Contents

Graphics with ggplot2

Today we will be covering some of the basics of how to use the R package **ggplot2**. The goal is to introduce you to some of the terminology used in **ggplot2** and then go through some basic and then more advanced examples to give you some exposure to the **ggplot2** style of coding.

We will spend some time preparing data for graphical display. In **ggplot2** there is a strong relationship between the dataset you are working with and the graphics you make. While you will see some code for data manipulation using packages **tidyr** and **dplyr**, we will not discuss it in much detail as the focus today is on creating graphics.

Online resources

There are many resources for **ggplot2** help online. Once you get started making graphics, you will be able to find answers to the majority of questions you have through online searches. Here are a few websites that I have found to be most helpful.

- For nice coverage of some of the basics, check out the Cookbook for R website: http://www.cookbook-r.com/Graphs.
- All the functions, complete with examples, are on the **ggplot2** website: http://ggplot2.tidyverse.org/reference/.
- Folks on Stack Overflow answer questions about **ggplot2** code pretty much daily. You can look at (and search in) all the questions tagged with the [ggplot2] tag: http://stackoverflow.com/questions/tagged/ggplot2.
- The active development of **ggplot2** is done on GitHub. To report bugs or check for updates, see the GitHub repository: https://github.com/tidyverse/ggplot2.
- In recent years, **ggplot2** has become easier to *extend* to make additional kinds of plots. A webpage that tracks packages that extend **ggplot2** to make other kinds of plots is here: http://www.ggplot2-exts.org/gallery/

Loading the package

The first thing we need to do is to load package **ggplot2**. If you are not working on a computer with a current version of **ggplot2**, which is version 2.2.1, you will need to install it first.

To install the package you can type `install.packages("ggplot2")` in your R Console or go the the RStudio Packages pane, click **Install**, type the name of all the packages you want to install, and press enter.

Once the package you want to use is installed, you can load it into R via the `library` function. Let's do that for **ggplot2** now.

```r
library(ggplot2)
```

The basics of ggplot2 and exploratory graphics

The goal of the first part of the workshop is to show you some of the basic **ggplot2** syntax while making graphics. We will be making these simple graphics using the built-in R dataset `mtcars`. Information about this dataset is available in the R help files. The variables we will be using from this dataset are:

`mpg` (miles per US gallon),
`wt` (car lbs/1000),
`cyl` (number of cylinders),
`am` (type of transmission), and
`disp` (engine displacement).

We will treat `mpg` as if it were our response variable of interest.

Let's take a quick look at the first six lines and structure of this dataset. You should recognize that `cyl` and `am` are both categorical variables. However, R reads them as numeric variables in the dataset because their categories are expressed as numbers.

```r
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```r
str(mtcars)
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

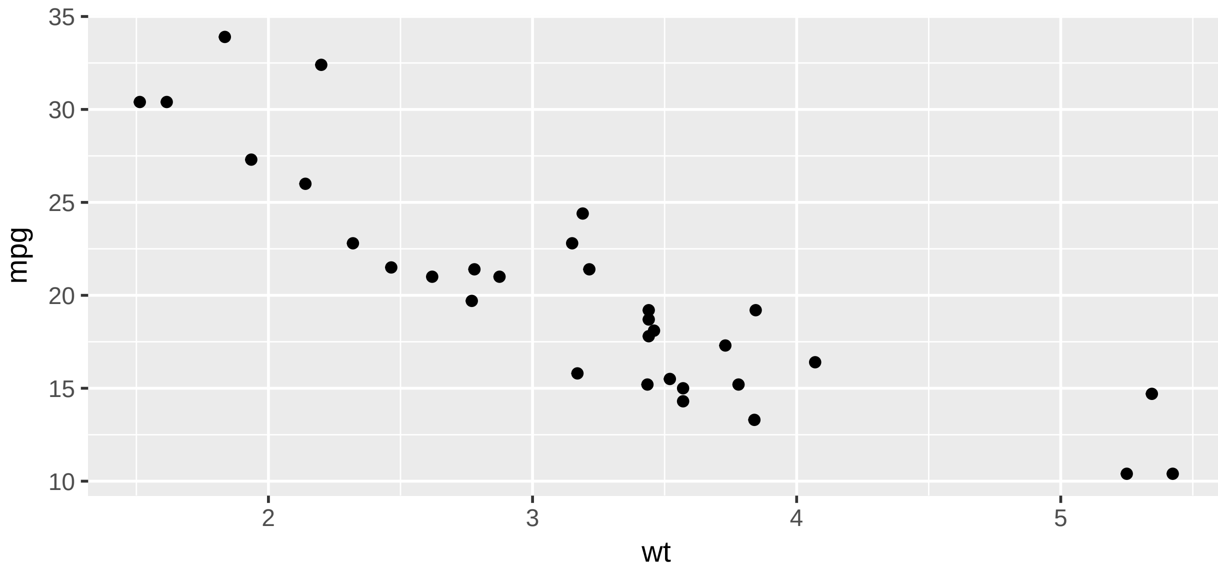
Basic plots with `qplot`

There is a convenient wrapper for the `ggplot` function called `qplot`. This function is convenient when making quick exploratory graphics (the `q` in `qplot` stands for *quick*). The syntax of `qplot` mirrors that of the `plot` function from base R, which is simpler than the `ggplot` syntax. We are going to use `qplot` to make a couple of simple graphics, and then we'll jump into using `ggplot` directly for the rest of the workshop.

Scatterplot

Our first graphic will be a scatterplot of `mpg` vs `wt`. A scatterplot is the default plot type in `qplot`.

```r
qplot(x = wt, y = mpg, data = mtcars)
```



Some things to notice from this first code and plot:

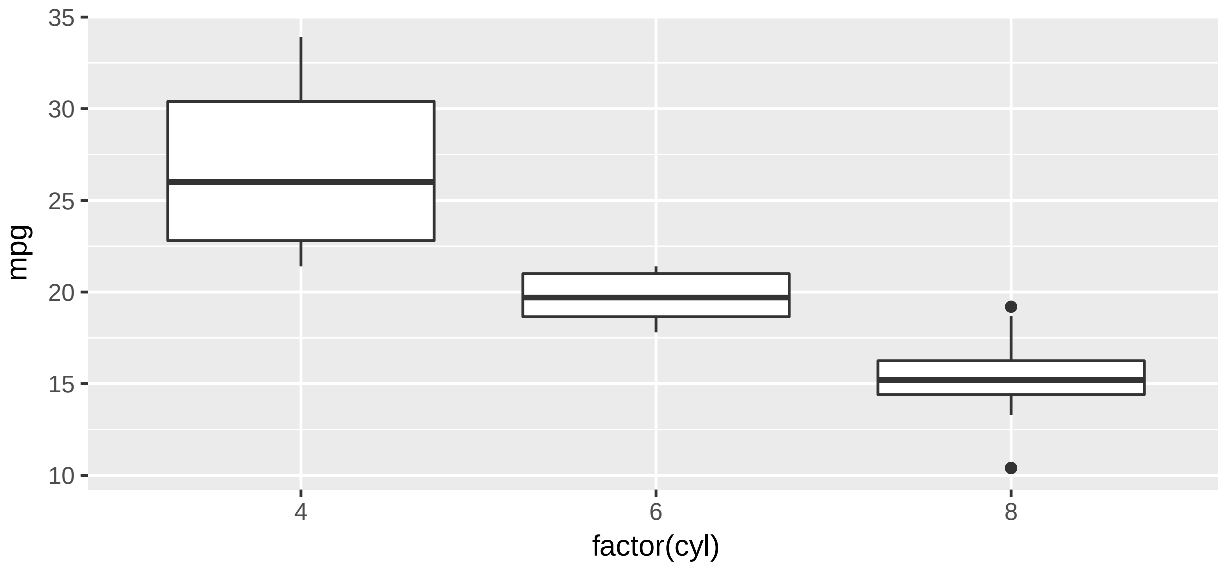
- We define the dataset we are working with **ggplot2** using the `data` argument.
- We define which variables are going to be placed on each axis in the plot via `x` and `y`.
- The default panel background for **ggplot2** graphics is gray with white grid lines.

Boxplot

Next we'll make boxplots of `mpg` for each level of the categorical variable `cyl`. The x axis variables needs to be categorical when making boxplots, so we have to change `cyl` to categorical using `factor(cyl)` since `cyl` is numeric in `mtcars`.

To change the type of plot we make, we use the `geom` argument. In **ggplot2** if we are talking about different *geoms* we are talking about different types of plots.

```
qplot(x = factor(cyl), y = mpg, data = mtcars, geom = "boxplot")
```



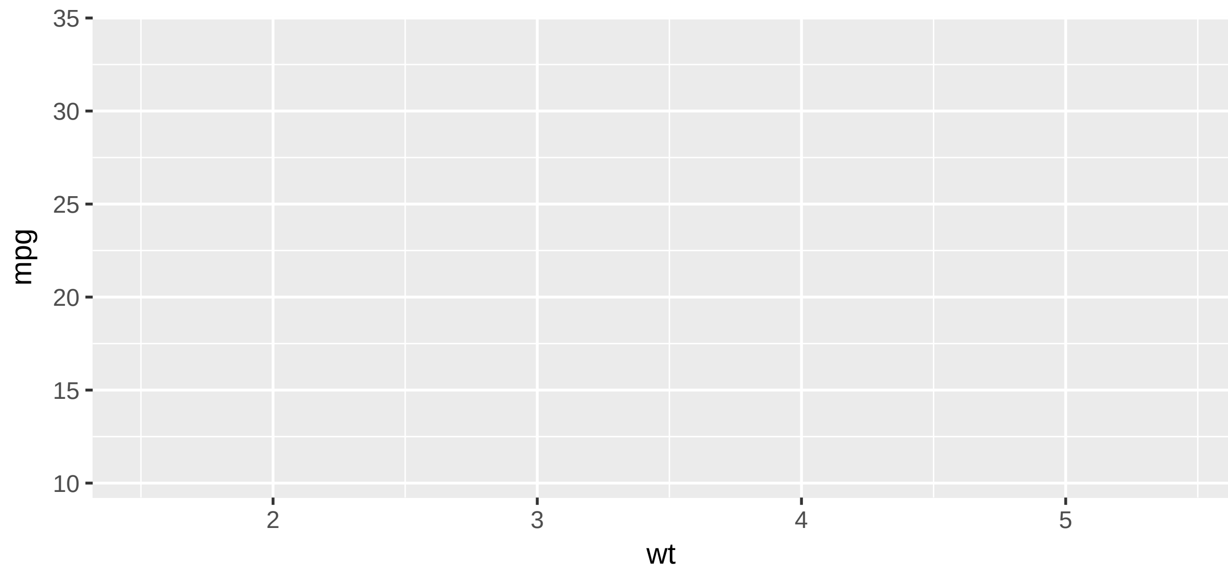
Basic plots with `ggplot`

Now we'll switch over to using `ggplot` directly to start learning the `ggplot` syntax. We'll begin by making the same two graphics that we created with `qplot`.

The standard way to use `ggplot` is to define the dataset and the axis variables within the `ggplot` function and then build the graphic by adding *layers* of geoms. Notice that the axis variables are defined within the `aes` function inside `ggplot`. The `aes` stands for *aesthetics*, which we will be talking more about in a few minutes.

Here is how we use `ggplot` to define the dataset and axes. We get a blank graph as we haven't chosen a plot type yet.

```
ggplot(data = mtcars, aes(x = wt, y = mpg) )
```

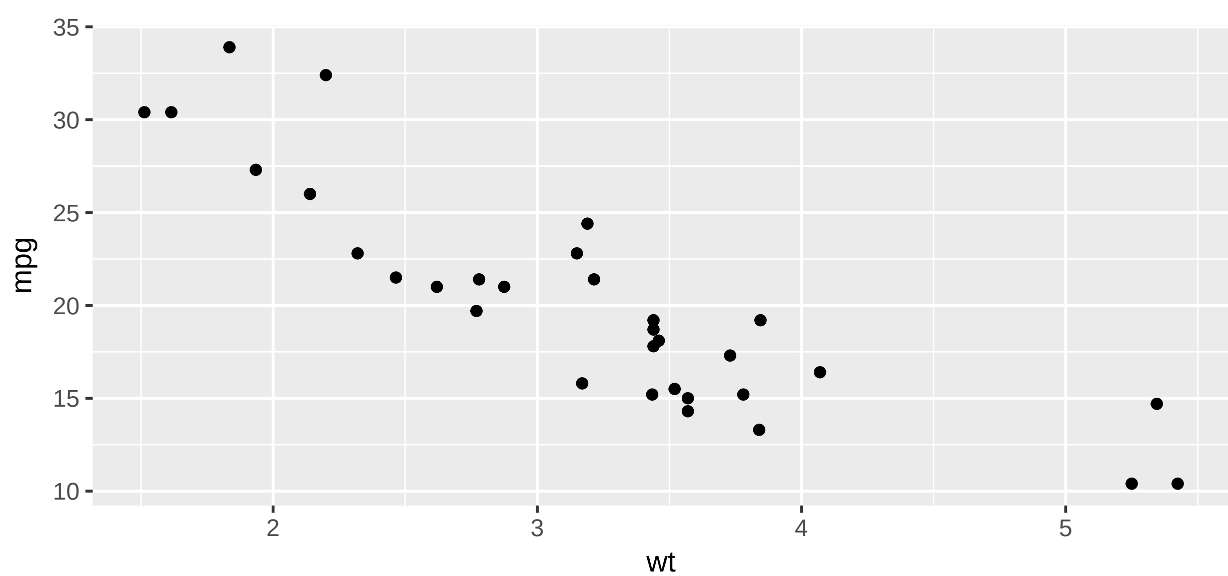


Scatterplot

We add layers that define the type of plot we want to make (the *geoms*) by using whatever geom function we want along with the `+` sign. In this case we are making a scatterplot, so we will add the `geom_point` layer.

It is standard coding to put new layers on new lines. This makes the code more readable, which is important when a collaborator (such as future you) needs to understand what you did. Putting spaces in your code, much like you would while typing a sentence, also makes your code more readable.

```
ggplot(data = mtcars, aes(x = wt, y = mpg) ) +
    geom_point()
```

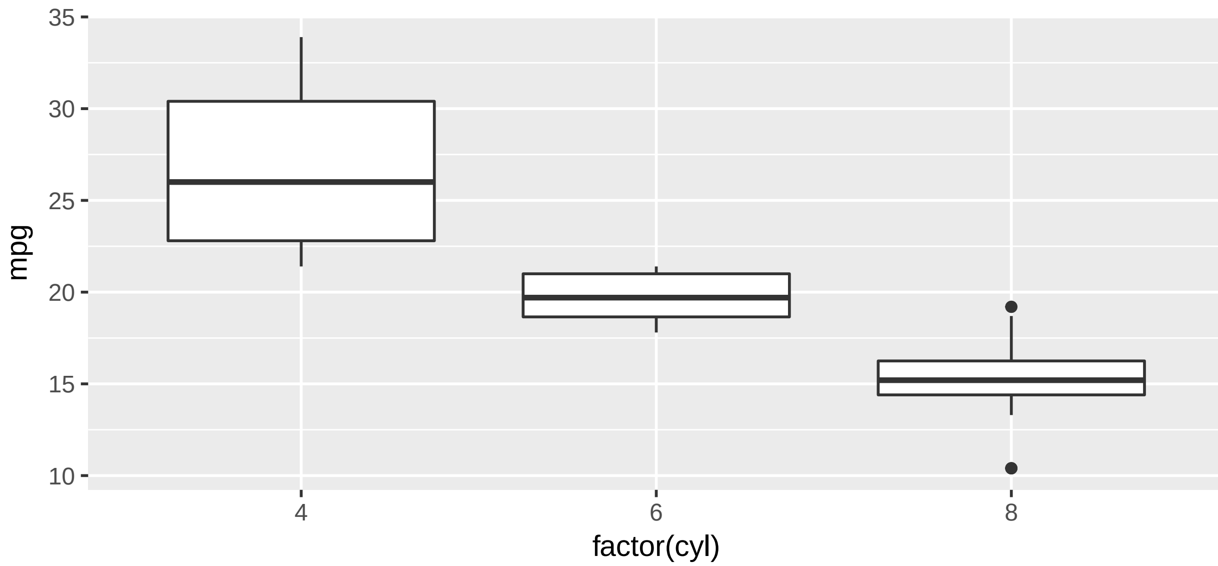


Boxplot

Next let's make the boxplot. We again define the dataset and axis variables within `ggplot`, but this time we add the boxplot geom, `geom_boxplot`, instead of `geom_point`.

Defining the dataset is an important part of the `ggplot` philosophy. It allows us to refer to columns in the dataset directly by name. A common mistake for new `ggplot` users is to use dollar sign notation to refer to column names (e.g., `mtcars$mpg`); don't do this, as it can lead to mistakes and graphics that don't display the dataset correctly.

```
ggplot(data = mtcars, aes(x = factor(cyl), y = mpg) ) +
    geom_boxplot()
```



Mapping aesthetics to variables

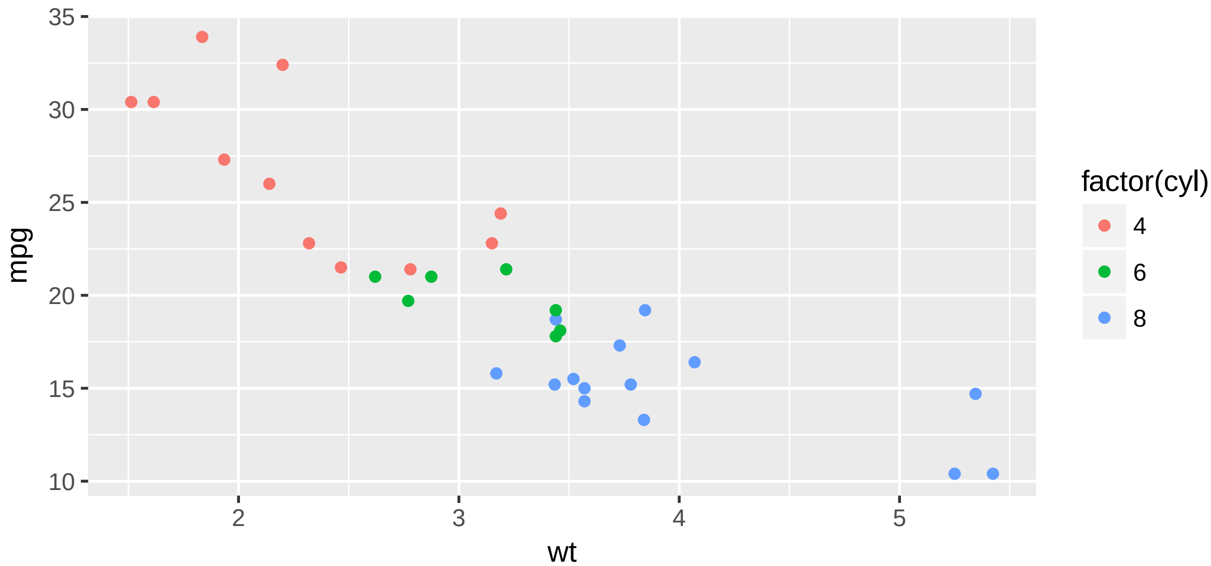
Package **ggplot2** does not support 3D graphics. However, you can display more than two dimensions in a plot by assigning variables in the dataset to colors, shapes, line types, etc. These are all examples of *aesthetic attributes*. An aesthetic attribute is a visual property that affects the way observations are displayed in the plot. The `x` and `y` positions are aesthetic attributes.

Assigning a variable from the dataset to an aesthetic is always done within the `aes` function. A term commonly used for the process of assigning variables to aesthetics is *aesthetic mapping*. We'll use this term throughout the rest of the workshop.

Let's make our scatterplot again, but this time we will map a different color to points for cars in different cylinder categories. We'll do this by mapping the `color` aesthetic to the `cyl` variable. Again we will need to use `factor(cyl)` to indicate that `cyl` should be considered categorical instead of continuous.

One of the convenient things about **ggplot2** is that we when map aesthetics to variables we get legends automatically. When making quick exploratory graphics I always use the default colors that **ggplot2** chooses. Later today we'll see that we can change these defaults by using some of the `scale` functions.

```
ggplot(mtcars, aes(x = wt, y = mpg, color = factor(cyl) ) ) +
    geom_point()
```



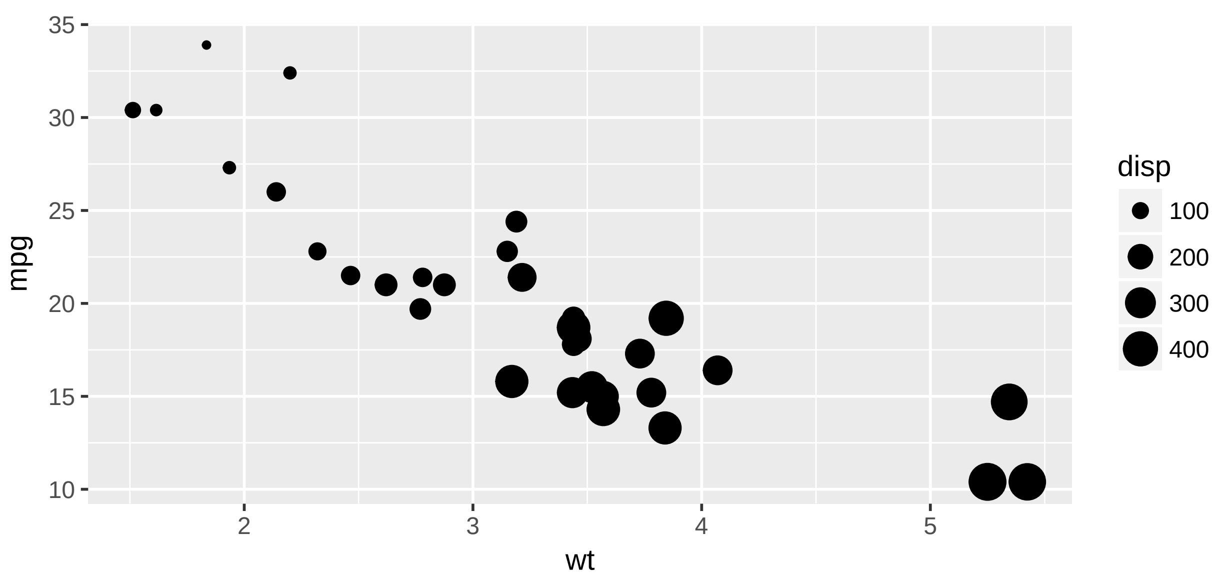
Continuous variables and aesthetic mapping

In the plots we are making today we will be only mapping categorical variables to aesthetics. You can map some aesthetics to continuous variables, though. For example, heat maps and bubble plots are created by mapping the aesthetics `color` and `size`, respectively, to continuous variables.

Note that some aesthetics can never be used with continuous variables. For example, `shape` and `linetype` can only be used with categorical variables as those aesthetics don't have a continuous nature. You would get an error message if you attempted to use continuous variables with either of those.

To see how aesthetic mapping with a continuous variable works, we'll make a bubble plot by mapping the `size` aesthetic to the continuous variable `disp` in our scatterplot. As always, the aesthetic mapping takes place inside `aes`.

```
ggplot(mtcars, aes(x = wt, y = mpg, size = disp) ) +
    geom_point()
```



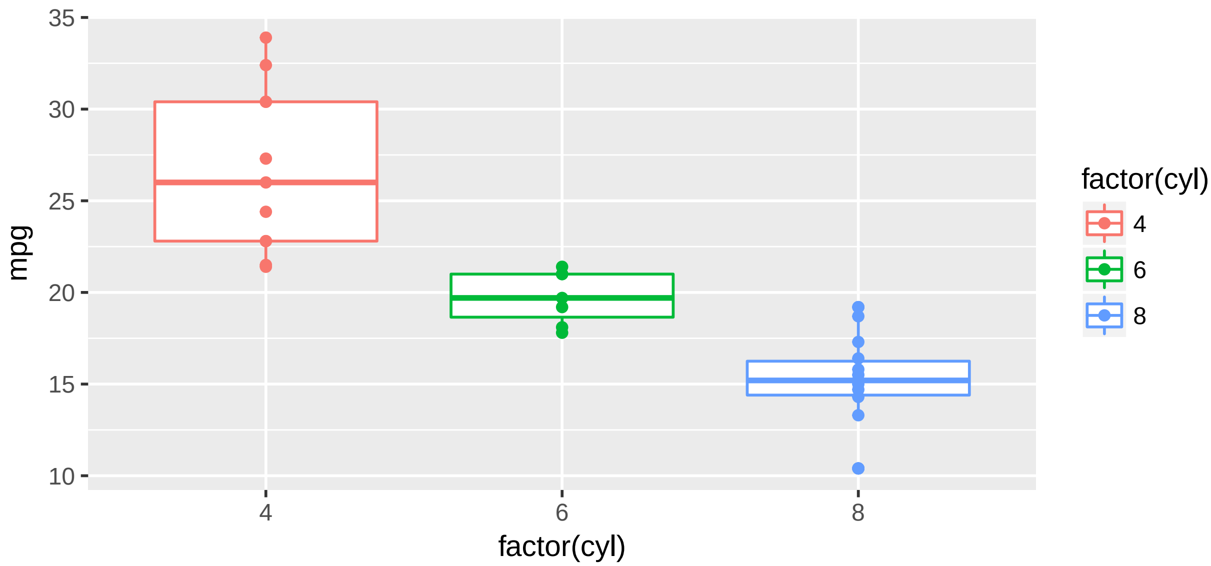
Adding more layers

Not only can we add additional aesthetics, we can also add additional geoms to a plot by adding more layers. This gives us a lot of freedom to make really informative graphics.

Let's add the data points on top of the boxplots we made by adding a `geom_point` layer. In addition, let's map `color` to the number of cylinders. We put the aesthetic mapping in `ggplot`, which causes both the boxplots and points to be colored by `cyl`. This is called mapping an aesthetic *globally*, as it affects all layers in the plot.

Notice that when we use `color` for plots such as boxplots, the `color` changes the outline color and not the color of the inside of the box. We would use the `fill` aesthetic for changing the color of the inside of the boxes.

```
ggplot(mtcars, aes(x= factor(cyl), y = mpg, color = factor(cyl) ) ) +
    geom_boxplot() +
    geom_point()
```



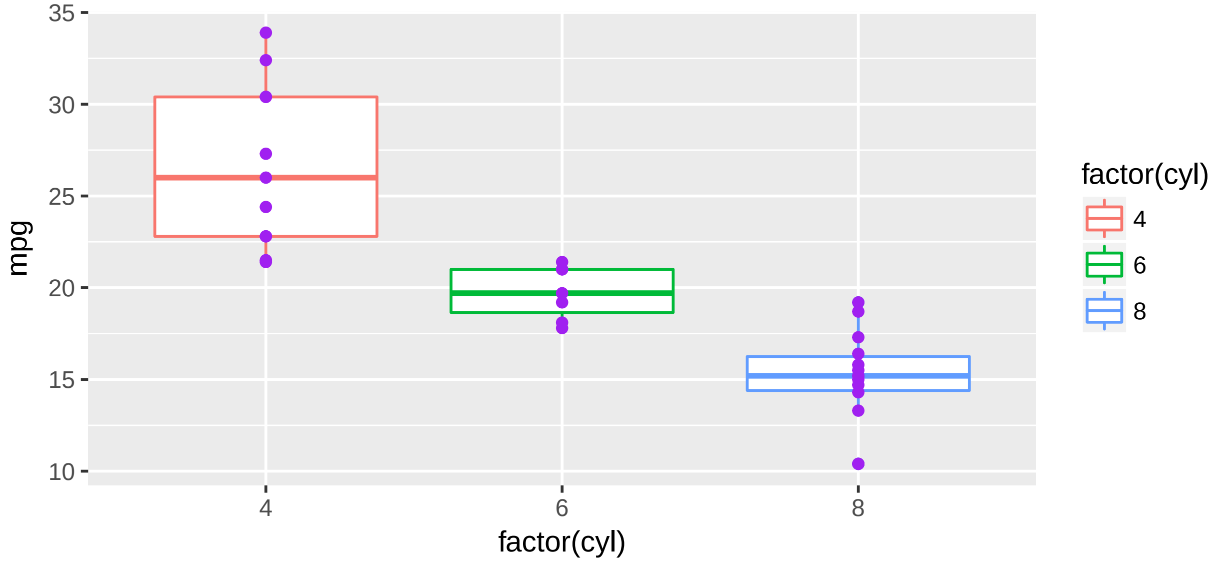
Setting aesthetics to constants within layers

We can add arguments to individual geom layers. In the next graphic, we will map the colors with `cyl` again for the boxplots, but we will *set* the color of the points to purple. We can do this by adding the `color` argument to `geom_point`. When we add an aesthetic to a geom that we've already defined within `ggplot`, the global `ggplot` aesthetic gets overridden *for that geom only*.

It is important to recognize that the argument within `geom_point` sets the aesthetic to a constant value (purple) instead of mapping to a variable. This means that we use `color` outside of the `aes` function for the first time. When we set aesthetics to constants the legends are not affected.

Knowing when to define aesthetics like `color` inside or outside the `aes` function can be confusing when you first start working with **ggplot2**. Generally, if you are mapping an aesthetic to a variable from your dataset you do it inside `aes` but if you are setting an aesthetic to a constant value you do it outside `aes`.

```
ggplot(mtcars, aes(x = factor(cyl), y = mpg, color = factor(cyl) ) ) +
    geom_boxplot() +
    geom_point(color = "purple")
```



Mapping aesthetics separately for different geoms

So far we've been using aesthetic mapping only within `ggplot`. We can map aesthetics to variables within specific geoms, as well.

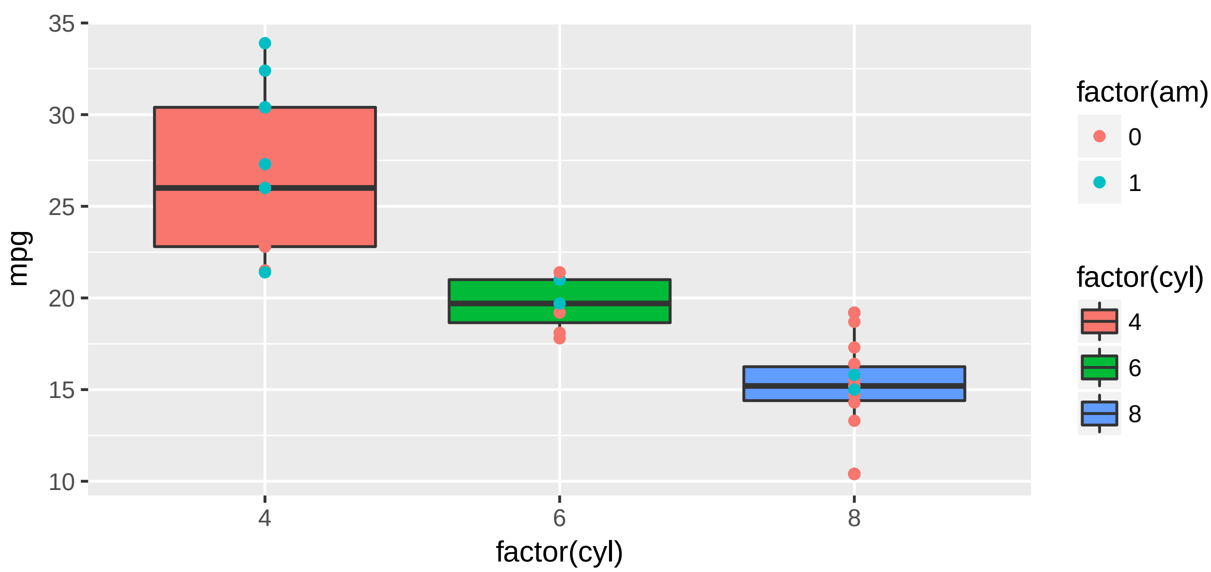
As mentioned earlier, when we map aesthetics in `ggplot` they are *global*, so those aesthetics mappings are used in all the layers throughout the graphic. We just saw that we can override global aesthetics within a geom if we need to. There are times when making complicated graphics that we might not want to map aesthetics globally because we'd have to override them in too many subsequent layers. In this case, we'd do the aesthetic mapping within specific geom layers instead.

In the next graph we will map the fill color for the boxplots to `cyl` and the color of the points to `am` (transmission type). We don't want to color the boxplot outline by `am`, though, so we map the aesthetics separately within the geom layers rather than globally in `ggplot`. Notice we do the mapping within the `aes` function still, but we did it within each layer instead of in `ggplot`.

In this example, `x` and `y` are mapped globally while `fill` is mapped to `cyl` only for the boxplot layer and `color` is mapped to `am` only for the points layer.

Mapping aesthetics to different variables means we'll get more legends by default.

```
ggplot(mtcars, aes(x = factor(cyl), y = mpg) ) +
    geom_boxplot( aes(fill = factor(cyl) ) ) +
    geom_point( aes(color = factor(am) ) )
```



Edit the dataset to change the graphic

This is a good place for us to talk a bit about the relationship between the dataset and the graphic. In the philosophy behind **ggplot2**, the dataset and the graphic go hand in hand. Making changes to the dataset is often the most straightforward way to make changes to the resulting graphic. The answer to the common help forum question "How do I change the appearance of xxx in **ggplot2**?" is very often "Make changes to your dataset."

In the example dataset we've been using, the two categorical variables we've used are considered numeric in the `mtcars` dataset so we've had to define them as factors every time we use them. In addition, the levels of the `am` variable are not useful for discerning what color represents an automatic vs manual transmission and so that legend is particularly uninformative.

Let's take a moment to go back to the `mtcars` dataset to define `cyl` and `am` as factors and make informative names for the levels of `am` (remember `0` in `am` stands for an automatic transmission). We'll call the new cylinder variable `numcyl`.

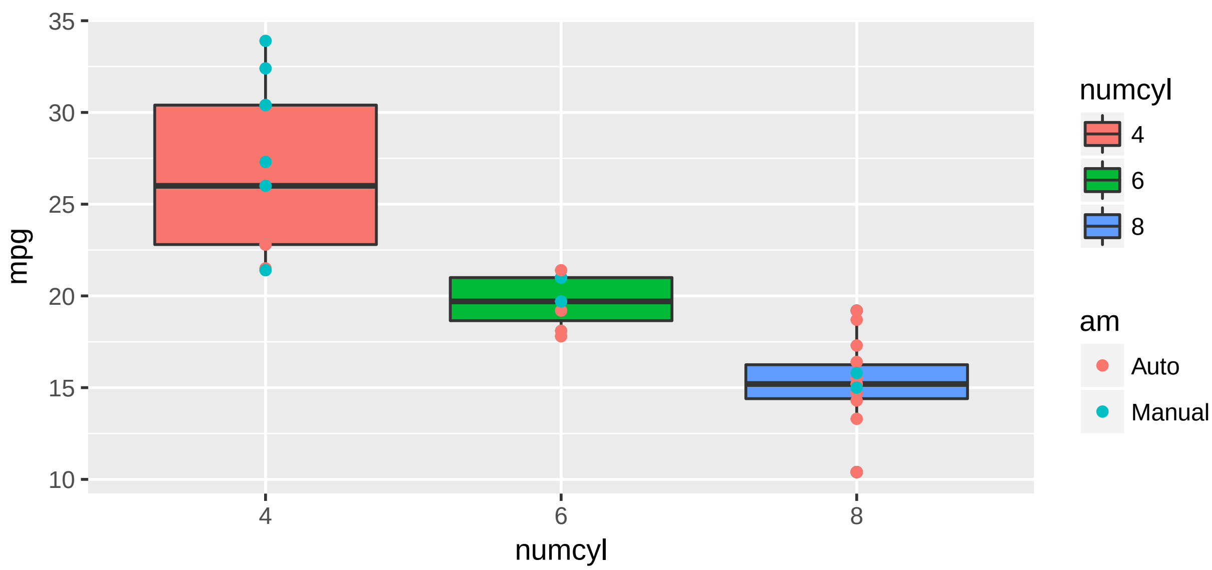
```
mtcars$numcyl = factor(mtcars$cyl)
mtcars$am = factor(mtcars$am, labels = c("Auto", "Manual") )

str(mtcars)
```

```
'data.frame':   32 obs. of  12 variables:
 $ mpg   : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl   : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp  : num  160 160 108 258 360 ...
 $ hp    : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat  : num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt    : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec  : num  16.5 17 18.6 19.4 17 ...
 $ vs    : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am    : Factor w/ 2 levels "Auto","Manual": 2 2 2 1 1 1 1 1 1 1 ...
 $ gear  : num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb  : num  4 4 1 1 2 1 4 2 2 4 ...
 $ numcyl: Factor w/ 3 levels "4","6","8": 2 2 1 2 3 2 3 1 1 2 ...
```

After those changes, let's recreate the last graphic we were working on using the edited variables.

```
ggplot(mtcars, aes(x = numcyl, y = mpg) ) +
    geom_boxplot( aes(fill = numcyl) ) +
    geom_point( aes(color = am) )
```



Choosing colors for `color` and `fill`

We won't be working with colors when we switch from exploratory graphics to creating polished graphics, so I thought I'd give a quick example here of changing the color scheme away from the default colors. You can change the *values* for any aesthetic by using the appropriate *scale* function. For example, to change the `fill` and `color` colors away from the defaults, we can use `scale_fill_manual` and `scale_color_manual`, respectively.

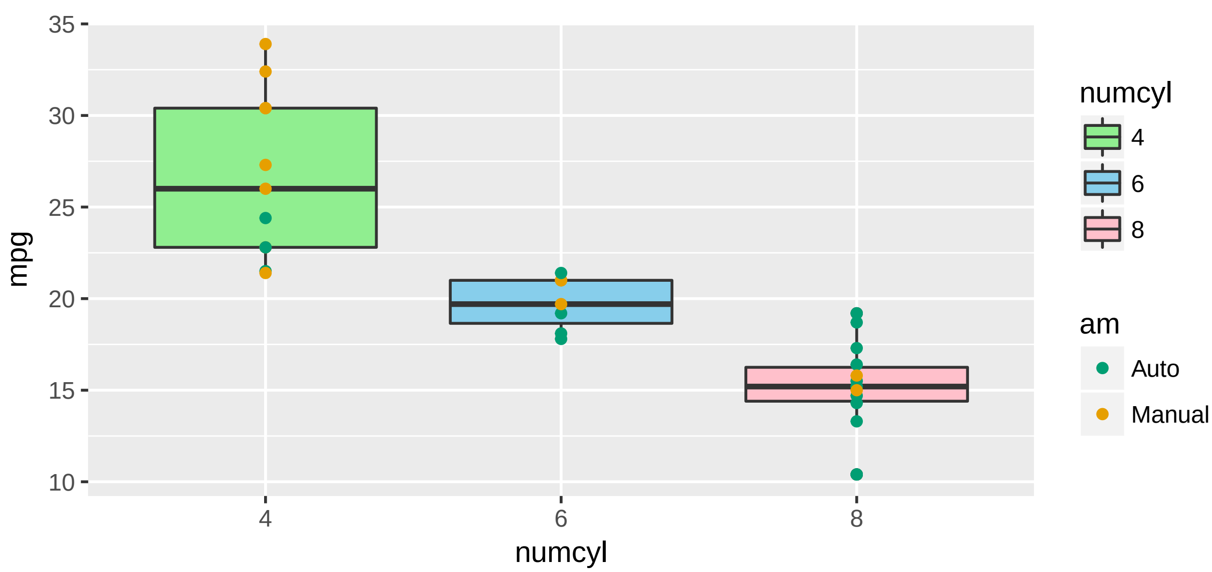
We change the default colors to the desired colors using the `values` argument. New colors are assigned in the order of the factor variable levels that we mapped the aesthetic to. The levels of the `numcyl` variable are ordered `4`, `6`, `8` and the levels of the `am` variable are `Auto`, `Manual`. See the `limits` argument for changing the order of the factor levels within the scale functions (we will not cover this today).

There is some nice information about available colors and color palettes at the Cookbook to R site, http://www.cookbook-r.com/Graphs/Colors_(ggplot2)/.

Let's change the fill colors to light green, sky blue, and pink using color names, and the point colors to a green and an orange using hexadecimal string names I pulled off the link I gave above.

Make sure you use the correct `scale` for the aesthetic you are using in your graphic. If you map a variable to `fill`, don't expect `scale_color_manual` to change the `fill` colors.

```
ggplot(mtcars, aes(x = numcyl, y = mpg) ) +
    geom_boxplot(aes(fill = numcyl) ) +
    geom_point(aes(color = am)) +
    scale_fill_manual(values = c("light green", "sky blue", "pink") ) +
    scale_color_manual(values = c("#009E73", "#E69F00") )
```



Dot plot example

There are many, many other geoms available in **ggplot2**, far too many to cover today. We'll hit a few more before moving on.

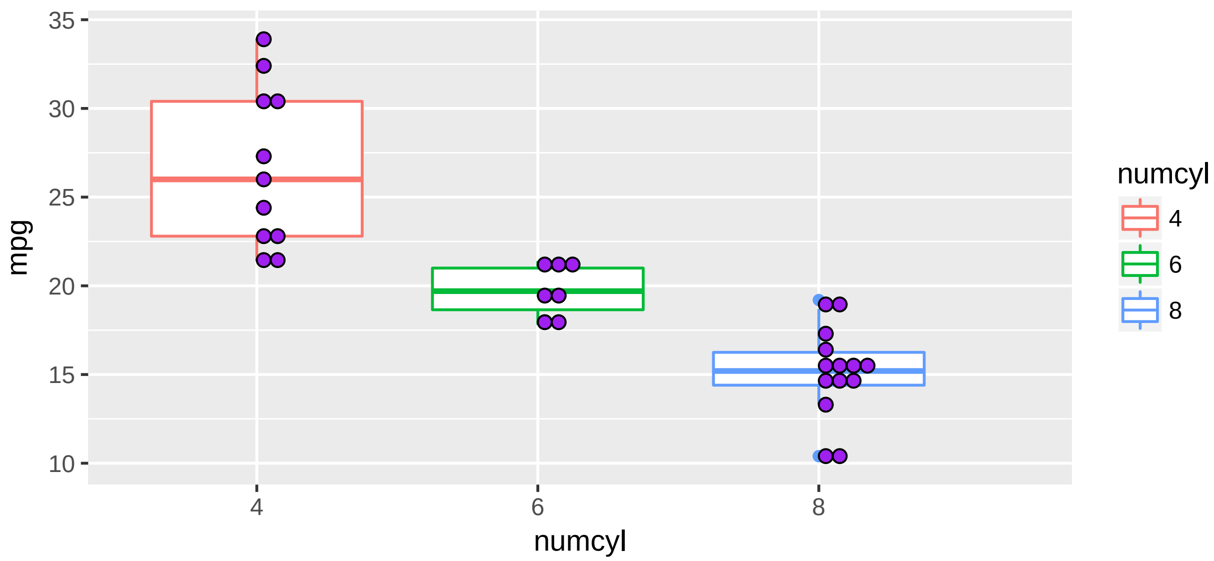
One useful graphic not everyone has seen before is a dot plot. Dot plots are essentially a type of histogram made using dots. This is an alternative way to show the raw data values along with a boxplot (which shows summary statistics).

Let's put a dot plot on top of our boxplot using `geom_dotplot`. Here we need to bin along the y axis, so this will look like a histogram turned on its side. You always get a message about the `binwidth` for dot plots and histograms. This is just information and does not indicate an error.

We'll set the dot fill color to purple, which is done outside of `aes` as it involves setting an aesthetic to a constant value. In dot plots, `color` changes the outline color of the dots and `fill` changes the inside color.

```r
ggplot(mtcars, aes(x = numcyl, y = mpg) ) +
    geom_boxplot(aes(color = numcyl) ) +
    geom_dotplot(fill = "purple", binaxis = "y")
```

```
`stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```

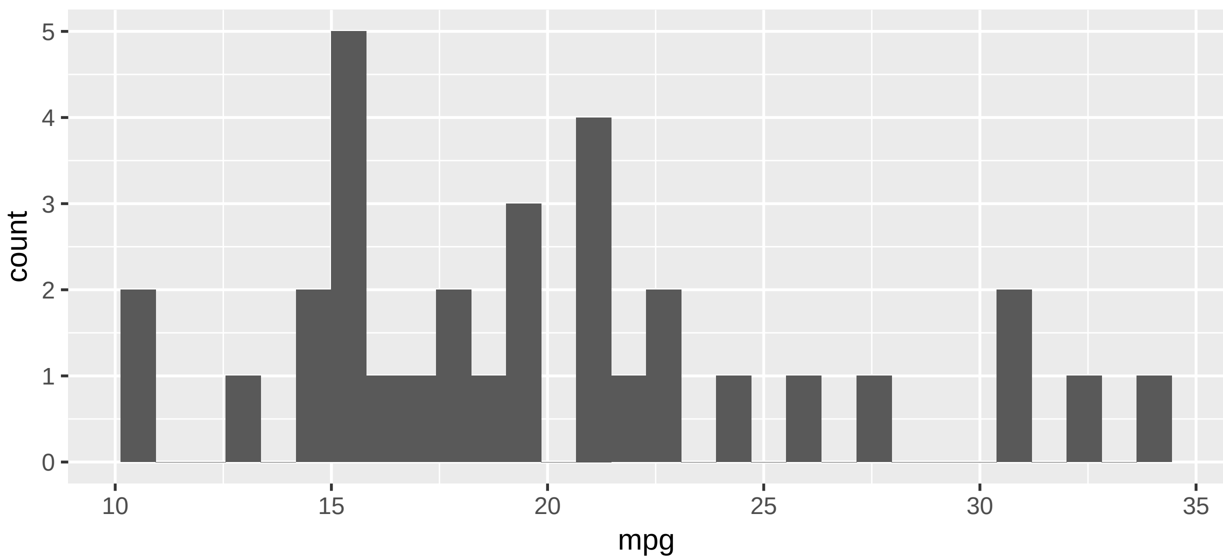


Histograms and density plots

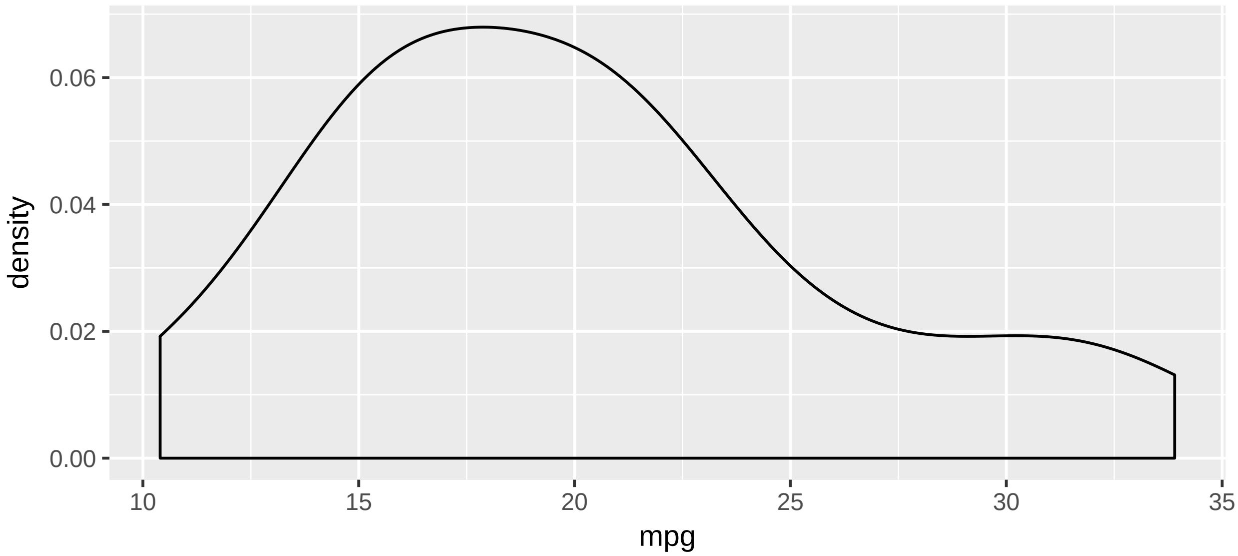
Histograms and density plots are both commonly used when making exploratory graphics of single variables. You can think of a density plot of a sort of smoothed histogram (although this is not the formal definition).

We only need to define the x axis for these plots, as the y axis is either a count or a density and **ggplot2** calculates those for us. Let's make a histogram and a density plot of `mpg`.

```r
ggplot(mtcars, aes(x = mpg) ) +
    geom_histogram()
```

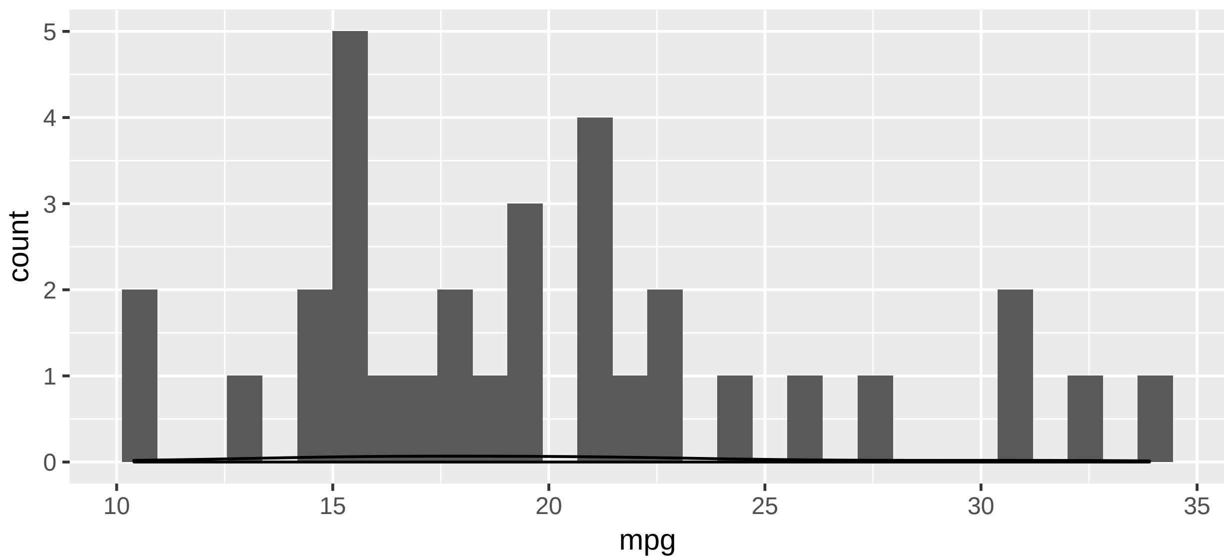



```r
ggplot(mtcars, aes(x = mpg) ) +
    geom_density()
```



We can also make a histogram with a density overlay. Be careful, though. Look at the following graphic.

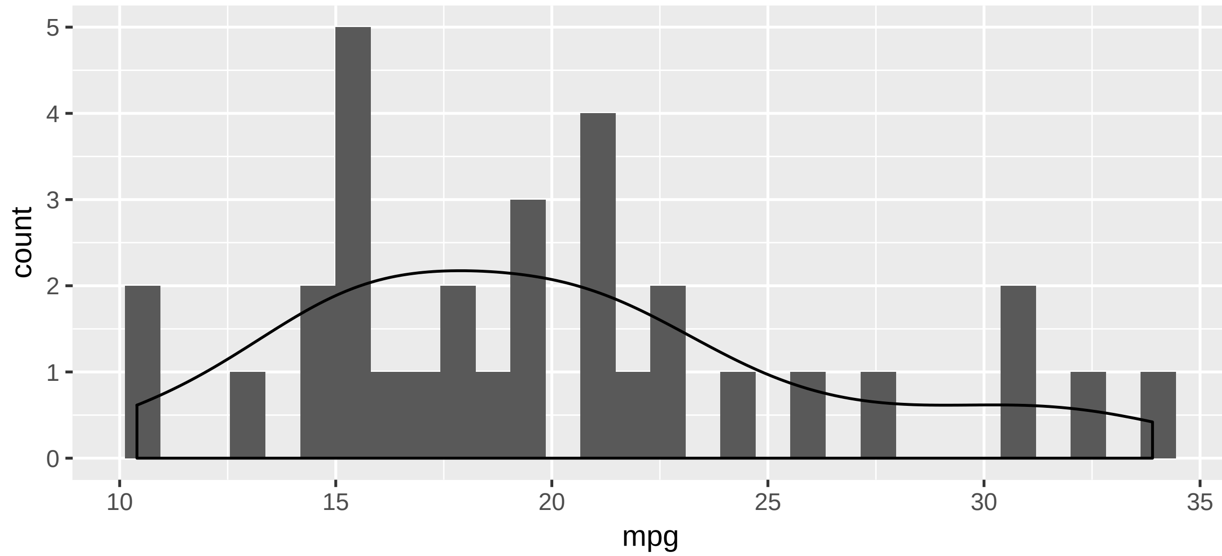
```
ggplot(mtcars, aes(x = mpg) ) +
    geom_histogram() +
    geom_density()
```



By default, histograms and density plots are on different scales. If we want both on the same plot we'd need to get both layers on the same scale, showing either counts or densities. There are special variables that these two geoms create, named `..count..` and `..density..`, that we can use to do this. We can map our `y` variable to one of these special variables. As this involves aesthetic mapping, we have to do this inside `aes`. Don't forget those two periods before and after the words or you will get an error message.

Let's put the density curve on the scale of counts and try the plot again.

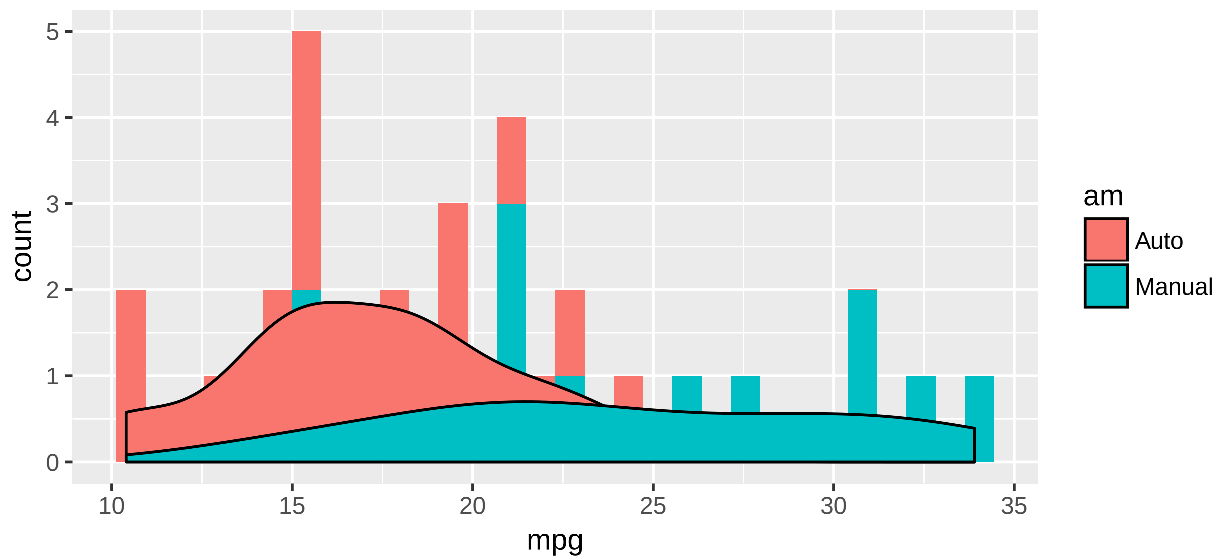
```
ggplot(mtcars, aes(x = mpg) ) +
    geom_histogram() +
    geom_density(aes(y = ..count..) )
```



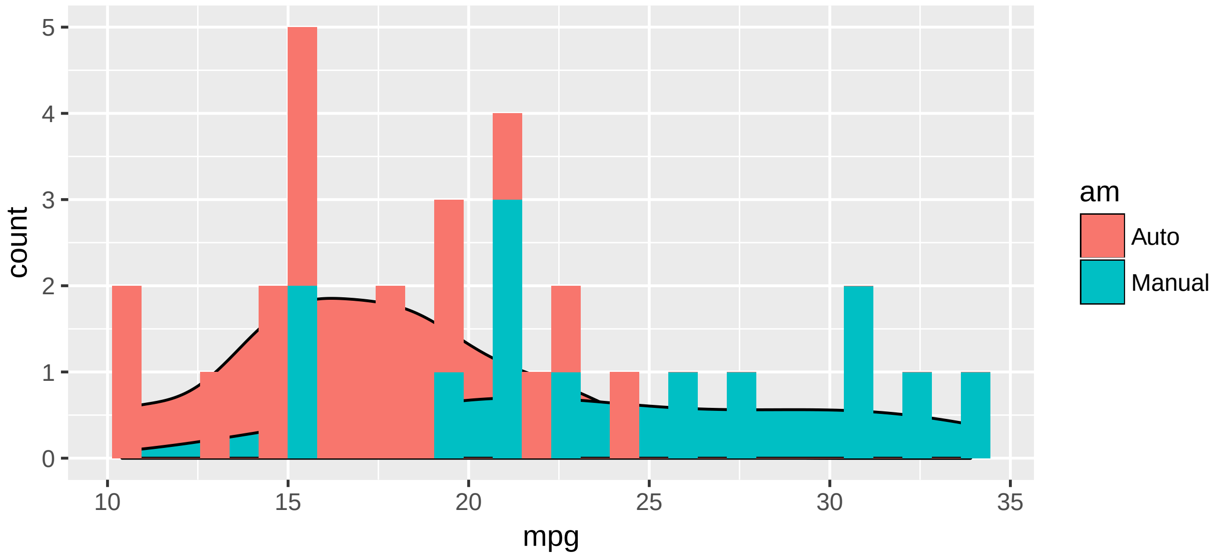
Layer order

The order we add the geom layers matters. Look what happens when we map `fill` to `am` for all the layers and change the order in which we add the histogram and density layers. The layer added last is always on top.

```
ggplot(mtcars, aes(x = mpg, fill = am) ) +
    geom_histogram() +
    geom_density(aes(y = ..count..) )
```



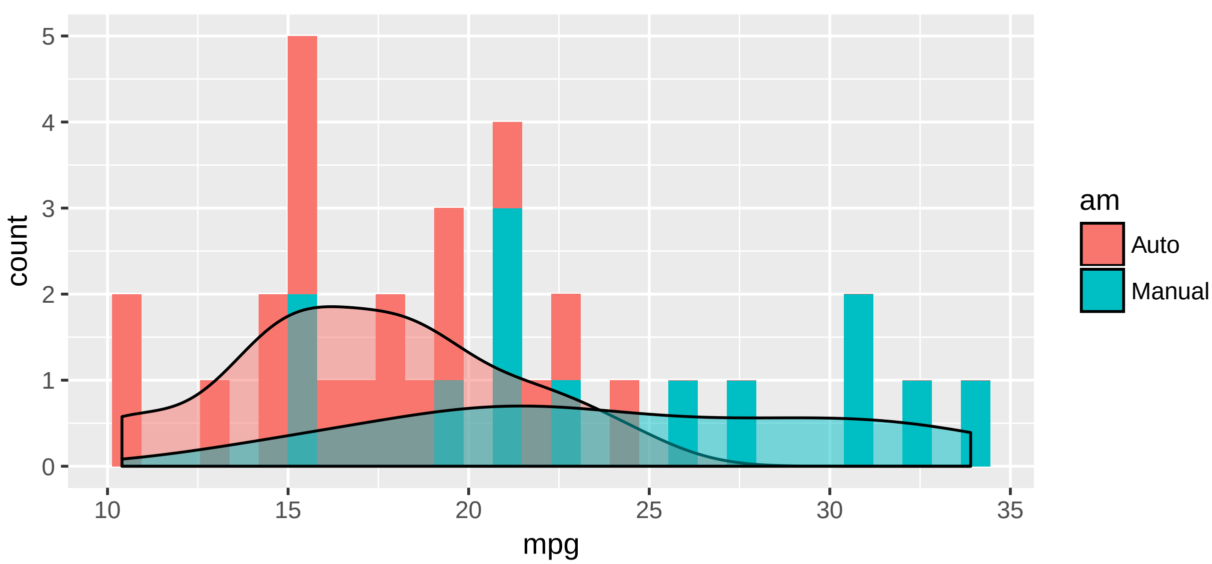

```
# Change the order of the geom layers
ggplot(mtcars, aes(x = mpg, fill = am) ) +
    geom_density(aes(y = ..count..)) +
    geom_histogram()
```



These filled graphics aren't particularly nice looking. The graphics become slightly more useful if we allow the color of the fill to be more transparent in the density layer. We control transparency using the `alpha` aesthetic. We can set `alpha` to be between 0 and 1, with 0 being completely transparent and 1 completely opaque.

We will set `alpha` to a constant inside `geom_density`. Because we are not aesthetic mapping, we will do this outside of `aes`. You can map the `alpha` aesthetic to variables, although we won't do this today.

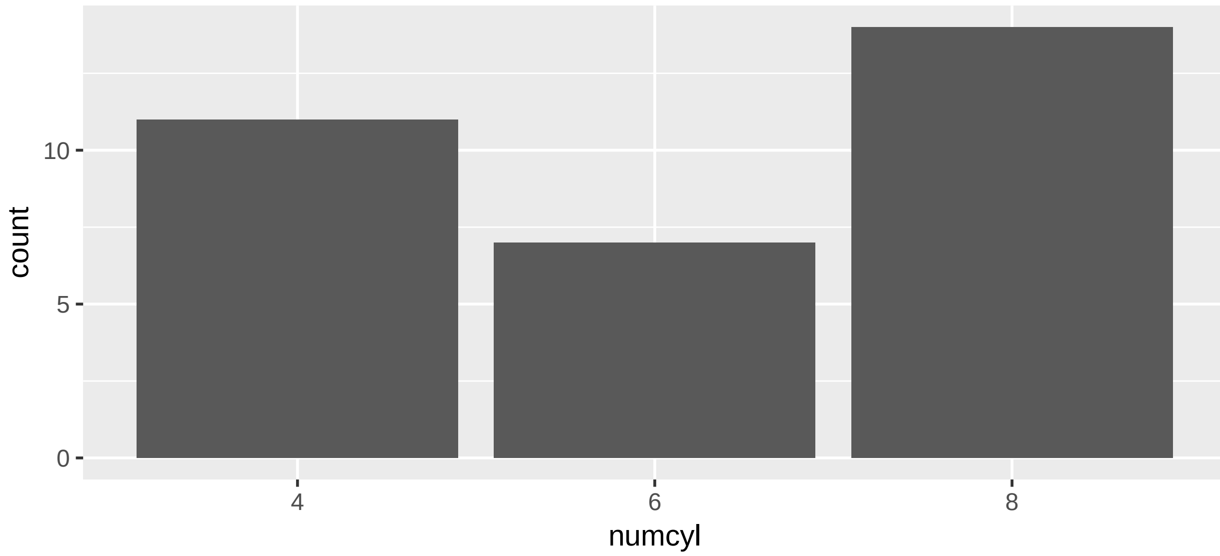
```
ggplot(mtcars, aes(x = mpg, fill = am) ) +
    geom_histogram() +
    geom_density(aes(y = ..count..), alpha = .5)
```



Bar graphs

Bar graphs are closely related to histograms, but involve graphing counts of a categorical variable instead of binning a continuous variable. You can map values other than counts to the bars using the `y` aesthetic, but `geom_bar` defaults to counts. See `geom_col` for mapping `y` to a variable.

```
ggplot(mtcars, aes(x = numcyl) ) +
    geom_bar()
```

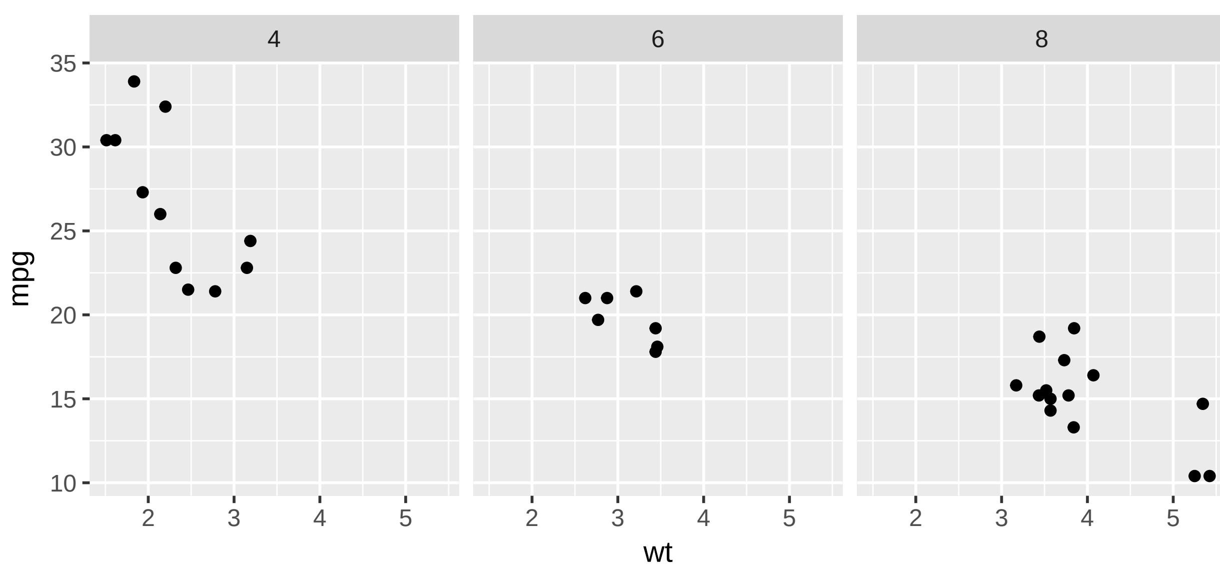


Facets

Sometimes we want to create plots for different groups in different panes within the same graphic. In **ggplot2** lingo this is called *faceting*. There are two faceting functions, `facet_wrap` and `facet_grid`. They actually can be quite different, so if you're doing faceting you should look at the help page examples for each. Today we're working with `facet_wrap`.

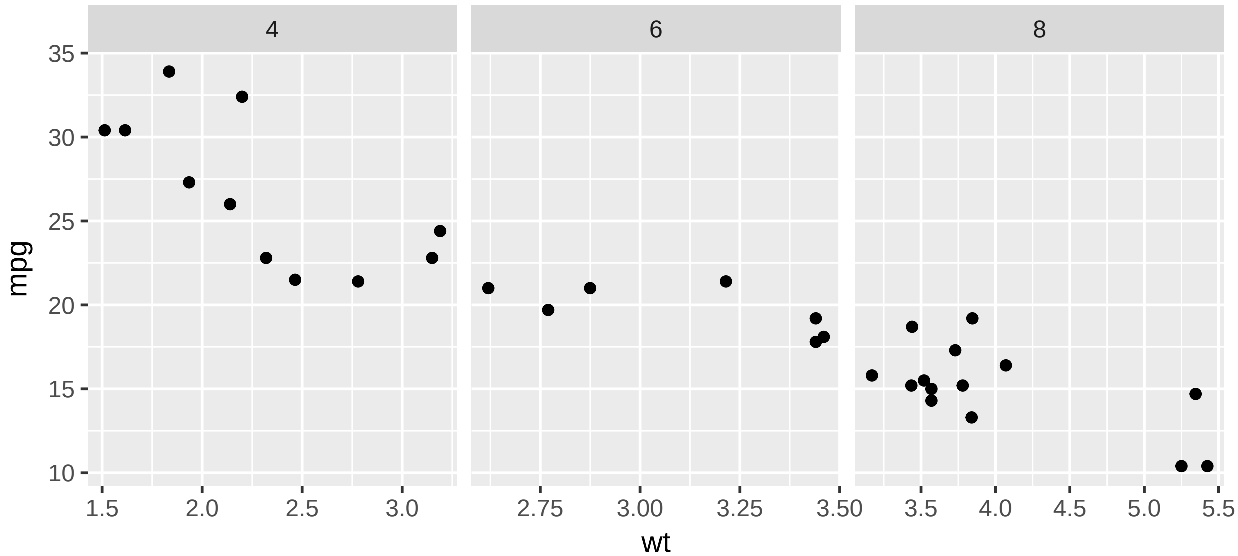
Here is the `mpg` vs `wt` scatterplot, with a separate plot for each level of `numcyl`.

```
ggplot(mtcars, aes(x = wt, y = mpg) ) +
    geom_point() +
    facet_wrap(~numcyl)
```



The three panes have the same x and y axis by default. That behavior isn't always desirable, and one or both axes can be allowed to vary using the `scales` argument with `"free"`, `"free_x"`, or `"free_y"`. Let's allow the x axis to vary among facet panels.

```
ggplot(mtcars, aes(x = wt, y = mpg) ) +
    geom_point() +
    facet_wrap(~numcyl, scales = "free_x")
```



Adding layers using summary statistics

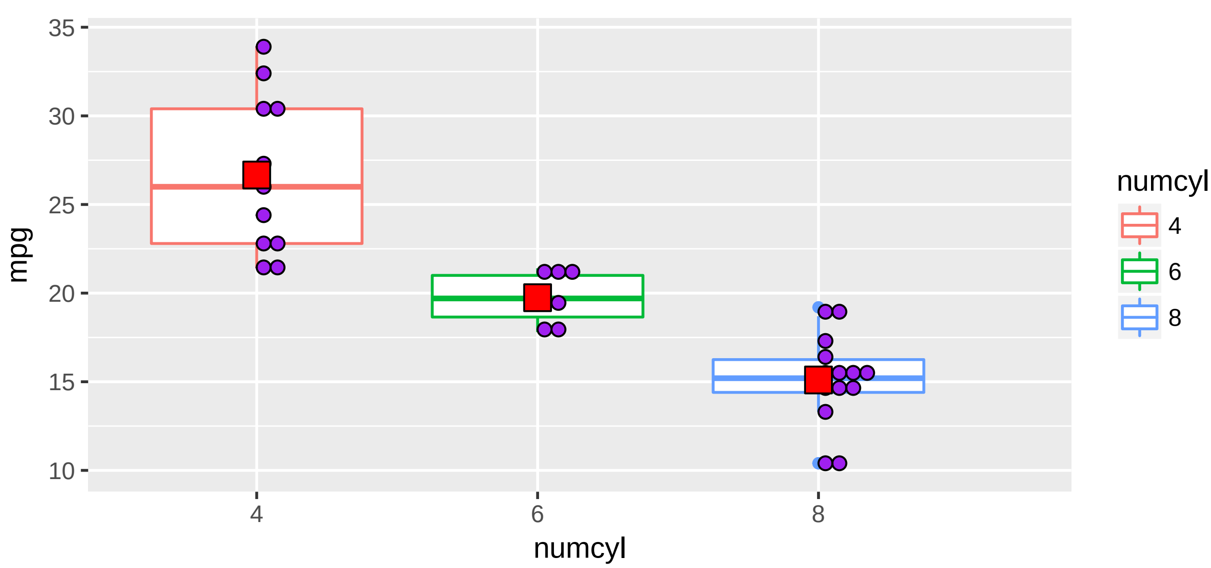
The function `stat_summary` can be used to add layers based on simple summary statistics. To see how this works, let's add mean `mpg` by group to our boxplot/dot plot graphic as a large red square.

We have to define which axis we want the summary statistics for (here `mpg` is on the y axis, so we use `fun.y`) and which geom we want to use to plot the summary statistics. In this case we'll use the `point` geom to add the means as points. Notice that all aesthetics we use in `stat_summary` are set to constants and not mapped to variables so we define them outside of `aes`.

Summary information can also be added to a graphic by creating a summary dataset and then using that summary dataset when adding additional layers to a graphic. We will see this approach later in the workshop.

I generally can't remember which numbers correspond to which shapes in R. There is a nice cheat sheet on Cookbook for R: http://www.cookbook-r.com/Graphs/Shapes_and_line_types/. Note that shapes 21 through 25 are *fillable*, meaning you can color the inside with `fill` and the outline with `color`.

```
ggplot(mtcars, aes(x = numcyl, y = mpg) ) +
    geom_boxplot(aes(color = numcyl)) +
    geom_dotplot(fill = "purple", binaxis = "y") +
    stat_summary(fun.y = mean, geom = "point", size = 5, shape = 22, fill = "red")
```

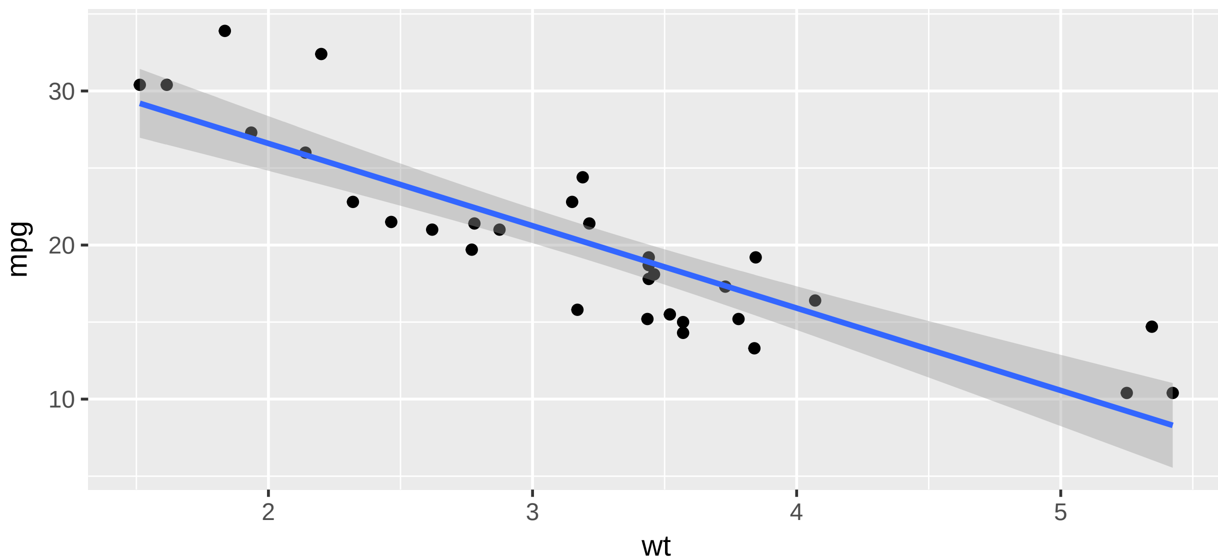


Adding regression lines

Along the same lines as summary statistics, it is easy to add regression lines to graphics using `geom_smooth` or `stat_smooth`. Let's add linear regression lines to a scatter plot using `geom_smooth`. We have to set the `method` to `"lm"` to get regression lines. The default method of `geom_smooth` is a `"loess"` line, which can also be useful when exploring your data.

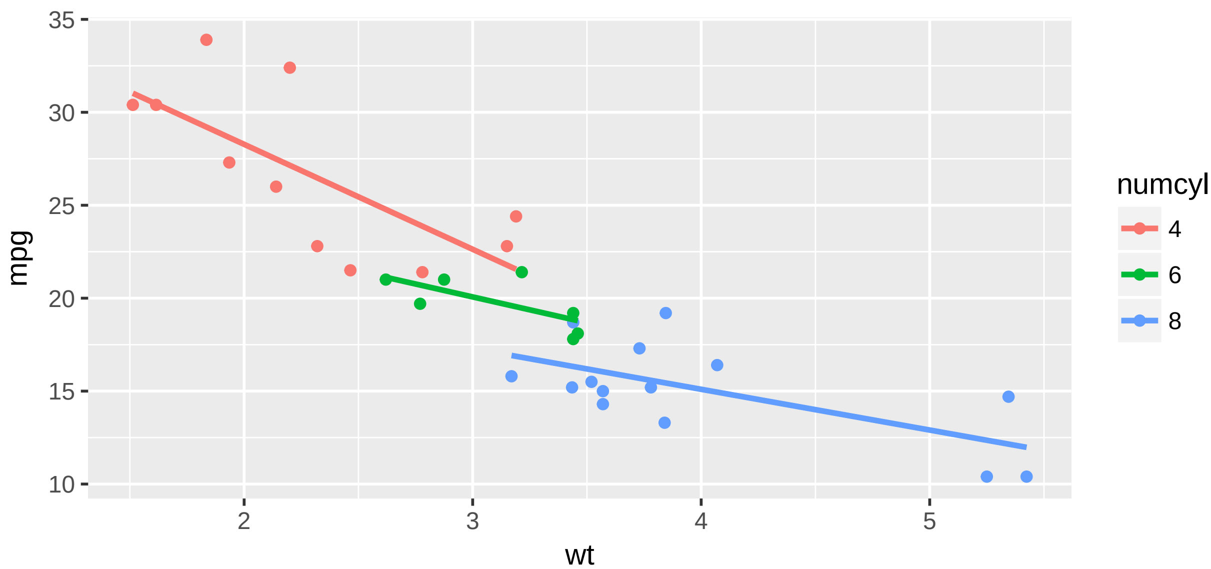
By default `geom_smooth` calculates a confidence envelope around the line, which isn't usually reasonable when working on exploratory graphics.

```
ggplot(mtcars, aes(x = wt, y = mpg) ) +
    geom_point() +
    geom_smooth(method = "lm")
```



We can fit regression lines separately for each group by mapping an aesthetic such as `color` to a categorical variable. We'll map `color` to `numcyl` to get a separate regression line for each cylinder category. Let's remove the confidence envelope in the next plot by using the argument `se = FALSE`. We could always add it back after we checked our assumptions and knew it was appropriate.

```
ggplot(mtcars, aes(x = wt, y = mpg, color = numcyl) ) +
    geom_point() +
    geom_smooth(method = "lm", se = FALSE)
```



Polishing ggplot2 graphics

Now that we've covered some of the basics of **ggplot2**, we are going to spend time making two "final" graphics. These are graphics that you might include in a thesis or manuscript or in a presentation. Such graphics need look nicer than the basic exploratory graphics you make just for yourself, and you will have to spend time tweaking the appearance of a graph until is is just right.

The second half of this workshop focuses on some of the ways we can control the overall appearance of a **ggplot2** graphic. I decided to present you two examples that end up being fairly complex. While I could have chosen to make simpler graphs, seeing some graphics with complex elements may give you an idea if **ggplot2** will be a useful tool for you when creating this sort of graphic.

I'm not advocating either of these as what any of your own final graphics should look like, as a lot of that depends on personal preference and, potentially, manuscript requirements. These examples are really just to show you some more **ggplot2** options.

Polished graphic #1: A plot of the raw data

The first polished graphic we will create will be based on the data in the file `egg length and width by species.csv`. These data were used to compare mean egg length and mean egg width of two bird species using two two-sample t-tests. While a display of the results from simple tests like these would be unnecessary, we can give our audience a nice picture of what the data look like in a well thought-out graphic.

Make sure the data file is in your working directory and read it into R.

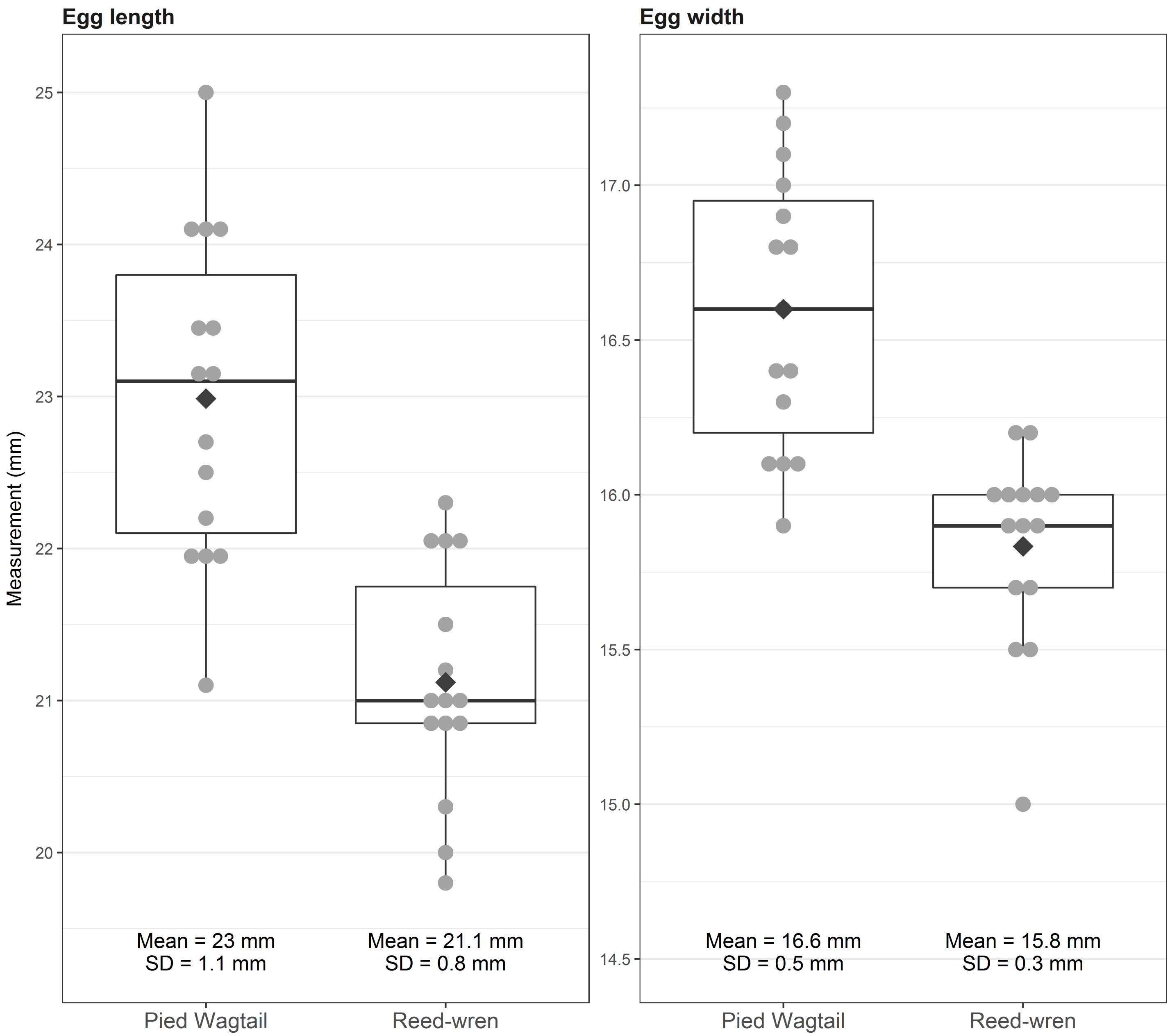
```
eggs = read.csv("egg length and width by species.csv")
head(eggs)
```

```
   id length width      species
1 198   23.1  16.4 Pied Wagtail
2 199   23.5  16.8 Pied Wagtail
3 200   24.1  17.1 Pied Wagtail
4 201   23.4  16.4 Pied Wagtail
5 202   23.2  16.8 Pied Wagtail
6 203   22.5  16.6 Pied Wagtail
```

```
str(eggs)
```

```
'data.frame':   30 obs. of  4 variables:
 $ id     : int  198 199 200 201 202 203 204 205 206 207 ...
 $ length : num  23.1 23.5 24.1 23.4 23.2 22.5 21.9 21.9 25 24.1 ...
 $ width  : num  16.4 16.8 17.1 16.4 16.8 16.6 16.1 16.1 16.9 15.9 ...
 $ species: Factor w/ 2 levels "Pied Wagtail",..: 1 1 1 1 1 1 1 1 1 1 ...
```

I provided you the final graphic we are working towards in `Workshop final graphic 1.pdf`, which is printed below so you can see what we are working towards.



Reshaping a dataset prior to graphing

You can see that I used faceting to put measurement type (`length` or `width`) in separate panes. However, the current dataset doesn't have a variable for us to facet with because `length` and `width` are in separate columns. We will need to reshape this wide dataset into long format using `melt` from package **reshape2** or `gather` from package **tidyr**. I've decided to show you the **tidyr** version today.

Using `gather`, we will create a new categorical variable representing measurement type in one column and all the quantitative measurement values in another. We define the names of the new columns we'll are making using the `key` and `value` arguments, and then list which columns we want to take and put into a single column.

Reshaping is often needed in order to take full advantage of **ggplot2**. Being comfortable with data manipulation in R is one key to success for creating **ggplot2** graphics.

We'll name the reshaped dataset `eggs2`. It is in long format, so it has twice as many rows as the original dataset `eggs`.

```
library(tidyr)
eggs2 = gather(eggs, key = type, value = measurement,
            length, width, factor_key = TRUE)
head(eggs2)
```

```
   id      species   type measurement
1 198 Pied Wagtail length        23.1
2 199 Pied Wagtail length        23.5
3 200 Pied Wagtail length        24.1
4 201 Pied Wagtail length        23.4
5 202 Pied Wagtail length        23.2
6 203 Pied Wagtail length        22.5
```

```
str(eggs2)
```

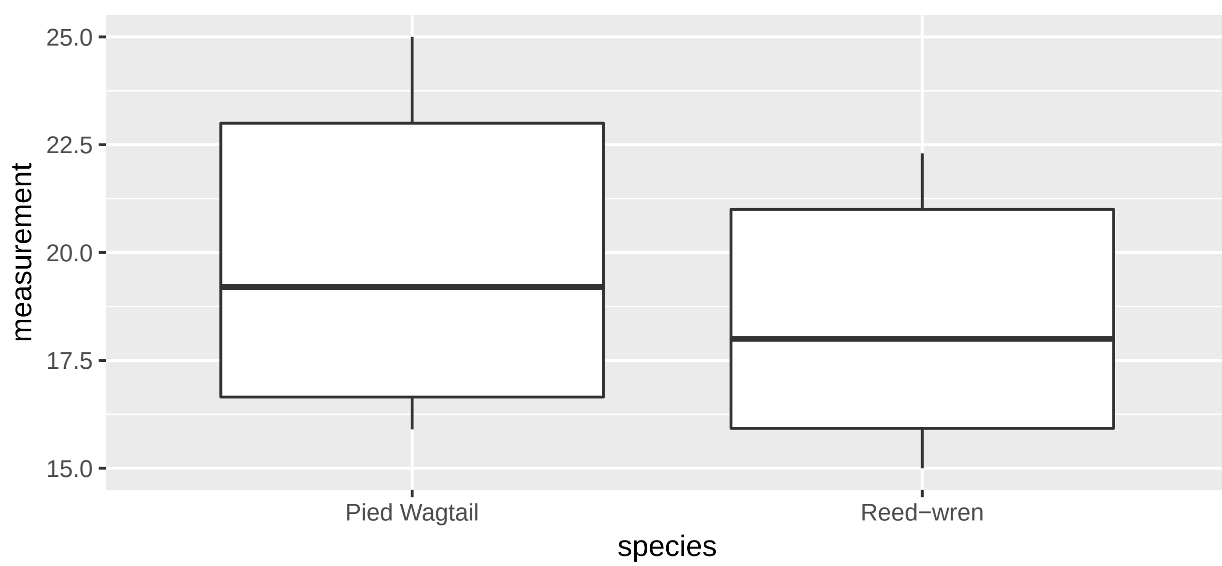
```
'data.frame':   60 obs. of  4 variables:
 $ id         : int  198 199 200 201 202 203 204 205 206 207 ...
 $ species    : Factor w/ 2 levels "Pied Wagtail",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ type       : Factor w/ 2 levels "length","width": 1 1 1 1 1 1 1 1 1 1 ...
 $ measurement: num  23.1 23.5 24.1 23.4 23.2 22.5 21.9 21.9 25 24.1 ...
```

I'm going to change the style of how I create the polished graphics for teaching purposes. You will likely see this style on help forums or in help documents, although I personally rarely use this style in my own graphics code.

This is how it will go: First I will create and name a basic graphic by defining the dataset and the x and y axes in `ggplot` and adding a geom layer. This first graphical object will be named `g1`. Then we will start to add layers to `g1`, renaming the graphic as `g1` each time we add a layer. This will allow us to see how the graphic changes layer by layer.

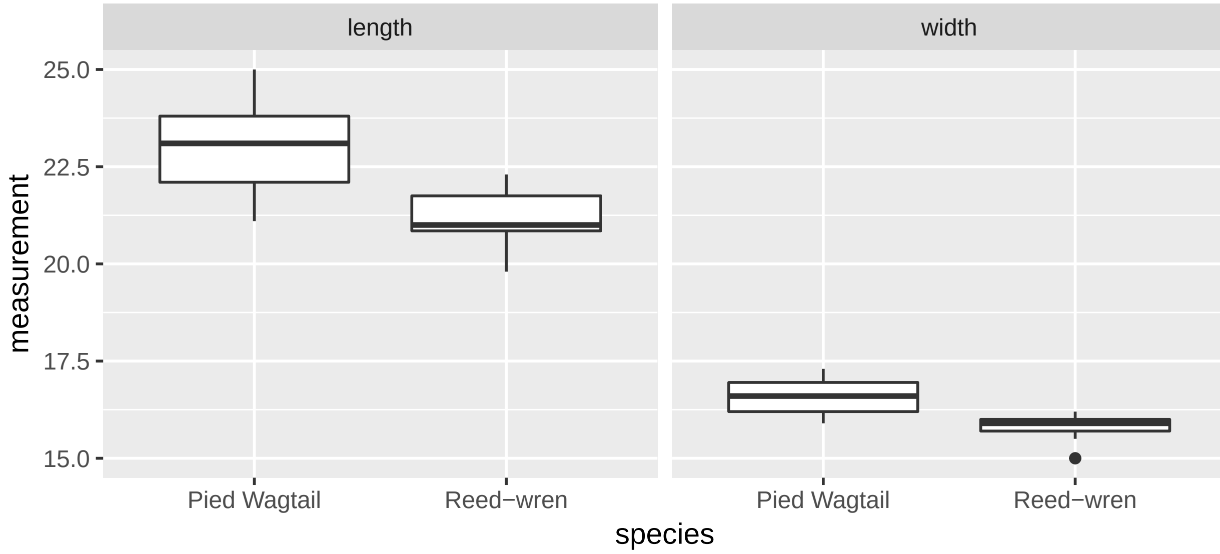
We will start with a simple boxplot, putting `species` on the x axis and `measurement` on the y axis. The extra pair of parentheses prints the graphic to the plotting pane so we can see the plot change as we go.

```
( g1 = ggplot(eggs2, aes(x = species, y = measurement) ) +
      geom_boxplot() )
```



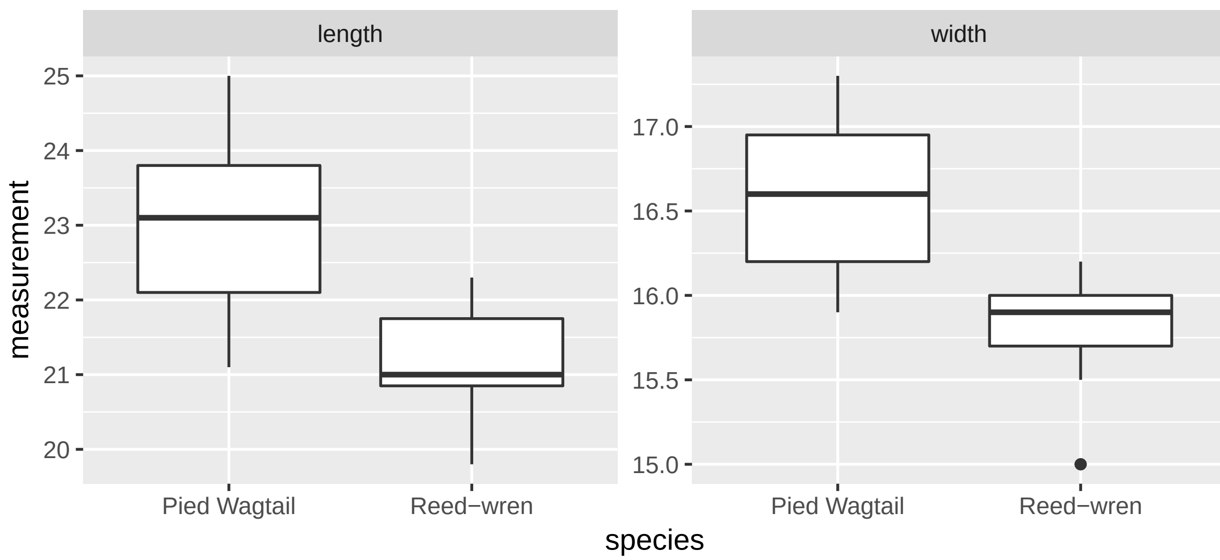
Let's add faceting by `type` to the graphic `g1`.

```
g1 + facet_wrap(~type)
```



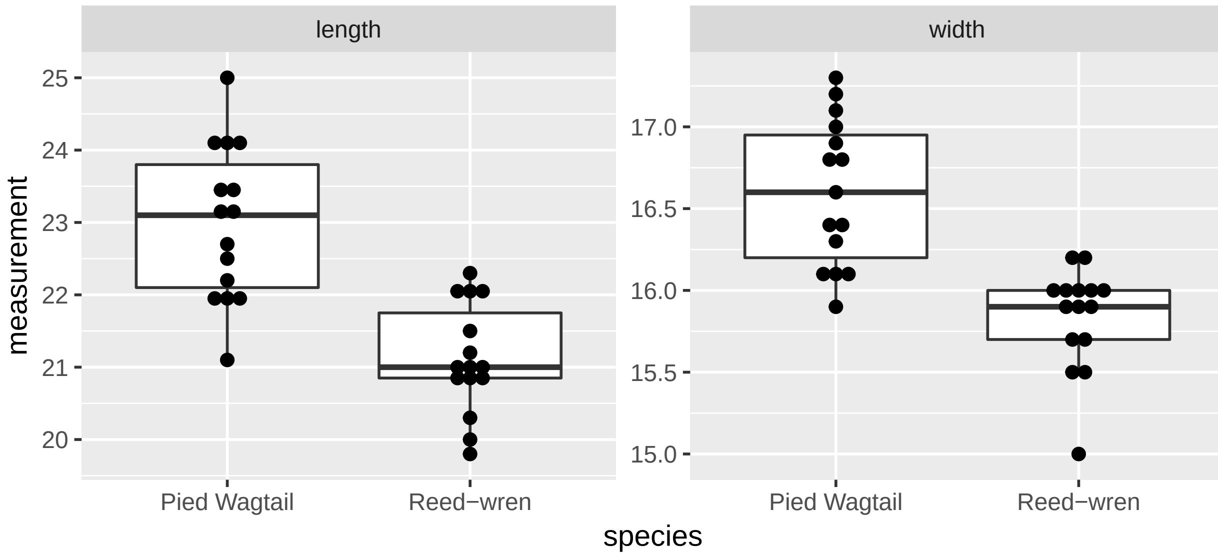
Not surprisingly, `length` and `width` cover fairly different ranges. It makes sense to allow the y axis to have different limits for the two different panes (i.e., we want to free the y axis scale). The x axis is the same for each pane, though, so we don't need to allow that to vary.

```
( g1 = g1 + facet_wrap(~type, scales = "free_y") )
```



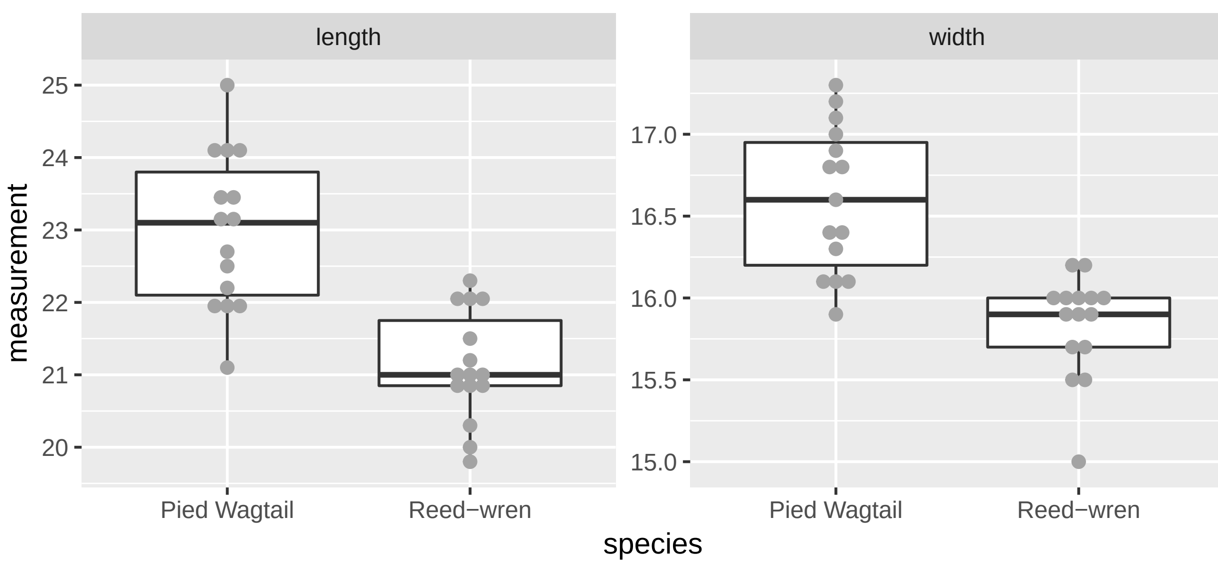
Adding the raw data on top of the boxplot as a dot plot will give a more complete picture of what the distribution of these data looks like. I use `stackdir = "center"` to make a centered dot plot, which is a little bit like a violin plot and a histogram.

```
g1 + geom_dotplot(binaxis = "y", stackdir = "center")
```



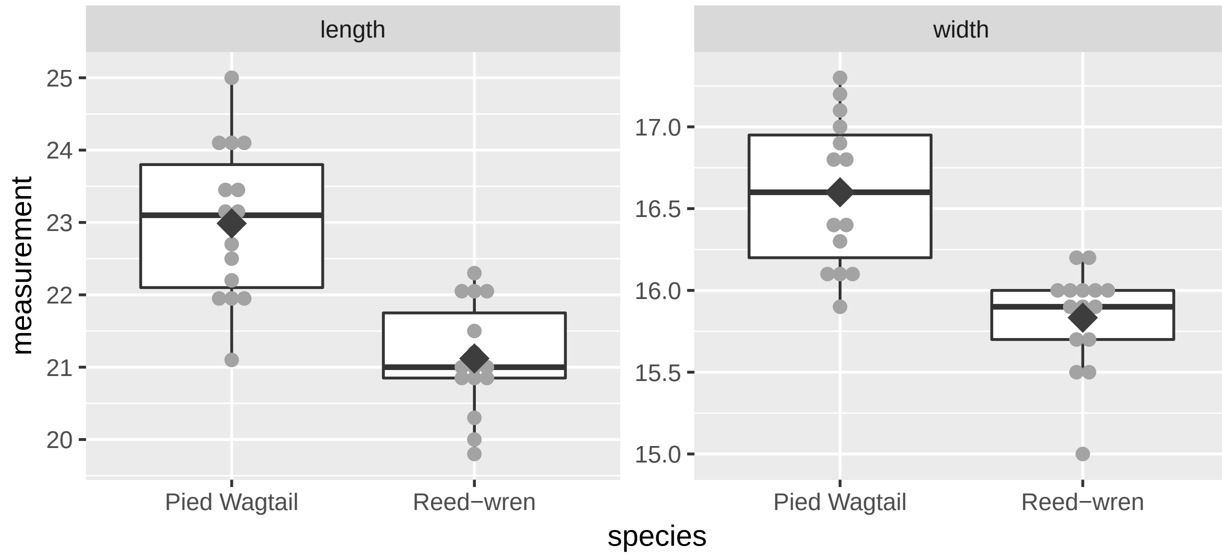
I thought the black dots were too dark, and decided to change the color to a lighter gray. Color choice is often a lot of trial and error for me, which I'm not going to show you here. I settled on a gray color called `grey64`. As we'll set both `fill` and `color` aesthetics to constants, these go outside of `aes`.

```
( g1 = g1 + geom_dotplot(binaxis = "y", stackdir = "center",
                         color = "grey64", fill = "grey64") )
```



Boxplots show medians but not means. It can be nice to add the means, especially when the statistical test used to analyze these data was about differences in means. Let's add the mean value for each `species` and `type` as diamonds, and fill them with a darker gray color, `grey24`. I didn't like how small these points were at first, so we'll increase the size of the diamonds, as well.

```
( g1 = g1 + stat_summary(fun.y = mean, geom = "point",
                         shape = 18, size = 5, color = "grey24") )
```



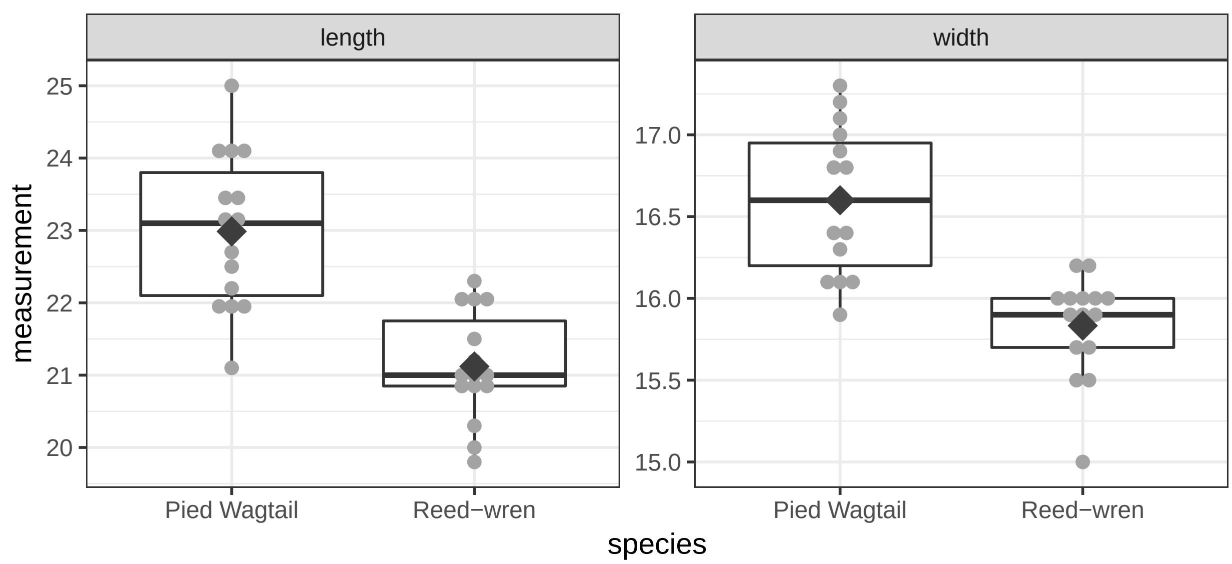
Changing plot appearance with `theme`

So far this is pretty much a review of things we did in the first part of the workshop. Let's change our focus to the overall appearance of the graphic, particularly the appearance of the panels and axes.

The default appearance of **ggplot2** graphics is called `theme_gray`. The default gray background of **ggplot2** can be nice for seeing differences in colors, but is not as useful if printing in black and white. If I'm going to include something in a printed document, I often use the built-in theme `theme_bw` to change the overall appearance of the graphic. This theme is called "theme black-and-white". You also might be interested in the built-in theme `theme_minimal`, which is a standard format in some fields. There are many themes out there - check out package *ggthemes* for more options.

While I could change all elements of the panel manually, I find `theme_bw` gives a nice starting point for further changes.

```
( g1 = g1 + theme_bw() )
```

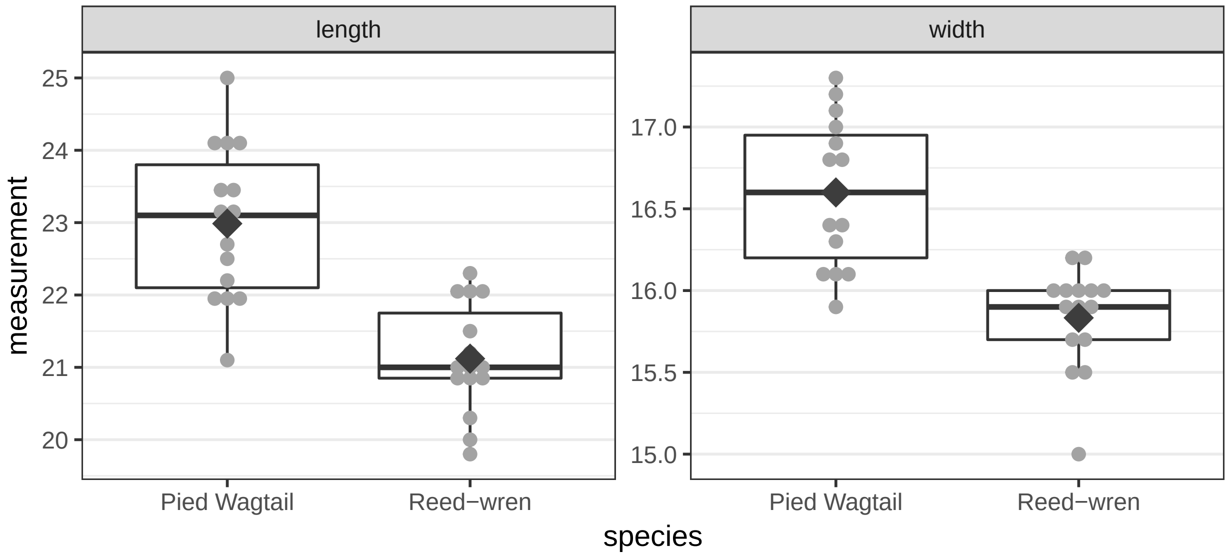


Although the result isn't actually black and white, as the grid lines and axis tick labels are still grey, you can see this changes the panel background to white.

Having grid lines along the y axis is useful for reading the graph, but grid lines along a categorical axis like the x axis seems unnecessary. Control of panel elements, including the removal of grid lines, can be done in `theme`. See the help page for `theme` via `?theme` for the many, many elements of the graphic that you can change.

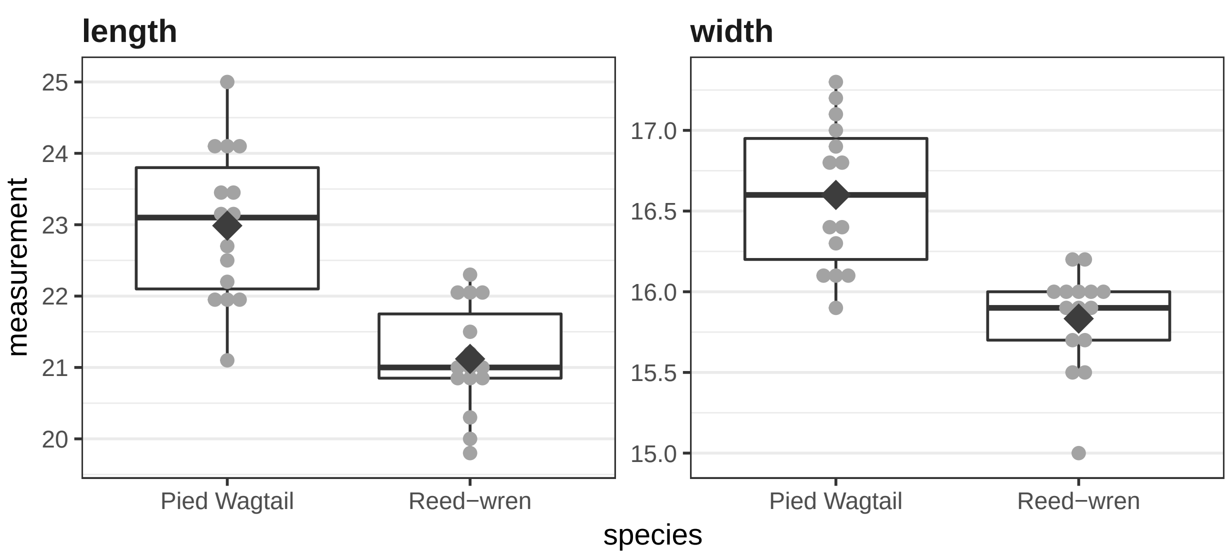
Because we only want to remove the grid lines on the x axis we will use the `panel.grid.major.x` argument set to `element_blank` within `theme`.

```r
g1 + theme(panel.grid.major.x = element_blank() )
```



While we're working in `theme`, let's make changes to the strips of the facets. We can change the color of the background (or remove it all together by setting it to `element_blank`) with `strip.background` and adjust the text placement and size with `strip.text`. Note we cannot actually change the labels in `strip.text`, only adjust how the text element is displayed.

```r
( g1 = g1 + theme(panel.grid.major.x = element_blank(),
                 strip.background = element_blank(),
                 strip.text = element_text(hjust = 0,
                                          face = "bold", size = 12) ) )
```



Changing facet labels

At this point I realized I'd made an error. Changing the text in the facet strips is something that is not particularly easy, although some work can be done via the `labeller` argument. I should have changed the levels of the variable `type` in the dataset to begin with to make them look nicer. Let's do that now, by assigning new `levels` to the `type` variable.

```r
# Changing the factor levels is easy
levels(eggs2$type)
```

```
[1] "length" "width"
```

```r
# We can change the names by assigning new levels
levels(eggs2$type) = c("Egg length", "Egg width")
levels(eggs2$type)
```

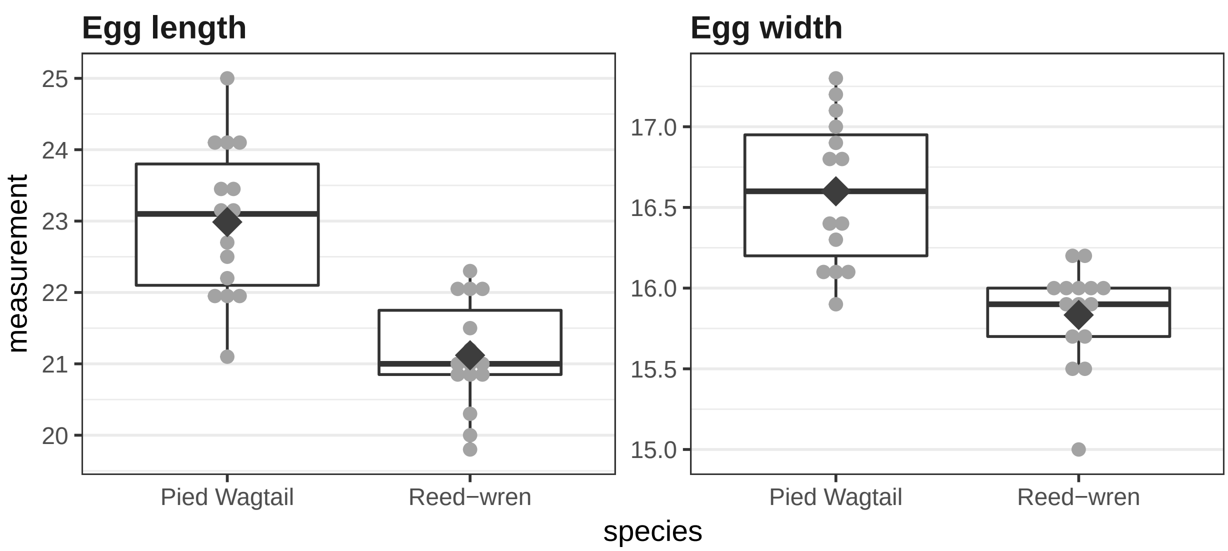
```
[1] "Egg length" "Egg width"
```

Creating the same plot using a different or edited dataset

Does that mean we have to rerun all the code we've done so far? We could - in a situation where we weren't going through things one step at a time, running the code again would be a piece of cake.

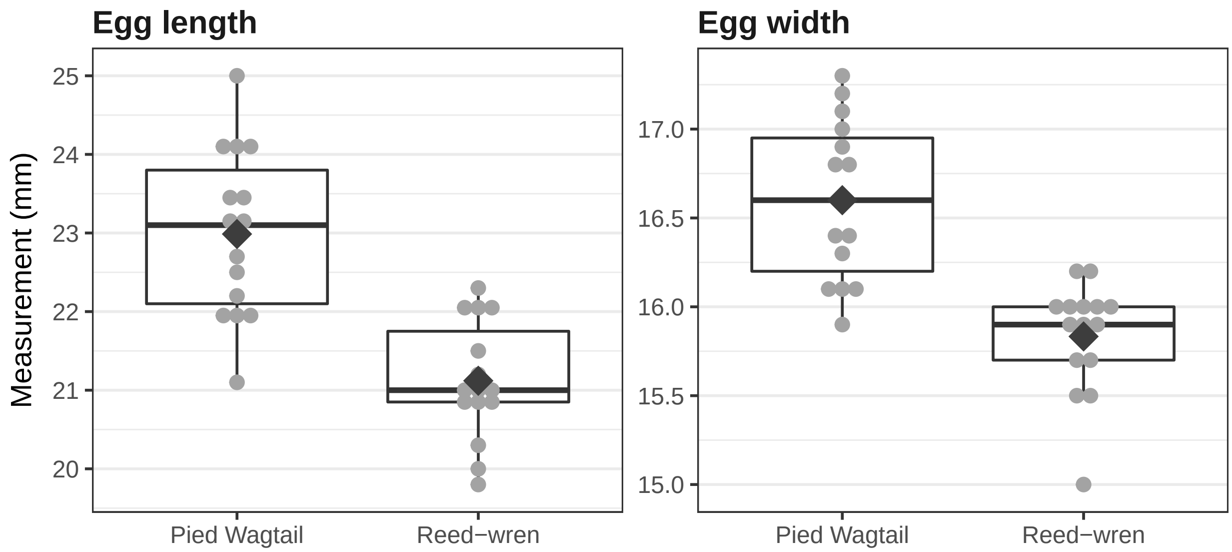
We do have another option, though, with the `%+%` function. This function allows us to recreate a graphic we've already made (and named) using a different dataset. This can be convenient if you want identical graphic appearance from data from separate datasets. Here we'll use the newly edited `eggs2` dataset with `g1` to recreate the graphic we've made so far.

```r
( g1 = g1 %+% eggs2 )
```



The last appearance change we really need is the axis labels. We can edit the axis labels easily using `ylab` and `xlab`. I decided to suppress the x axis label because it didn't seem necessary. By using `NULL` instead of `""` I removed all of the space between the tick labels and the bottom of the plot. The labels on the y axis just needed cleaning up, with capitalization and the units of measurement added.

```r
( g1 = g1 + labs(x = NULL,
                 y = "Measurement (mm)") )
```



Adding summary statistics as text

I could have stopped here and had a perfectly nice graphic. However, I like adding additional information to graphics so I decided to walk you through the process I went through to add summary statistics as text to this graphic.

To do this, we first need to calculate the summary statistics and figure out where to place them along the x and y axes. Then we'll add the information the graphic as *labels* with `geom_text`. You can see the final result in `Workshop final graphic 1.pdf`. The other way to do this, which I use more often, is to add tables to graphical objects using the **gridExtra** package. This package is also useful for putting two graphical objects in one pane.

We are going to add the mean and standard deviation of each `type` of egg measurement for each `species` as text in the graphic. First let's calculate the statistics we want for each group. I often use `group_by` and `summarise` from the **dplyr** package for this kind of work. We'll name the new summary data.frame `sumdat`. Notice that we need to round the results here to a reasonable number of significant digits.

```
library(dplyr)

# Don't forget to round to a reasonable number of digits
( sumdat = eggs2 %>%
        group_by(type, species) %>%
        summarise(Mean = round(mean(measurement), 1),
                  SD = round(sd(measurement), 1) ) )
```

```
# A tibble: 4 x 4
# Groups:   type [?]
        type      species  Mean    SD
      <fctr>       <fctr> <dbl> <dbl>
1 Egg length Pied Wagtail  23.0   1.1
2 Egg length    Reed-wren  21.1   0.8
3  Egg width Pied Wagtail  16.6   0.5
4  Egg width    Reed-wren  15.8   0.3
```

You can see in the final plot that I decided to add the summary statistics beneath each boxplot. This means we can still use `species` as the variable that defines where the text will be placed on the x axis, and `type` will define which facet it will be in. However, we'll need to calculate a new variable to define where the text will be positioned on the y axis.

After some trial and error (which you don't get to see), I decided the minimum y value from each facet minus 0.5 would be a good place to put the text in the plot. We can calculate this number separately for each `type` using `group_by/summarise` again. We'll name the y position variable `yloc`. We can then add `yloc` to `sumdat` using `inner_join` from `dplyr` to *join* or *merge* the two datasets.

```
( loc = eggs2 %>%
      group_by(type) %>%
      summarise(yloc = min(measurement) - .5) )
```

```
# A tibble: 2 x 2
        type  yloc
      <fctr> <dbl>
1 Egg length  19.3
2  Egg width  14.5
```

```
( sumdat = inner_join(sumdat, loc) )
```

```
Joining, by = "type"
```

```
# A tibble: 4 x 5
# Groups:   type [?]
        type      species  Mean    SD  yloc
      <fctr>       <fctr> <dbl> <dbl> <dbl>
1 Egg length Pied Wagtail  23.0   1.1  19.3
2 Egg length    Reed-wren  21.1   0.8  19.3
3  Egg width Pied Wagtail  16.6   0.5  14.5
4  Egg width    Reed-wren  15.8   0.3  14.5
```

Now all we have left to do is to create the text labels to add to the graphic. This involves combining all the appropriate text and values that we want to use as labels. Here we will do that using `paste0`. We need the labels we are making in our dataset, which we will achieve via `mutate` from `dplyr`. You can see in the final graphic that I put the means and standard deviations on separate lines. A line break in R is represented by `\n`, which we will include when making the labels.

If you don't know data manipulation in R very well, this may seem like a huge amount of work in order to get what you want. I am showing you this, though, because I wanted to drive the point home that working with a dataset outside of **ggplot2** is often required in order to take full advantage of the strengths of this package. If you aren't comfortable with this sort of work in R then **ggplot2** likely won't be a useful tool for you for making complicated final graphics. Another option is to add such information outside R using a different software package.

```
sumdat = mutate(sumdat, label = paste0("Mean=", Mean, " mm", "\n", "SD=", SD," mm") )

sumdat$label
```

```
[1] "Mean=23 mm\nSD=1.1 mm"   "Mean=21.1 mm\nSD=0.8 mm" "Mean=16.6 mm\nSD=0.5 mm"
[4] "Mean=15.8 mm\nSD=0.3 mm"
```

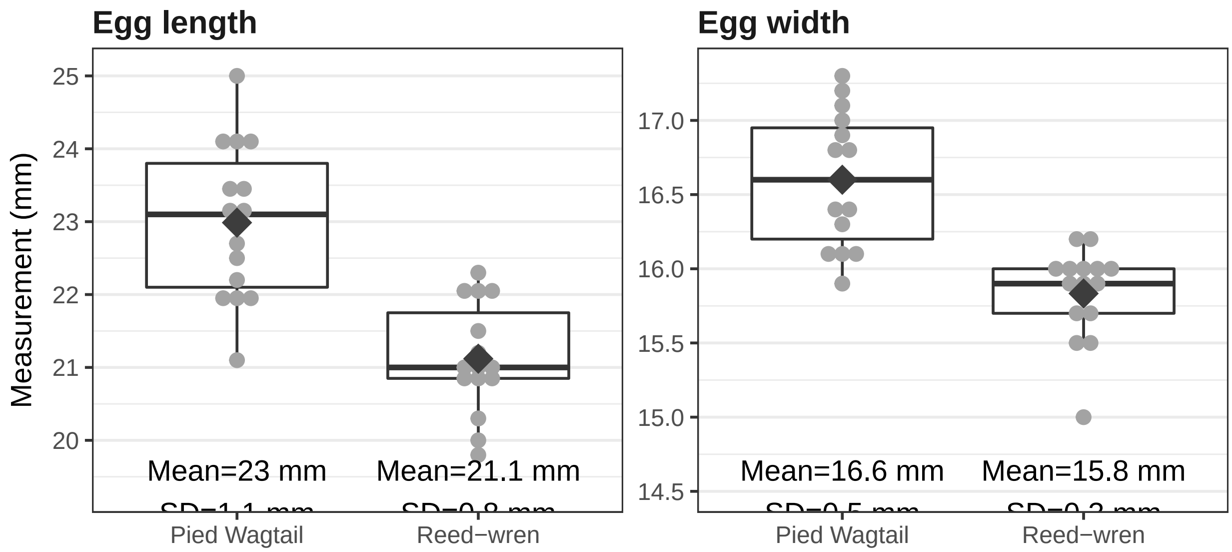
Defining a new dataset for plotting for a specific layer

We can now add text to a graphic using `geom_text`. For the first time we'll be defining a new dataset for plotting within a specific geom layer. Defining a new dataset in a geom will override the global dataset for the graphic that we defined in `ggplot`. The rest of the graphic will still plot information from the dataset `eggs2`, but `geom_text` will plot what is in the `sumdat` dataset. When defining a dataset within a layer it is safest to type out the `data` argument, as the dataset is not the first argument in the geom layers like it is in `ggplot`.

Notice we have to set the y position for `geom_text` because we're using a different variable for `y` than we had for the rest of the graphic. We map the y axis position to `yloc` within `aes`.

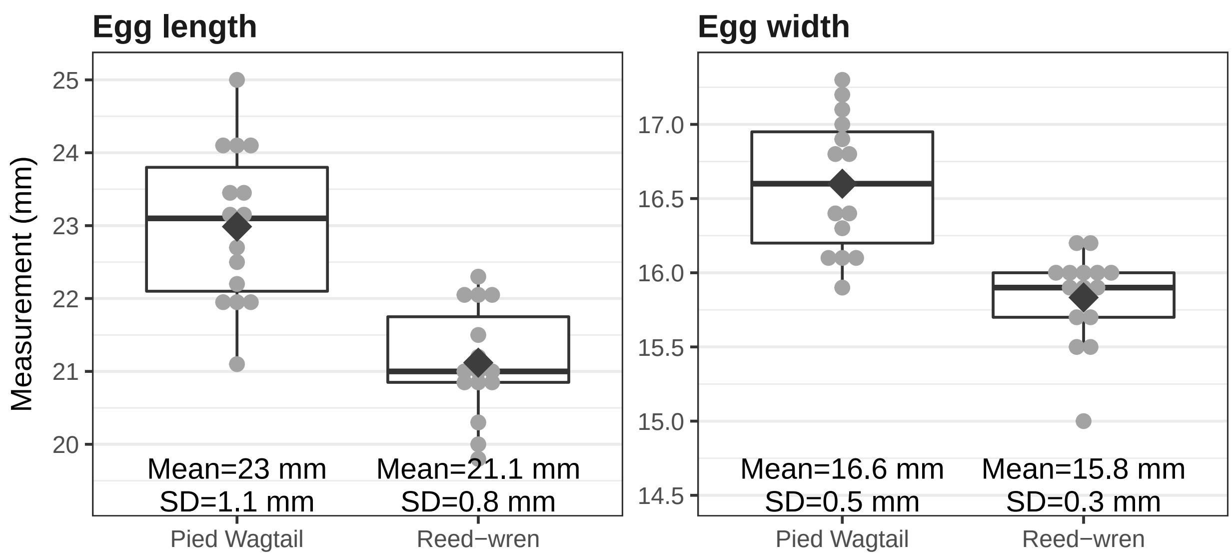
The `geom_text` geom requires us to define a `label` variable within `aes`. The help files for individual geoms will tell you if there are required aesthetics, which you should always check if you are using a geom for the first time. We will map the variable `label` to the `label` aesthetic.

```
g1 + geom_text(data = sumdat, aes(label = label, y = yloc) )
```



As often happens, we need to do some adjustments to the text we've added to make the graphic look nicer. Here we'll adjust the line height, change the vertical justification, and change the size of the text within `geom_text`.

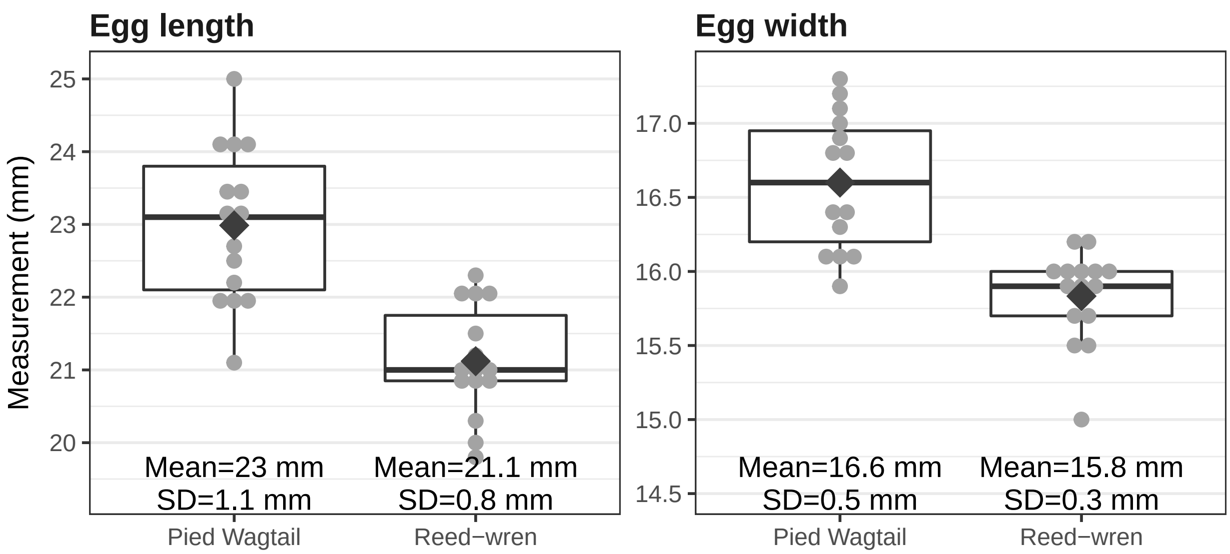
```
( g1 = g1 + geom_text(data = sumdat, aes(label = label, y = yloc),
                      lineheight = .9, vjust = .3, size = 4) )
```



That's the final graphic that I showed you. This is what the code looks like when it is put all together instead of adding layers line by line.

```
( g1 = ggplot(eggs2, aes(x = species, y = measurement) ) +
    geom_boxplot() +
    facet_wrap(~type, scales = "free_y") +
    geom_dotplot(binaxis = "y", stackdir = "center",
                 color = "grey64", fill = "grey64") +
    stat_summary(fun.y = mean, geom = "point", shape = 18,
                 size = 5, color = "grey24") +
    theme_bw() +
    theme(panel.grid.major.x = element_blank(),
          strip.background = element_blank(),
          strip.text = element_text(hjust = 0, face = "bold", size = 12) ) +
    labs(x = NULL,
         y = "Measurement (mm)") +
```

```
    geom_text(data = sumdat, aes(label = label, y = yloc),
              lineheight = .9, vjust = .3, size = 4) )
```



In making this graph, I've relied on the graphic displayed in the plotting window to help decide on spacing and point sizes and such. Be careful using this method, because we haven't actually set the plotting window size to anything. In fact, you can't set the RStudio plotting window to a specific size at this time. Depending on the size and shape we save the graphic as, we may decide we need to make a few more adjustments. One way to check how things will look at different sizes is to preview the plot at difference sizes using the `Export` drop down menu in the RStudio **Plots** pane. You can also just save the graphic to your desired size and take a look at it.

Saving a plot with `ggsave`

We'll save the plots using `ggsave` from **ggplot2**. You can save graphics in all kinds of formats, but here we'll save the graphic as a PDF and as a PNG file. The sizes I chose are arbitrary, and are just to demonstrate some of the `ggsave` options.

By default `ggsave` saves the last plot made to whatever size your plot window is. You can set the graphic size via `width` and `height`.

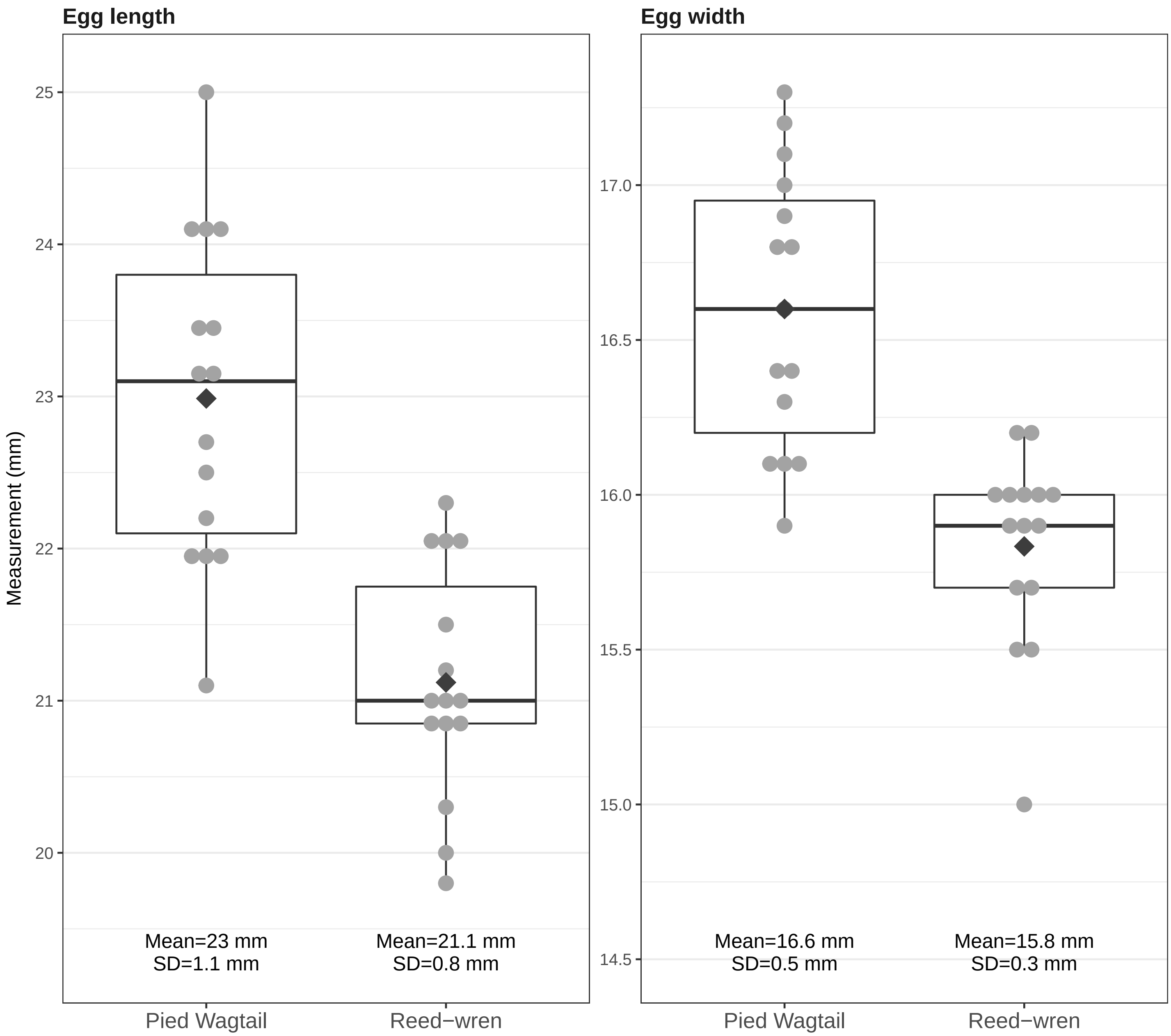
It can be safer to define the graphic you want saved by name using the `plot` argument. If you want the plot to look like it does in the plotting window, simply leave `height` and `width` at their default values.

```
ggsave("final plot 1.pdf", width = 7, height = 7) # setting width and height (default inches)
ggsave("final plot 1.png", plot = g1) # using default size based on plotting window
```

At this point I decided that the dot size was too big and the tick text on the x axis was too small. I edited the `geom_dotplot` argument `dotsize` and the `theme` argument `axis.text.x` and then re-saved the final plot at a final size, which was 9 inches wide and 8 inches high. The graphic at this size is shown below.

```
(g1 = ggplot(eggs2, aes(x = species, y = measurement) ) +
    geom_boxplot() +
    facet_wrap(~type, scales = "free_y") +
    geom_dotplot(binaxis = "y", stackdir = "center",
                 color = "grey64", fill = "grey64", dotsize = .5) +
    stat_summary(fun.y = mean, geom = "point", shape = 18,
                 size = 5, color = "grey24") +
    theme_bw() +
    theme(panel.grid.major.x = element_blank(),
          strip.background = element_blank(),
          strip.text = element_text(hjust = 0, face = "bold", size = 12),
          axis.text.x = element_text(size = 12) ) +
```

```
    labs(x = NULL,
         y = "Measurement (mm)") +
    geom_text(data = sumdat, aes(label = label, y = yloc),
              lineheight = .9, vjust = .3, size = 4) )
```



Notice we can change `dpi` (dots per inch) for the PNG format

```
ggsave("final plot 1.pdf", plot = g1, width = 9, height = 8)
ggsave("final plot 1.png", dpi = 600, width = 9, height = 8)
```

Polished graphic #2: A "results" plot

The last graphic we are going to make is one that displays the results from an analysis of a dataset that had two factors. The results we will be graphing are the tests of differences in mean growth of each group against the control group. I pulled the estimated differences in means and the associated confidence intervals from the statistical model and saved the results as `all vs control results.csv`. Let's load this dataset now and take a look at it.

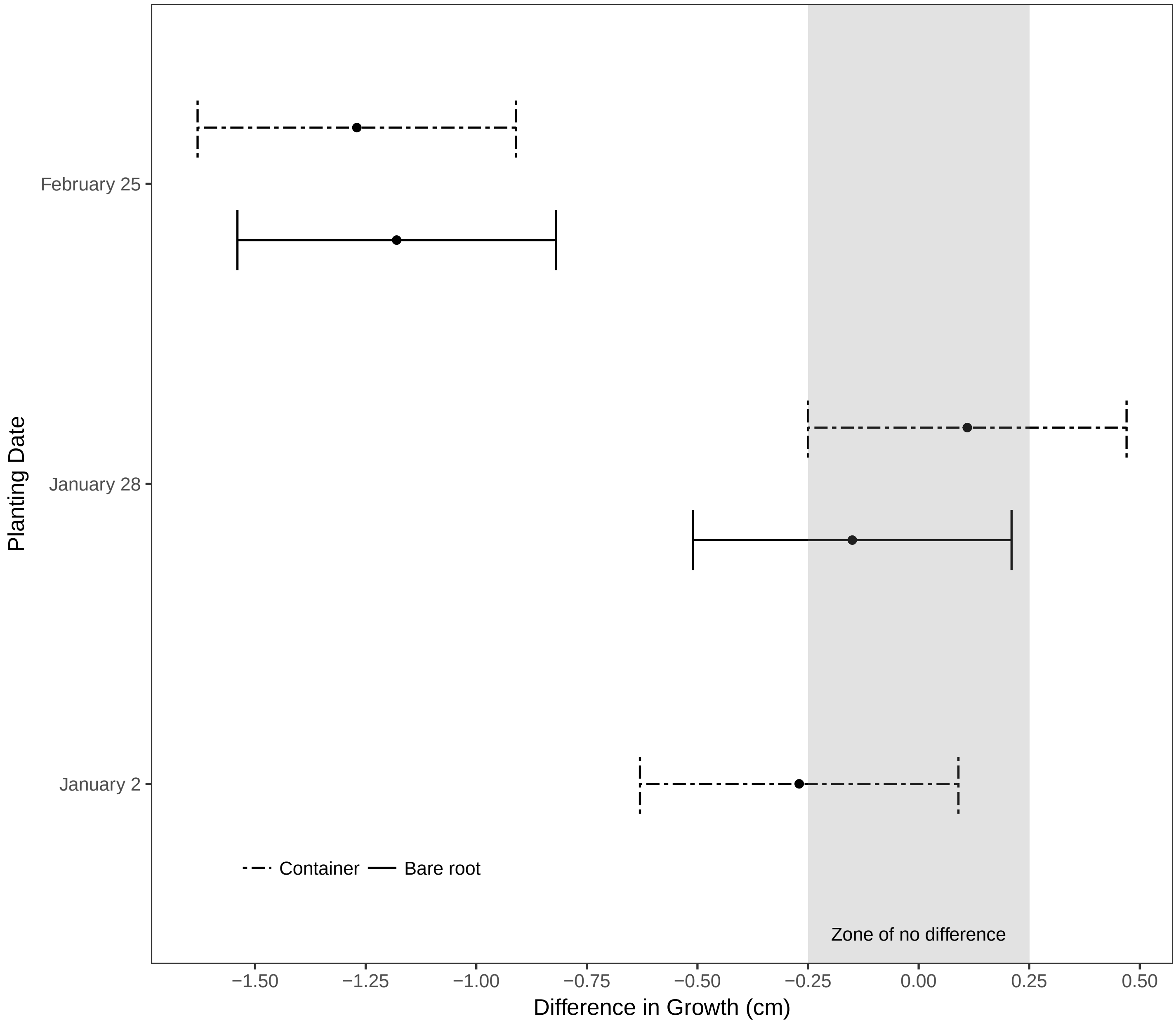
```r
res = read.csv("all vs control results.csv")
str(res)
```

```
'data.frame':   5 obs. of  5 variables:
 $ Diffmeans: num  -0.27 0.11 -0.15 -1.27 -1.18
 $ Lower.CI : num  -0.63 -0.25 -0.51 -1.63 -1.54
 $ Upper.CI : num  0.09 0.47 0.21 -0.91 -0.82
 $ plantdate: Factor w/ 3 levels "Feb25","Jan2",..: 2 3 3 1 1
 $ stocktype: Factor w/ 2 levels "bare","cont": 2 2 1 2 1
```

```r
res
```

```
  Diffmeans Lower.CI Upper.CI plantdate stocktype
1     -0.27    -0.63     0.09      Jan2      cont
2      0.11    -0.25     0.47     Jan28      cont
3     -0.15    -0.51     0.21     Jan28      bare
4     -1.27    -1.63    -0.91     Feb25      cont
5     -1.18    -1.54    -0.82     Feb25      bare
```

I added the two factor variables to the results dataset before I saved it so I would be able to tell which groups were being compared to the control on each row. The two factors are `plantdate`, which has three levels that indicate the date that trees were planted, and `stocktype`, which has two levels and indicates the kind of stock (either bare root or in a container). The control group was bare root trees planted on January 2. There were six groups including the control group, and so there are five comparisons in this table of results. I provided you the graphic we are working towards as `Workshop final plot 2.pdf`, which is shown below.



Setting the factor order to control axis order

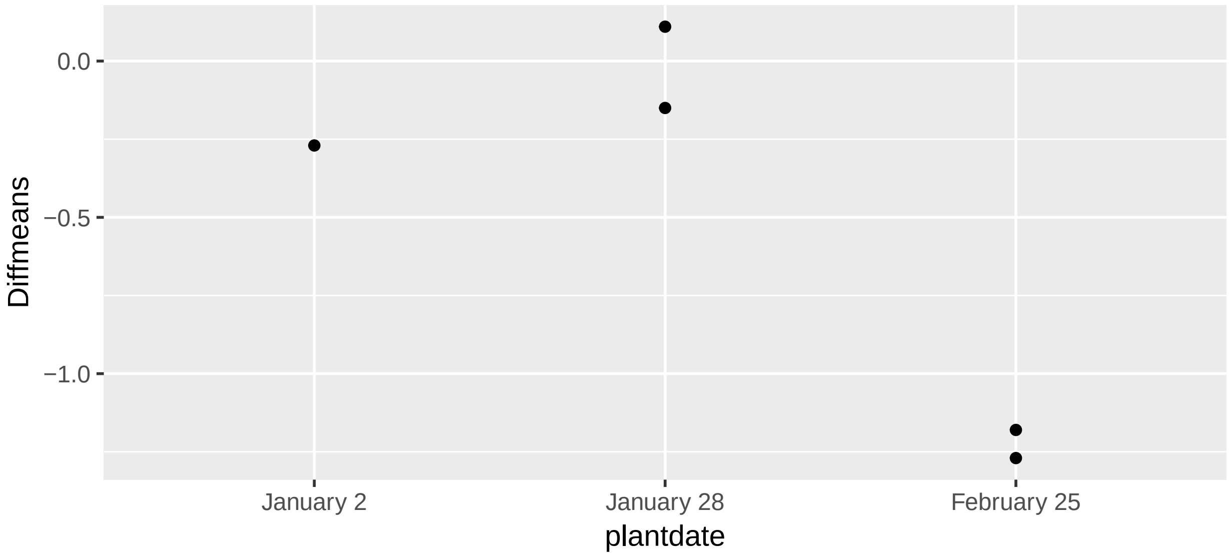
We now know that it is often easiest to clean up the dataset before we are ready to begin plotting it, so let's make some changes to the variable `plantdate`. The levels of this variable aren't in date order because R by default orders levels of factors alphanumerically. We can always change the display order of a factor in **ggplot2** by changing the order of the factor levels in the dataset. We will relabel the levels of the factor to make them look nicer while we're at it to save us some work later.

```
res$plantdate = factor(res$plantdate,
                   levels = c("Jan2", "Jan28", "Feb25"),
                   labels = c("January 2", "January 28", "February 25") )
```

This example turns out to be more complex than I was originally planning on because of the odd number of comparisons. Two of the plant dates have comparisons between both stock types and the control but January 2 only has one stock type comparison. However, I decided to stick with the complex graphic so you can see the kinds of problems you have might run into as your graphic complexity increases.

We'll start by plotting the estimated differences in means as points with `plantdate` on the x axis. We'll name this graphic `g2`, and then build on it layer by layer.

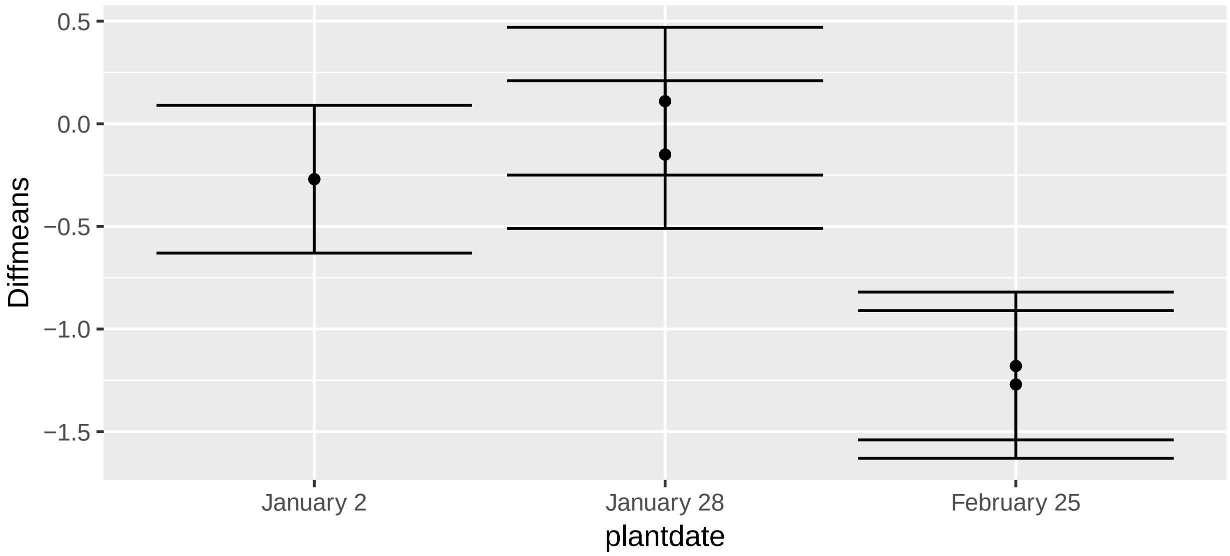
```r
( g2 = ggplot(res, aes(x = plantdate, y = Diffmeans) ) +
      geom_point() )
```



Adding error bars to a plot

We can add the 95% confidence intervals around the estimated means as error bars with `geom_errorbar`. We have to map the aesthetics `ymin` and `ymax` to variables that represent the *ends* of the error bars. In fact, `ymin` and `ymax` are required aesthetics in `geom_errorbar`. We have the upper and lower confidence limits in our dataset for this purpose.

```r
( g2 = ggplot(res, aes(x = plantdate, y = Diffmeans) ) +
      geom_point() +
      geom_errorbar( aes(ymin = Lower.CI, ymax = Upper.CI) ) )
```

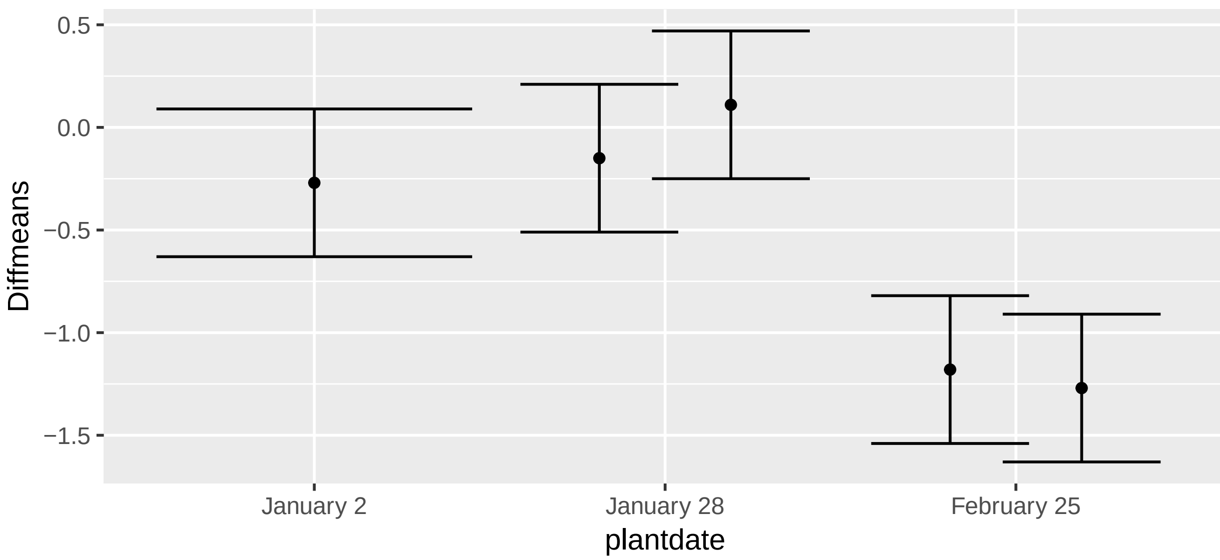


Dodging to avoid overlap

When we have two different stock types at a single planting date, the error bars overlap and the graphic is difficult to understand. To avoid this, we will do what is called *dodging*. In order to *dodge* the overlapping points, we need to define which variable tells us that we have two values at a single planting date. In this case that variable is `stocktype`, so we map the `group` argument in `aes` to `stocktype`.

We want to dodge whenever we get an overlap along the x axis, so we'll choose the amount we want to dodge by using the `width` argument in `position_dodge`. Notice we have to dodge both the points and the error bars, and that we do it by the same `width` amount. Common dodging amounts are 0.9 and 0.75.

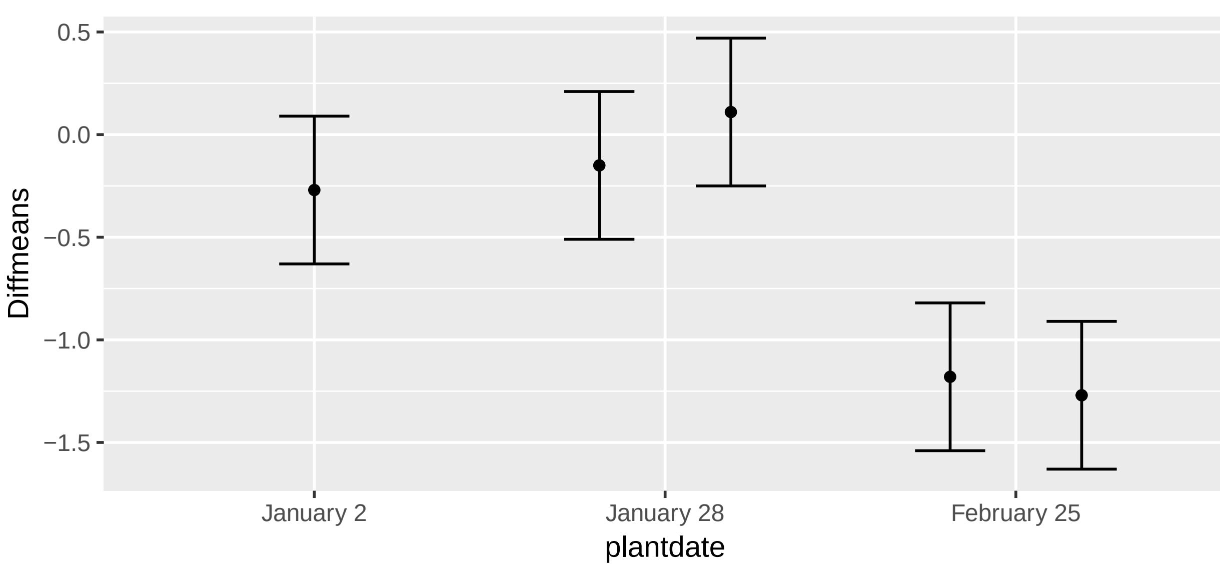
```
( g2 = ggplot(res, aes(x = plantdate, y = Diffmeans, group = stocktype) ) +
      geom_point(position = position_dodge(width = .75) ) +
      geom_errorbar( aes(ymin = Lower.CI, ymax = Upper.CI),
                     position = position_dodge(width = .75) ) )
```



Setting the `width` of error bars for unbalanced factors

Now we have gotten rid of the overlap problem. But this is where things start getting a bit complicated. The width of the error bar is twice as big for the single stock type on January 2 as it is when we have two stock types on the other dates. We can control this using the `width` argument in `geom_errorbar`. Usually we set `width` to a constant and all the widths would be changed the same way. Here we'll need to set the `width` of the first error bar to *half* the value of the others. This means we'll have to set `width` to a vector defining the width of all five error bars (so the vector has a length of 5).

```
( g2 = ggplot(res, aes(x = plantdate, y = Diffmeans, group = stocktype) ) +
      geom_point(position = position_dodge(width = .75) ) +
      geom_errorbar( aes(ymin = Lower.CI, ymax = Upper.CI),
                     position = position_dodge(width = .75),
                     width = c(.2, .4, .4, .4, .4) ) )
```



That looks much nicer. But look what happens when we try to map the `linetype` aesthetic to `stocktype` in the error bar layer.

- Note that this has appeared to have been fixed in the development version of ggplot2, so if you are using versions of ggplot2 later than 2.2.1 you may be able to do this without issues and without the fix in the next step.*

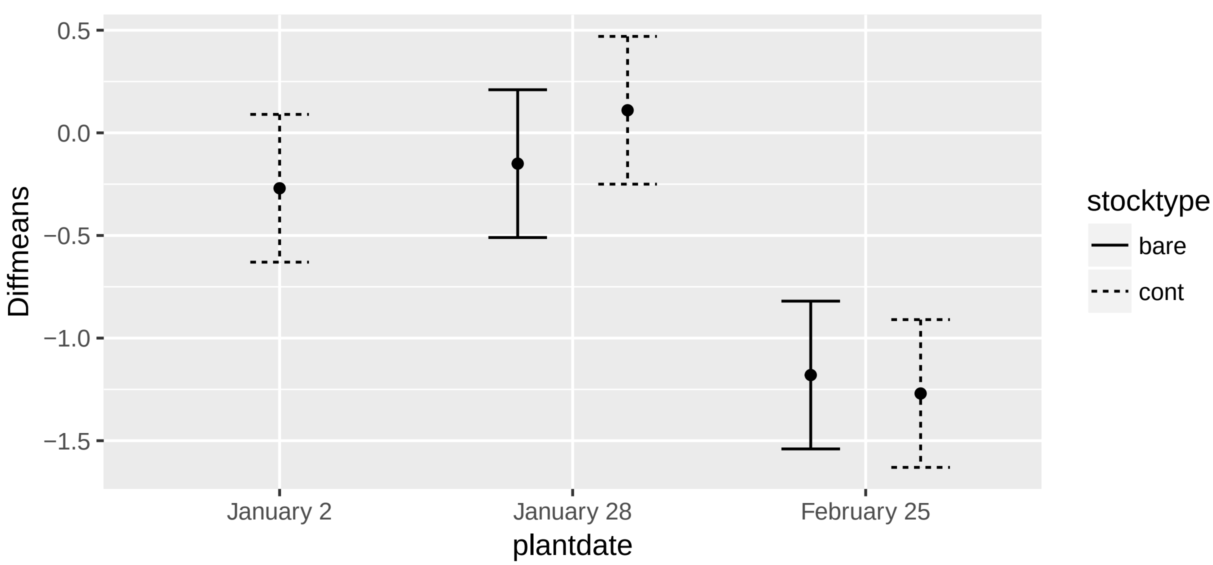
```
( g2 = ggplot(res, aes(x = plantdate, y = Diffmeans, group = stocktype) ) +
      geom_point(position = position_dodge(width = .75) ) +
      geom_errorbar( aes(ymin = Lower.CI, ymax = Upper.CI,
                         linetype = stocktype),
                     position = position_dodge(width = .75),
                     width = c(.2, .4, .4, .4, .4) ) )
```

```
Error: Aesthetics must be either length 1 or the same as the data (2): width
```

This is our first error message of the day. You can often tell the difference between error messages and other messages by whether or not the plot prints to the plotting window. In this case, no plot is printed in the plotting window and so something needs to be changed in the code before we get the plot we want.

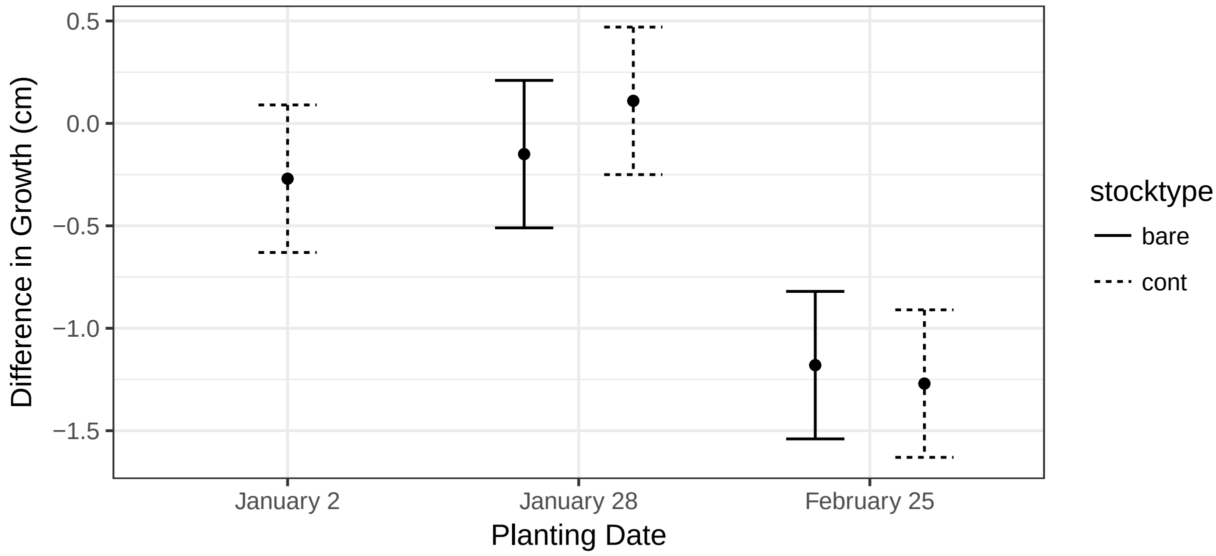
The error message isn't entirely clear, but it does give us some hints about where the problem may be. It appears that something is wrong with the `width` aesthetic compared to the other aesthetics. I found that I had to move `width` inside `aes` to get `linetype` and `width` to play nicely together. It is unusual, but it solved the problem and things went more smoothly once that was done. If we'd had even numbers of observations per planting date we wouldn't have had this problem.

```
( g2 = ggplot(res, aes(x = plantdate, y = Diffmeans, group = stocktype) ) +
      geom_point(position = position_dodge(width = .75) ) +
      geom_errorbar( aes(ymin = Lower.CI, ymax = Upper.CI,
                         linetype = stocktype,
                         width = c(.2, .4, .4, .4, .4) ),
                     position = position_dodge(width = .75) ) )
```



Now that we have the basic graphic built, let's start polishing it up by setting the theme to `theme_bw` as well as changing the axis labels.

```
( g2 = g2 + theme_bw() +
      labs(x = "Planting Date",
           y = "Difference in Growth (cm)") )
```

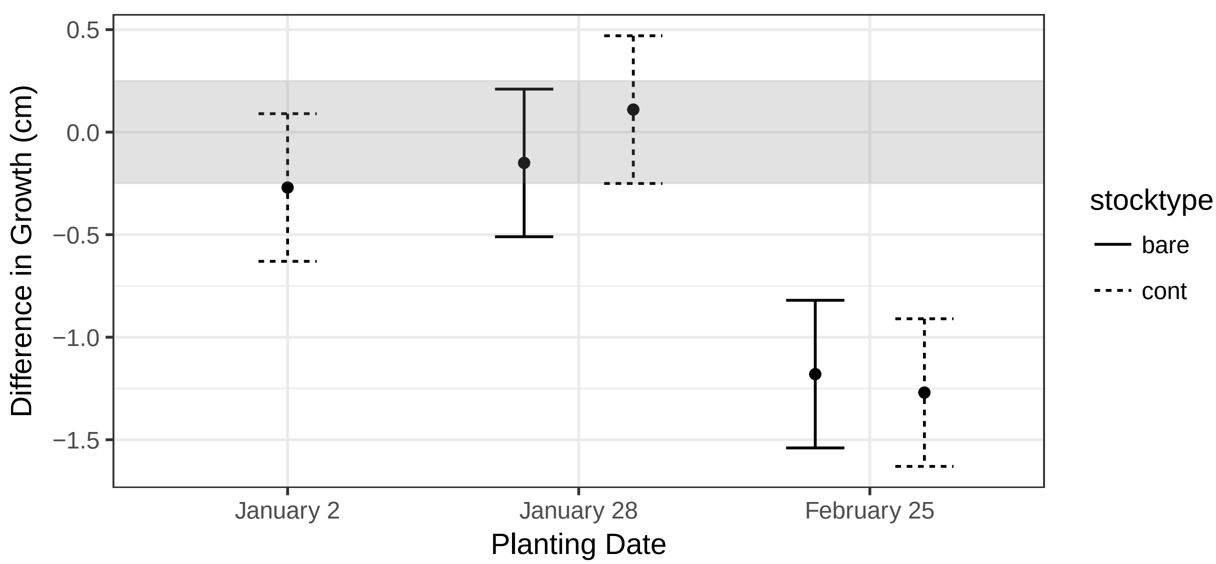


Adding shaded rectangles to an area of the plot

It can be informative to give some indication of what a practical difference would be on the graphic. In this case, a difference in mean growth of 0.25 cm more or less compared to the control would be practically meaningful to growers. In the past I've indicated the practically important limits using horizontal lines with `geom_hline`. Today we'll add a shaded rectangle to indicate this "Zone of no difference" using `geom_rect`.

There are some required aesthetics in `geom_rect` that indicate where the *edges* of the rectangle should be drawn. We are setting these to constants, so these required aesthetics will be set outside of `aes`. We'll set `alpha` to make the rectangle see-through and set the fill color to some gray shade with `fill`. Because we want the rectangle to go all the way across the x axis we use `Inf`/`-Inf`.

```
( g2 = g2 + geom_rect(xmin = -Inf, xmax = Inf, ymin = -.25, ymax = .25,
                      fill = "grey54", alpha = .05) )
```

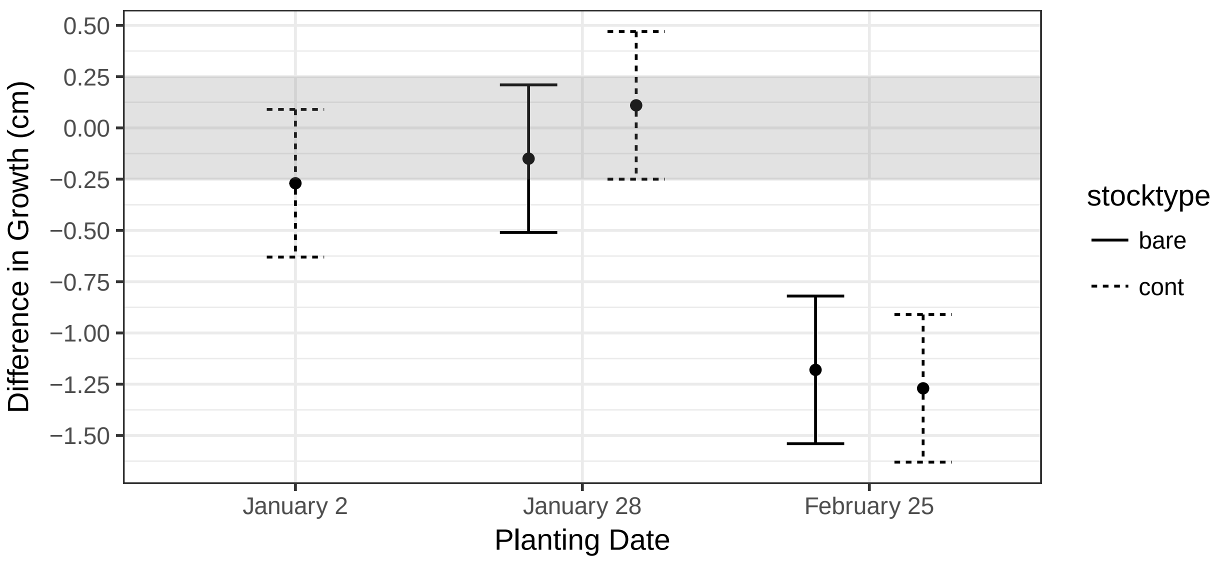


Changing the axis breaks

If 0.25 cm is the practically important difference, we may want to show finer tick marks or *breaks* on the y axis. Notice that right now the breaks are set for every 0.5 cm change. We can control axes with the appropriate scale functions. In this case, the y axis is continuous and we'll use `scale_y_continuous`. If we wanted to control the x axis, which has a discrete scale in this plot, we would use `scale_x_discrete`.

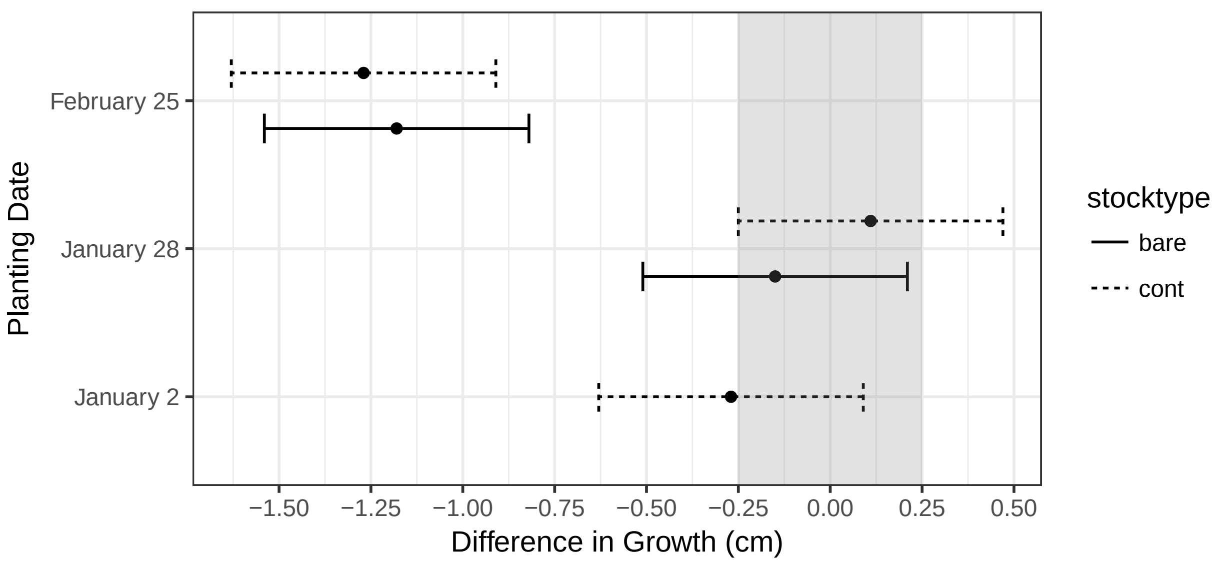
We'll make axis breaks every 0.25 cm instead of 0.5 cm using a sequence of numbers from the -1.5 to 0.5 by .25 (see `?seq` for more info).

```
( g2 = g2 + scale_y_continuous(breaks = seq(-1.5, .5, by = .25) ) )
```



The axes of the current graphic are reversed compared to the final graphic that I showed you when we started this exercise. We could have built this entire graphic from the start by defining the axes differently and using `geom_errorbarh` to create horizontal error bars instead of the vertical ones we get with `geom_errorbar`. However, there are few horizontal geoms available in **ggplot2** (but see package **ggstance**), so I wanted to show you how `coord_flip` works for making horizontal plots. It simply flips the axes.

```
( g2 = g2 + coord_flip() )
```

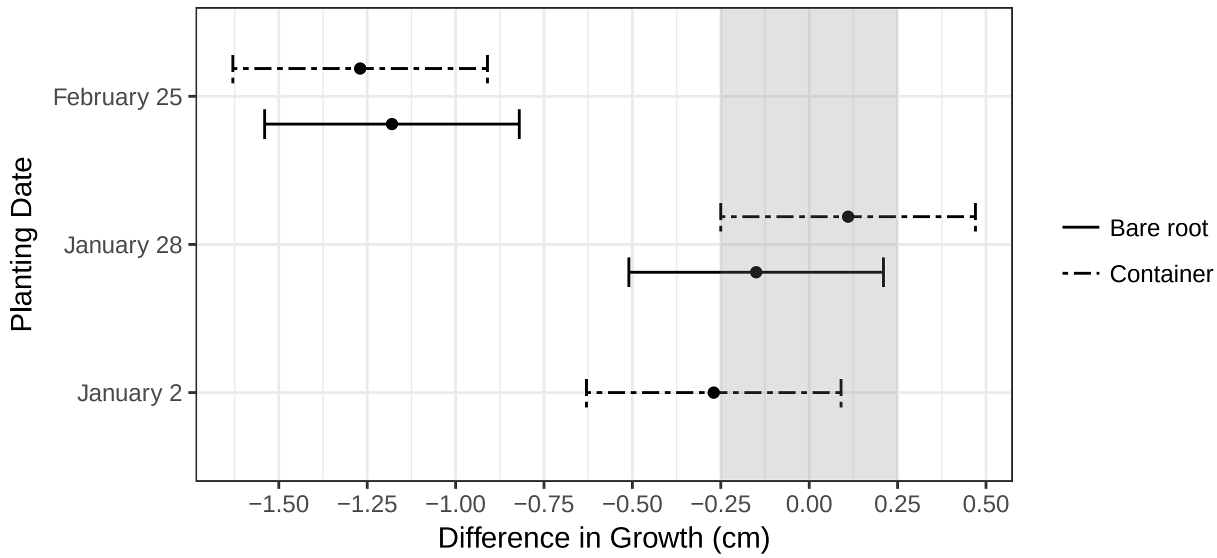


Changing the appearance of the legend

There are a few final appearance changes we need to make on this graphic. First, I didn't like the dotted `linetype` that we got by default. We can change the type of lines we get with the function `scale_linetype_manual`. Most aesthetics have a scale function that you can use to change the default aesthetic settings (we saw `color` and `fill` scales earlier in the workshop).

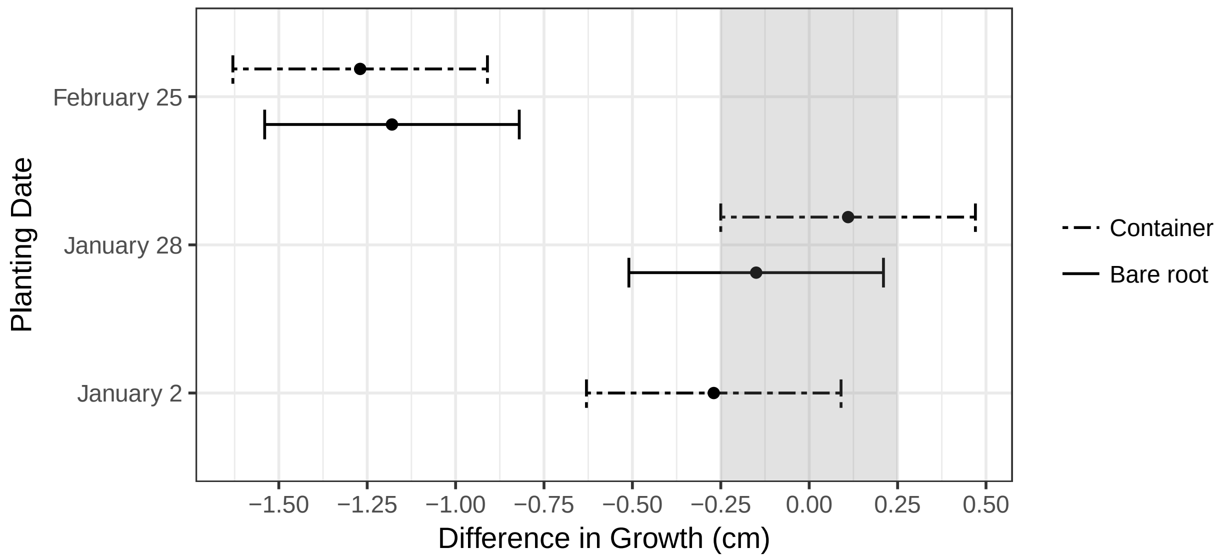
We'll set the `values` of the line types in `scale_linetype_manual`. We'll also change the title and the labels of the groups that are displayed in the legend with this function, as well, via `name` and `labels`, respectively.

```r
g2 + scale_linetype_manual(values = c("solid", "twodash"),
                           name = element_blank(),
                           labels = c("Bare root", "Container") )
```



It would be more aesthetically pleasing to have the order that the lines show in the legend match the order that the lines show up in the graphic - in this case the dashed line/`Container` group comes first in the plot. We can control this inside `scale_linetype_manual` as well, using the `guide` argument with `guide_legend`. We set `reverse` to `TRUE` to reverse the order of the legend. The `guide_legend` function allows us to control many aspects of the legend.

```r
( g2 = g2 + scale_linetype_manual(values = c("solid", "twodash"),
                                  name = element_blank(),
                                  labels = c("Bare root", "Container"),
                                  guide = guide_legend(reverse = TRUE) ) )
```



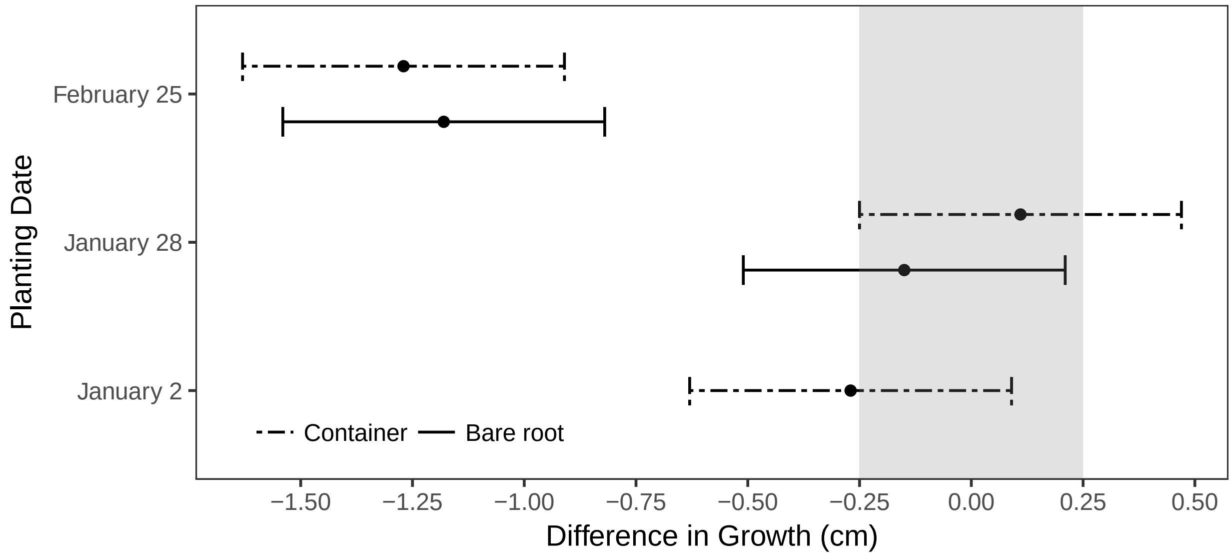
Moving the legend inside the plot panel

We have plenty of blank space within the graphic to fit the legend. Let's move the legend inside the plot panel and switch it to horizontal from vertical. This can be done in `theme`. I decided to remove all the grid lines, so we'll do that in `theme` while we're at it. Keeping the x axis grid lines would have been a reasonable, as well.

We use `legend.position` to move the legend inside the plot panel and `legend.direction` to switch it to a horizontal format. The legend position here is based on coordinates between 0 and 1 on each axis, and took trial and error to get a nice placement. You can also use options such as, e.g., "bottom" in legend.position to leave the legend outside the plot panel but move it to a different side of the plot. See the help page for theme for more options.

On a side note, the amount of blank space in the panel of this graphic means it would be a good candidate for adding a table of grouped summary statistics or a small plot displaying the raw data. Both of those things could be achieved fairly easily by combining **ggplot2** with **gridExtra**.

```
( g2 = g2 + theme(legend.position = c(.2, .1),
                  legend.direction = "horizontal",
                  panel.grid = element_blank() ) )
```

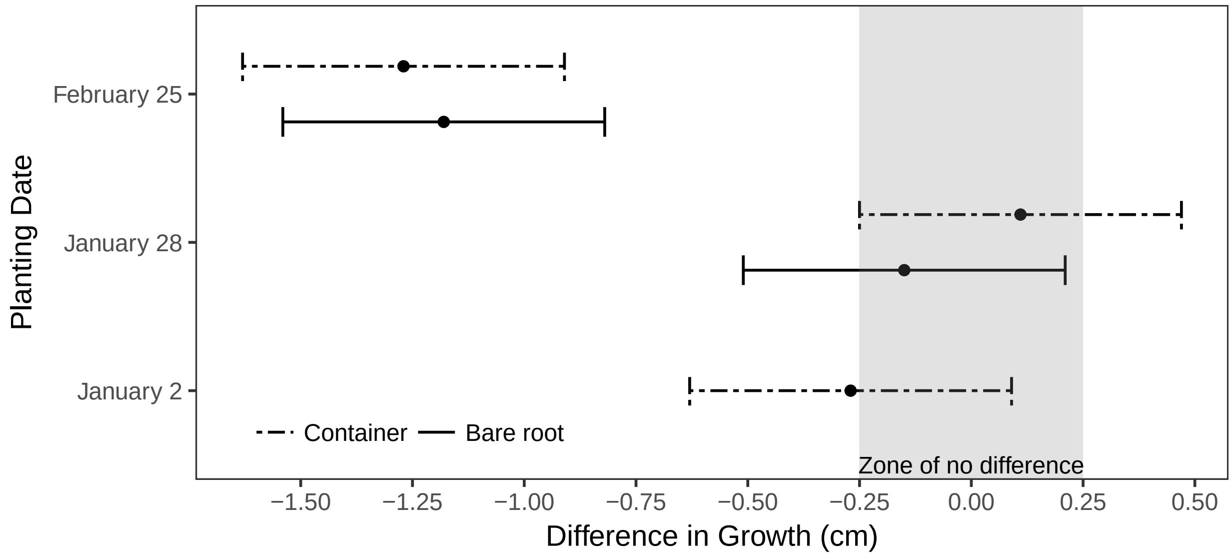


Adding a label with annotate

The last thing we want to do is add the label to our "Zone of no difference" rectangle. Unlike adding multiple labels all positioned separately by group like in the last graphic, we want to add a single label. This is a clue that we should add the text using annotate instead of geom_text. It would have made sense to add the rectangle via annotate, as well.

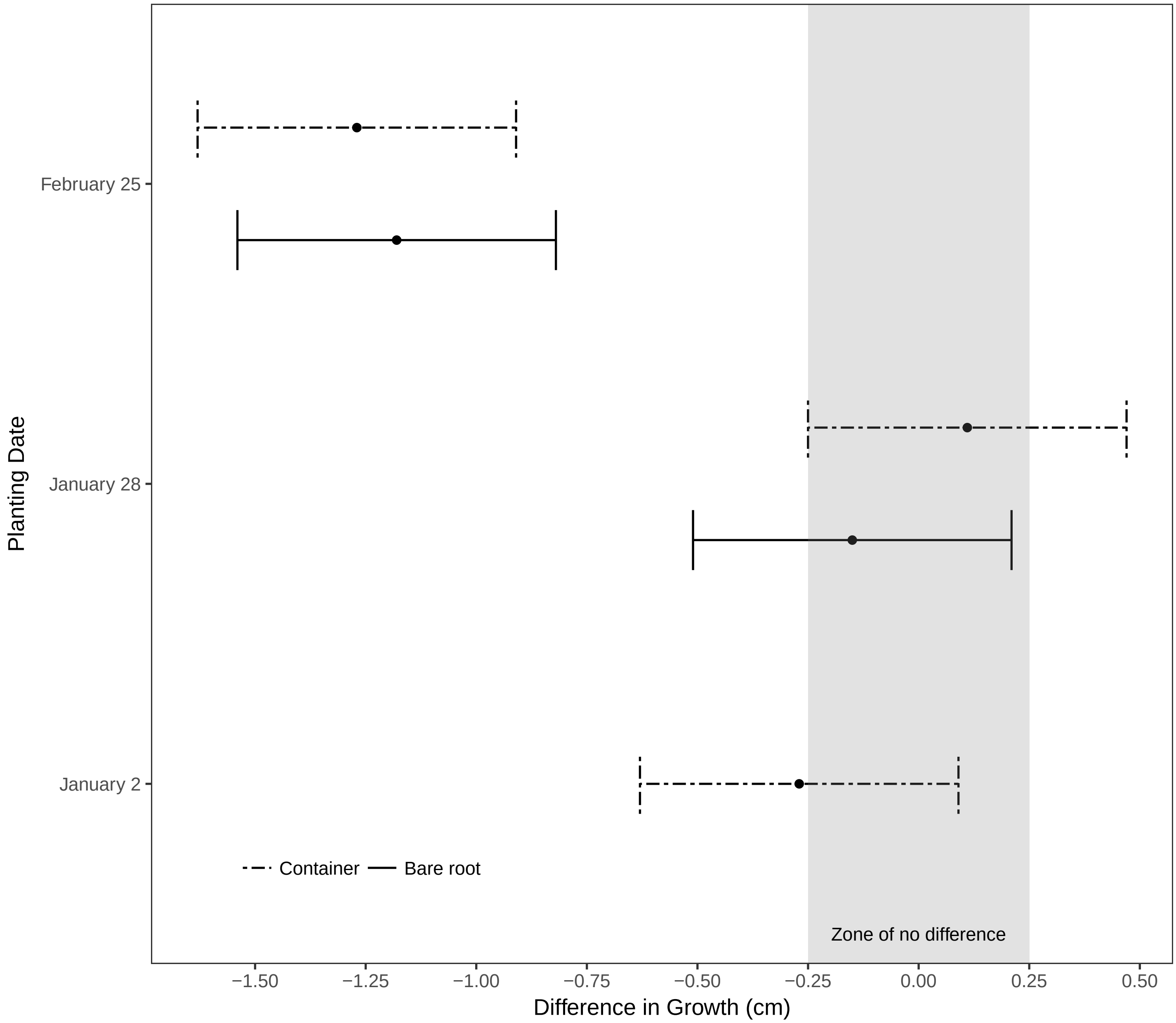
The only complicated thing here is to remember that we flipped the plot axes. This means that while the y axis now shows up on the x axis, we still need to refer to it as the y axis when placing our label via annotate. In addition to placing the label we'll also shrink the size of the text to make it fit nicely inside the rectangle. Because we are setting all aesthetics, we do it outside aes. Notice we define which geom we want to annotate with in annotate; in this case, we use "text".

```
( g2 = g2 + annotate(geom = "text", x = .5, y = 0, label = "Zone of no difference", size = 3) )
```



And there we have it, the final graphic. Here's what the code looks like all together. The graphic is displayed at the size of the final graphic I showed you before we began working on this graphic.

```
( g2 = ggplot(res, aes(x = plantdate, y = Diffmeans, group = stocktype) ) +
      geom_point(position = position_dodge(width = .75) ) +
      geom_errorbar( aes(ymax = Upper.CI, ymin = Lower.CI,
                        linetype = stocktype,
                        width = c(.2, .4, .4, .4, .4) ),
                    position = position_dodge(width = .75) ) +
      theme_bw() +
      labs(x = "Planting Date",
             y = "Difference in Growth (cm)") +
      geom_rect(xmin = -Inf, xmax = Inf, ymin = -.25, ymax = .25,
                fill = "grey54", alpha = .05) +
      scale_y_continuous(breaks = seq(-1.5, .5, by = .25)) +
      coord_flip() +
      scale_linetype_manual(values = c("solid", "twodash"),
                           name = element_blank(),
                           labels = c("Bare root", "Container"),
                           guide = guide_legend(reverse = TRUE) ) +
      theme(legend.position = c(.2, .1),
            legend.direction = "horizontal",
            panel.grid = element_blank()) +
      annotate(geom = "text", x = .5, y = 0,
               label = "Zone of no difference", size = 3) )
```



You would save this graphic in the same way as before with `ggsave`.

```
ggsave("final plot 2.pdf", width = 8, height = 7)
```

That is all the material we will cover today. There is much more that you can do in **ggplot2** that we didn't have time to review, but this should give you a start on making your own graphics with **ggplot2** in R.