Iterative Plotting for Easy and Aesthetic Figures

Author: Cole Brookson **Date:** 20 July 2022

When making a plot, typically the best way to go about it is to iteratively make a version of the plot, observe the outcome, and then make changes in an iterative fashion. This prevents us from making mistakes in a long set of code that we then can't diagnose the problem with, and further, helps us catch mistakes (typos, colour problems, etc) as we only have to take in a few new components each time, and not a whole pile.

For our example, let's use the example from the Intro to ggplot2 section, looking at salamanders and trout:

Building a Basic Plot

Let's again plot a scatter plot of length_1_mm vs. weight_g. We'll start out with the absolute minimum number of arguments:

1. The call to ggplot() 2. The aesthetics (i.e. what variables we want on the x- and y-axis) 3. The geom_ function denoting what type of plot we're going for

Ok so we see our plot! Perfect. So the first thing to do, before we make any stylistic additions, is to deal with the components of the plot that are must-haves to have a legible, complete plot. First step is the axis labels. So to iteratively work up our plot, we'll add that one component on, by using the new function `labs()`:

OK good. Now, this is more aesthetic than anything, but we might want to change the appearance of this plot away from this ugly grey background and grid lines. This is a more personal decision regarding what exactly you want your plot to look like. However, there are lots of themes available in the ggplot2 package to choose from. For example, we could use the theme_classic() like so:

But I personally prefer a different theme that comes in the ggthemes package, which we can install now:

Now, let's use the theme_base() option, which will give us a very classic look and feel to the plot:

Perfect! You'll also notice the text sizes are now larger which is preferable for reading on a screen or in a report.

Adding Colour & Shape

Let's now add a colour to our points and change the shape we're using as well.

Note: Arguments like this go in the aes() section, but there is a difference between simply assigning a colour and using colour to group points. To start, we'll just make the colour red, change the shape to my personal favourite, and up the size just a smidge:

So this is interesting! We see that somehow we've made our points hollow on the inside. Well, that has to do with the shape we picked. There are lots of options for shapes we could use, and we can see all our options in this handy graphic here:

So notice we specified point # 21. This point has a dark outer circle and a fill colour in the middle. When using a point option with an outter boundary and an inner fill, the argument colour refers to the outter boundary line, and the argument fill refers to the colour filling inside**. Let's see how this works by changing colour = "black" and fill = "red":

Okay, so this looks like maybe more along the lines of what we were expecting! One more thing though is that some spots on the plot have a high density of points. To better identify those high density areas, let's make our points slightly transparent, with the alpha commmand:

And now we can see where there really are many points.

Variables by Group

One of the most useful things to be able to do is group our points by either shape, fill, colour, or even size to show some difference between them. For example, in these data, we can see two very clearly different trends.

So we can see our two main point colours plotted out nicely here. Although, there should be four colours and we can't really see them. Let's change our alpha to fix that:

That's better. Looking at this plot, this brings up a few common tasks we might want to do from here. First of all, it's clear from this plot that the vast majority of these points are only from two species. We can see how the samples in our dataframe fall out with the simple table() function:

So there are only 15 of the Cascade salamanders, and we know from our plot, at least a few NA's. Now the following decisions depend HIGHLY on what message one is trying to communicate with the plot, but we could imagine a scenario wherein perhaps it's actually important to have the Cascade salamander points be visible. But, it's clear from our plot, we may first want to get rid of NA's at least fo this plot. So let's do that via a filter() Remind yourself about this here.

Okay, let's go ahead and make our plot again:

This looks better. Now perhaps one way we could make the Cascade salamander points stand out is by making them significantly larger in size. We can do that my scaling our size manually according to the levels of our grouping variable. Note, that scaling, whether manually or otherwise, can be done for all grouping aesthetics such as fill, colour, alpha, etc. Further documentation on this topic can be found on the ggplot2 website

Since we see that the Cascade Torrent Salmander is first in our list of the legend, that means we will need to make the larger size first up in our values argument:

Hmm. We can vaguely see in the bottom corner the points we want, but they're hard to see. Why is this? Well, the problem here is that ggplot() will automatically plot data in the order they are passed. we can see by a quick head() call ...

That the first species in the list is Cutthroat trout, and with a quick tail() call (tail() returns the last six rows instead of the first six) ...

... we can see that the Coastal salamander is the last. That means that the Coastal salamander will be plotted after (i.e "on top of") our focal larger points. Luckily this is an easy fix by simply rearranging the dataframe such that the focal species is the last one to be plotted. This can be accomplished with dplyr::arrange(). We'll pass first the dataframe we're working with, then the column we want to sort by. To check that this will work, let's print the call to tail() from the arrange() version of our dataframe.

That doesn't work! That's because arrange() will automatically default to sortin alphabetically. Luckily we can convert this column into factor to change the order in a custom way. Lets do so like this (and again check with tail()):

And we see our friend the Cascade salamander is at the end. Now we don't necessarily want to reassign our dataframe to take this rearranged form, so we can simply wrap our call to the dataframe df_filtered in our arrange() function and see if that works:

We can now see our Cascade salamander data points! While we could stop here, there are a few other things we can do. We can capitalize the legend easily enough, by simply adding it as the first argument in the scale_size_manual() function. We put it here since the size parameter is what's being displayed in our plot:

Ahh! Now we've run into a new problem. We have two legends! This is annoying. However, we can fix this, but first, let's choose different colours, these are not my favourite. We can do that (in this case) with scale_fill_manual(). We use the fill because the colour for all the points is black since it refers to the outer border of the points.

SIDE NOTE: Note that when making figures, when possible, choose colours that if printed in black and white are still differentiable, and that also are distinguishable for colour blind people.

Also, note that often the easiest way to add in specific colours (if you're only using a handful of specific ones like we are here) is to pass the RGB code for that colour. If you have never heard of RGB colours, essentially it's a code of alphanumeric characters that form a specific colour on the colourwheel. Learn more on RGB colours here. Interestingly,

While these colours are harsh, they're easy to see. We could choose others if we want to be more aesthetic :) Also, we notice that our legend problem is gone! Why? Well, what created the problem before was having technically two different legends being asked for from our aes() arguments - size & fill. However, by manually providing values for both and then giving them the same legend title, we fixed up the problem. On your own, try adding a different title for one of them and see what happens!.

For this plot, we are pretty much done! This looks good to go. This section has hopefully showed you how to iteratively add components to your plot to troubleshoot throughout and end up with a nice plot quickly. At this point, we can save our plot if we want to (and if it has been assigned to a variable). How to save ggplot objects.