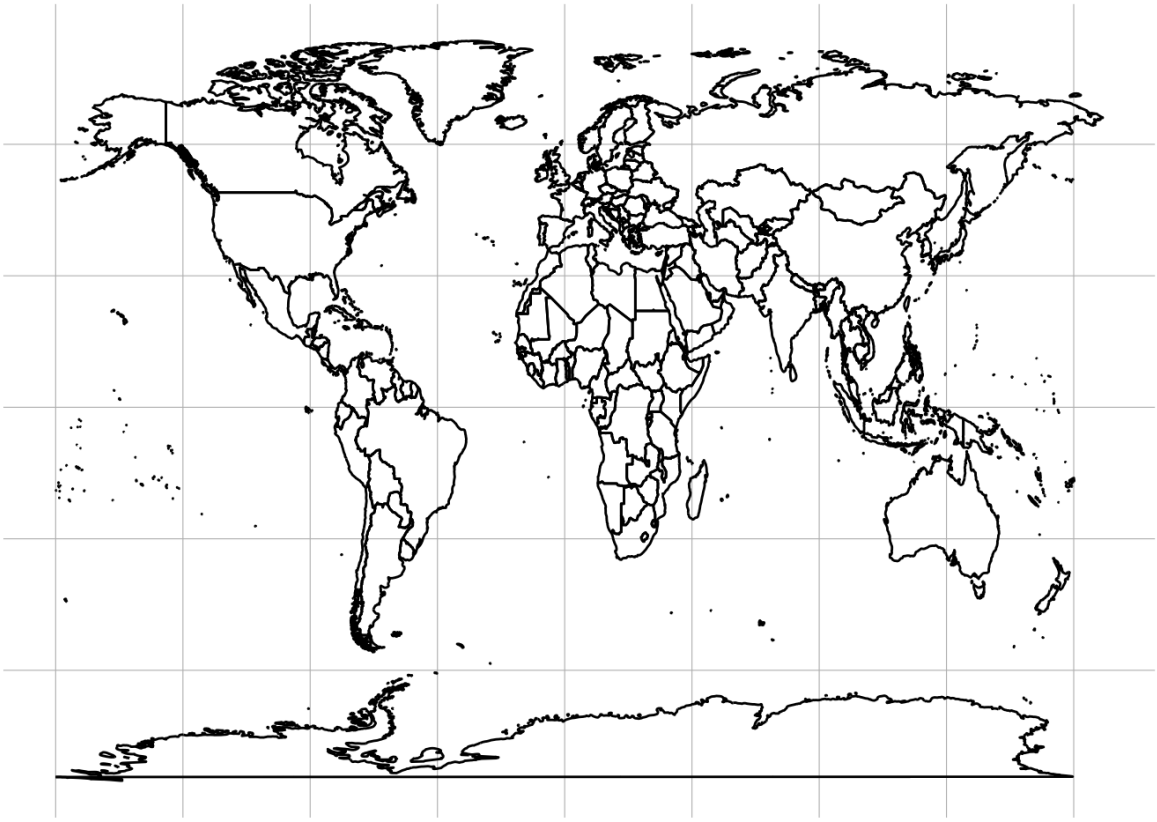
Not my area, but some basics are covered here.

A function for setting the look of the map:

```
theme_map <- function(...) {
  theme_minimal() +
  theme(
    text = element_text(family = "Ubuntu Regular", color = "#22211d"),
    axis.line = element_blank(),
    axis.text.x = element_blank(),
    axis.text.y = element_blank(),
    axis.ticks = element_blank(),
    axis.title.x = element_blank(),
    axis.title.y = element_blank(),
    # panel.grid.minor = element_line(color = "#ebebe5", size = 0.2),
    panel.grid.major = element_line(color = "grey", size = 0.2),
    panel.grid.minor = element_blank(),
    panel.border = element_blank(),
    ...
  )
}
```

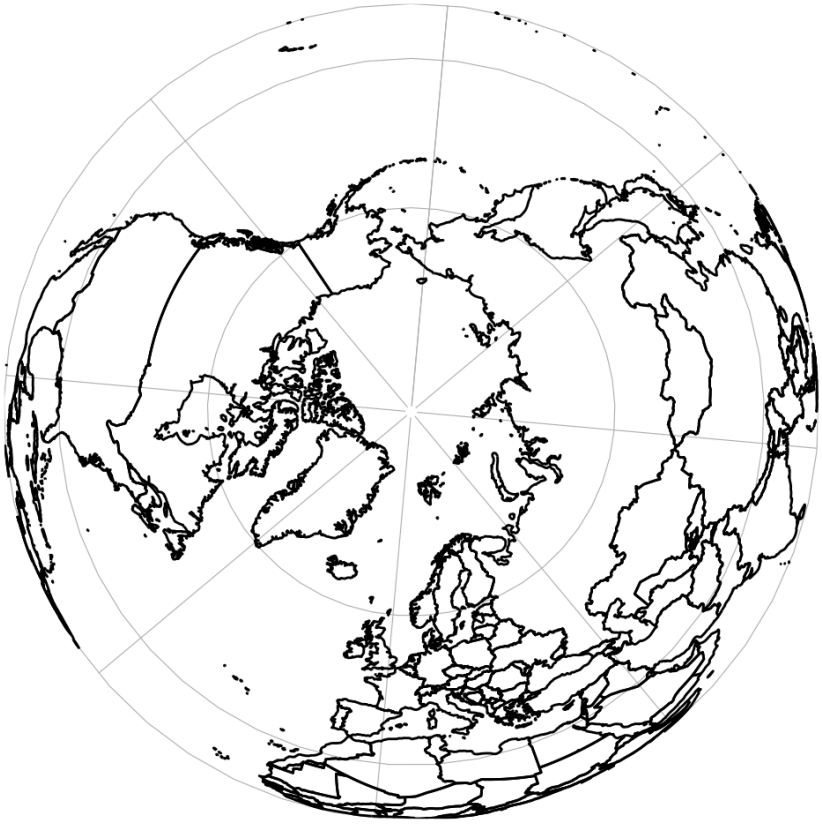
And here the actual code to make the maps:

```
world <- map_data("world")

worldmap <- ggplot(world, aes(x = long, y = lat, group = group)) +
  geom_path() +
  scale_y_continuous(breaks = (-2:2) * 30) +
  scale_x_continuous(breaks = (-4:4) * 45)

worldmap + theme_map()
```

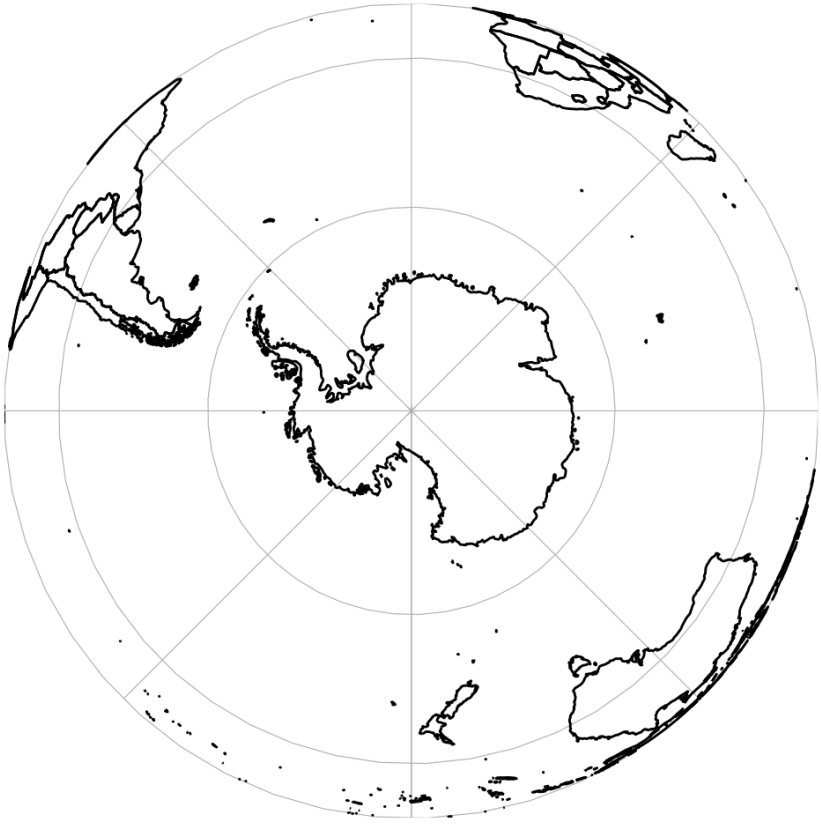


One of the really nice things is that you can easily change the orientation and projection type for the maps:

```
worldmap + coord_map("ortho") + theme_map()
```



```
worldmap + coord_map("ortho", orientation = c(-90, 0, 0)) + theme_map()
```



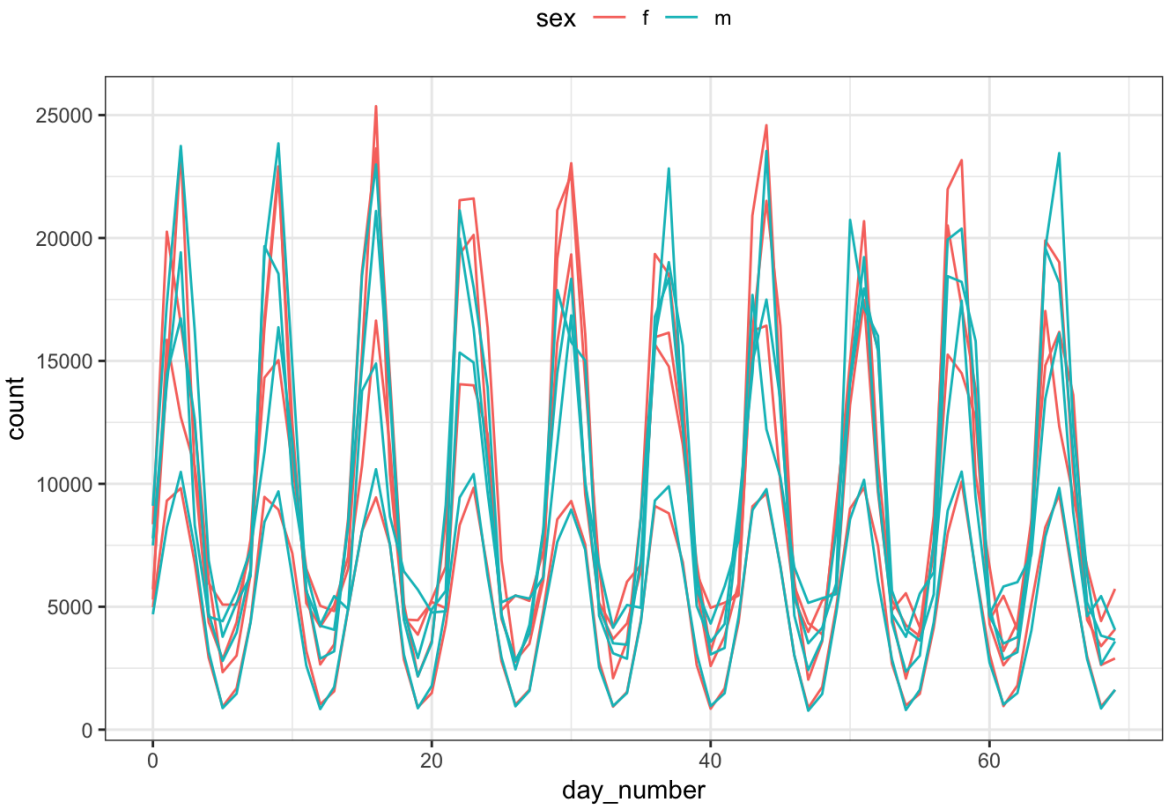
There are plenty of great resources for learning how to plot maps in ggplot, e.g.: - http://eriqande.github.io/rep-res-web/lectures/making-maps-with-R.html (http://eriqande.github.io/rep-res-web/lectures/making-maps-with-R.html) - http://zevross.com/blog/2018/10/02/creating-beautiful-demographic-maps-in-r-with-the-tidycensus-and-tmap-packages/ (http://zevross.com/blog/2018/10/02/creating-beautiful-demographic-maps-in-r-with-the-tidycensus-and-tmap-packages/)

Advanced functions

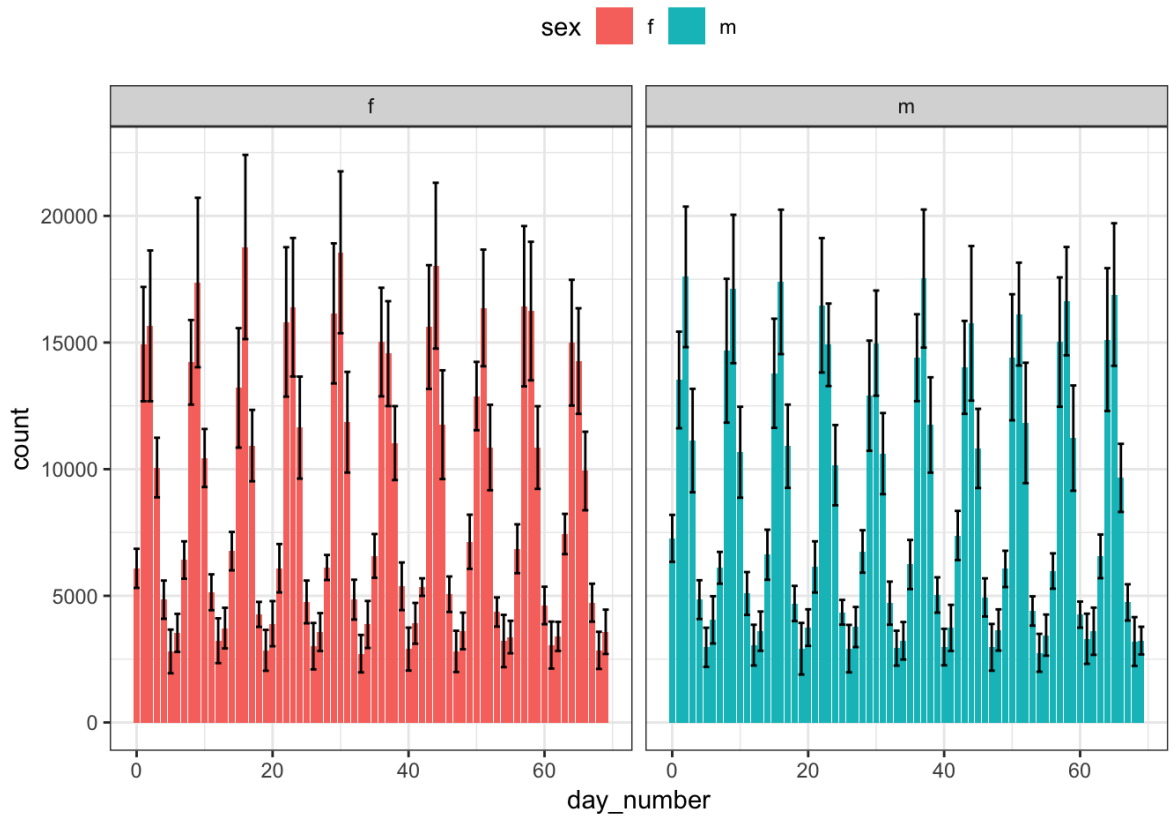
Combining plots with a shared legend

Using grid.arragne with two (or more) plots which require legneds will produce a legend for each plot, which is annoying if the legends are all identical. Plots can be combined using the following which produces a single plot with shared legends:

```
## we will make two plots with a shared legend:

## first a plot of the time series dynamics for a subset of the populations:
p6<-ggplot(subset(time_series_2, ts_number<=4), aes(x=day_number, y=count, group=interaction(ts_nu
mber, sex), col=sex))+geom_line()+theme_bw()
##if we place the legend position on top for now, it will help with the plotting later
p6<-p6+theme(legend.position = "top")
p6
```

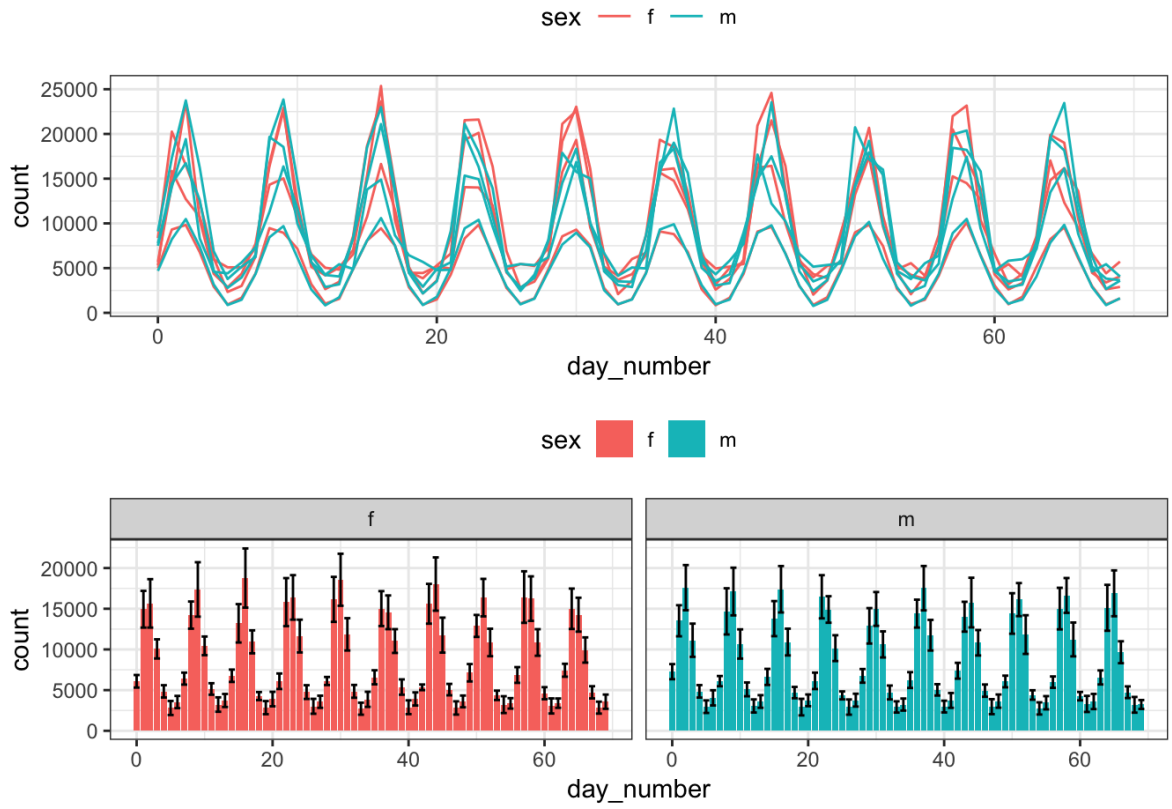



```
##and the mean population size at each time point across the populations:
p5 <- ggplot(subset(time_series_2, ts_number<=4), aes(x=day_number, y=count, fill=sex)) +
      stat_summary(fun.y = mean, geom = "bar") +
      stat_summary(fun.data = mean_se, geom = "errorbar") +
      facet_wrap(~sex) +
      theme_bw() +
      theme(legend.position="top")
p5
```



This is what it would look like if you just used grid.arrange:

```
grid.arrange(p6, p5)
```



Grim.

So to solve this we will then need a function which grabs the legend information from a specific plot:

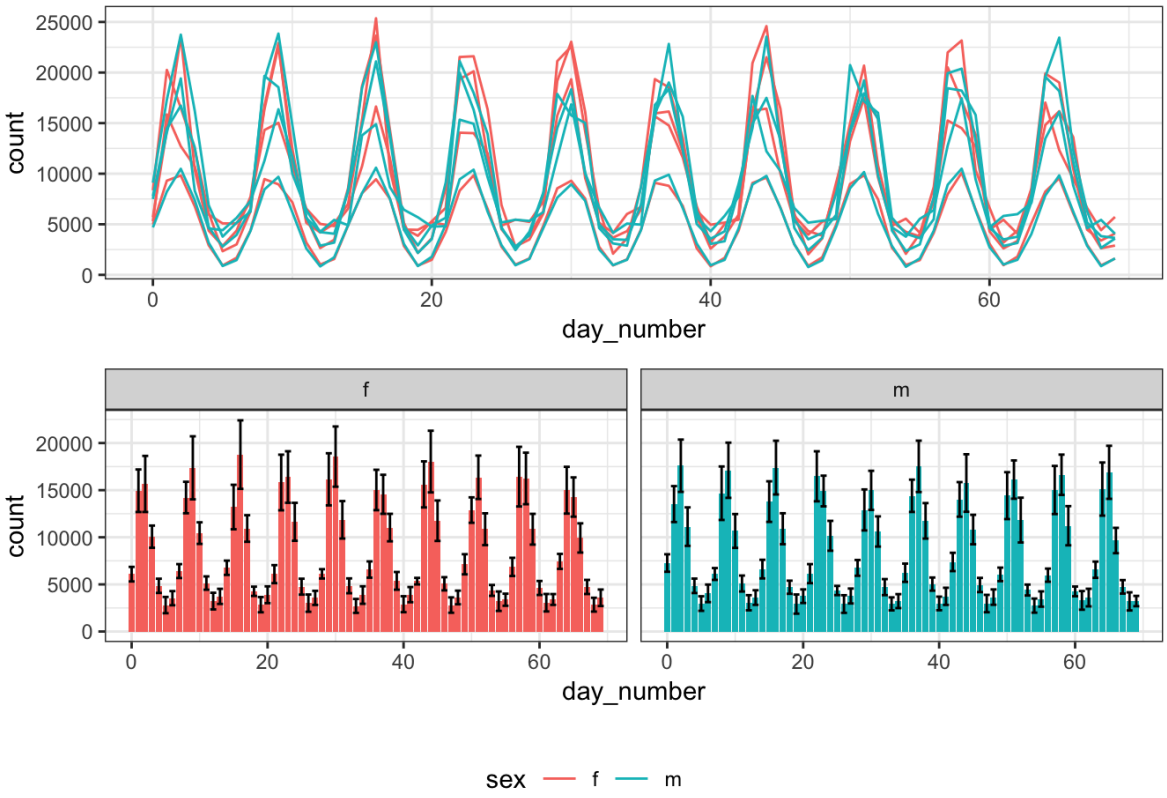
```
##a function to save out the legend information to a seperate object
g_legend<-function(a.gplot){
  tmp <- ggplot_gtable(ggplot_build(a.gplot))
  leg <- which(sapply(tmp$grobs, function(x) x$name) == "guide-box")
  legend <- tmp$grobs[[leg]]
  return(legend)}
```

Take the legend data from p6:

```
p6legend<-g_legend(p6)
```

and make the multipannel plot:

```
grid.arrange(arrangeGrob(  p6 + theme(legend.position="none"),
                           p5 + theme(legend.position="none"),
                            nrow=2),
             p6legend,
             nrow=2,
             heights=c(6, 1))
```



Competition!

So you have had a brief introduction to what ggplot can do. Now lets see how beautiful a plot you can make. Split up into groups of 3-4 (and try to mix abilities so you can learn from one another). Have a look at the example data set I sent round in the email (and on slack in the “general” thread).

Use this data set to make some summary plots - they can be of anything, and can comprise of 1 or more plots (but make them pannel if there are more than 1). Think about making the plots information dense, whilst remaining understandable! Make it as aesthetically pleasing as possible, the winning group will be good for a round of beers from me at friday drinks.

You have 15 minutes. Then drop your group’s finished plot into the slack channel and we can look at the,