Lab 05, Fall 2020

Your Friendly GEs!

November 01, 2020

Contents

Note: This material on loops is an excerpt from Grant McDermott's Data science for economist course.

Lesson 0: Graphing with ggplot2

Before we get into ggplot, let's talk about `plot()` which is R's base plotting package. It does a fine job of plotting things, and is quite powerful, but ggplot is more user friendly and is definitely a resume booster if you can use it well. Many companies, researchers and journalists use ggplot to produce graphs for wide audiences. The best part? It's totally free to use.

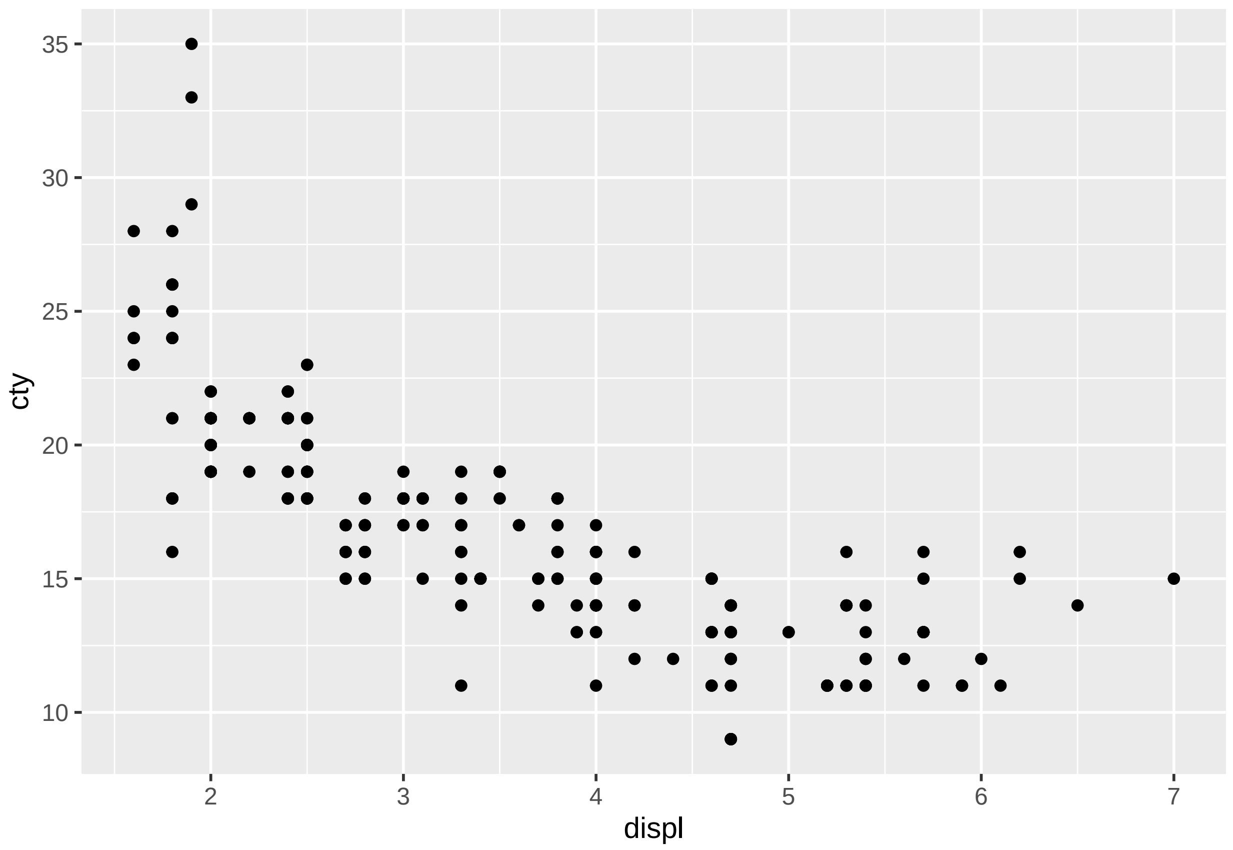
The fastest way to make a simple plot in R using `plot()` is by using the function qplot which stands for "quick plot"

```r
library(pacman)
p_load(tidyverse, magrittr, ggplot2, ggthemes)
p_load(ISLR)
#if p_load doesn't work, use below commands.
```

```r
#install.packages("ISLR",repos = "http://cran.us.r-project.org")
#library(ISLR)
names(Auto)
```

```
## [1] "mpg"          "cylinders"    "displacement" "horsepower"   "weight"
## [6] "acceleration" "year"         "origin"       "name"
```

```r
qplot(x = displ, y = cty, data = mpg)
```



but we can do better than that with ggplot2! But first we need to understand how to communicate with ggplot so it can do what we want it to.

I will introduce you to the basics. The *gg* in ggplot refers to the **g**rammar of **g**raphics. This is Hadley Wickham's adoption of Leland Wilkinsons brainchild. There are three principles

- **Aesthetic mappings** -This is the map through which your data is linked to the plot
- **Geoms** -Tells us how the mappings are defined. These could be points, lines, bars, etc
- **Layers** -This is how we construct our plots: adding different layers of the plot together.

This is sort of hard to get your head around when you first start thinking about it. Some examples will help. See here for an excellent resource

So: the general idea here is that we build a linking of data through ggplot(), then add layers with the + operator, where these layers can be geometries that could be bar graphs, histograms, scatterplots, whatever you want. They could also be graph themes, new functions, even animation directions.

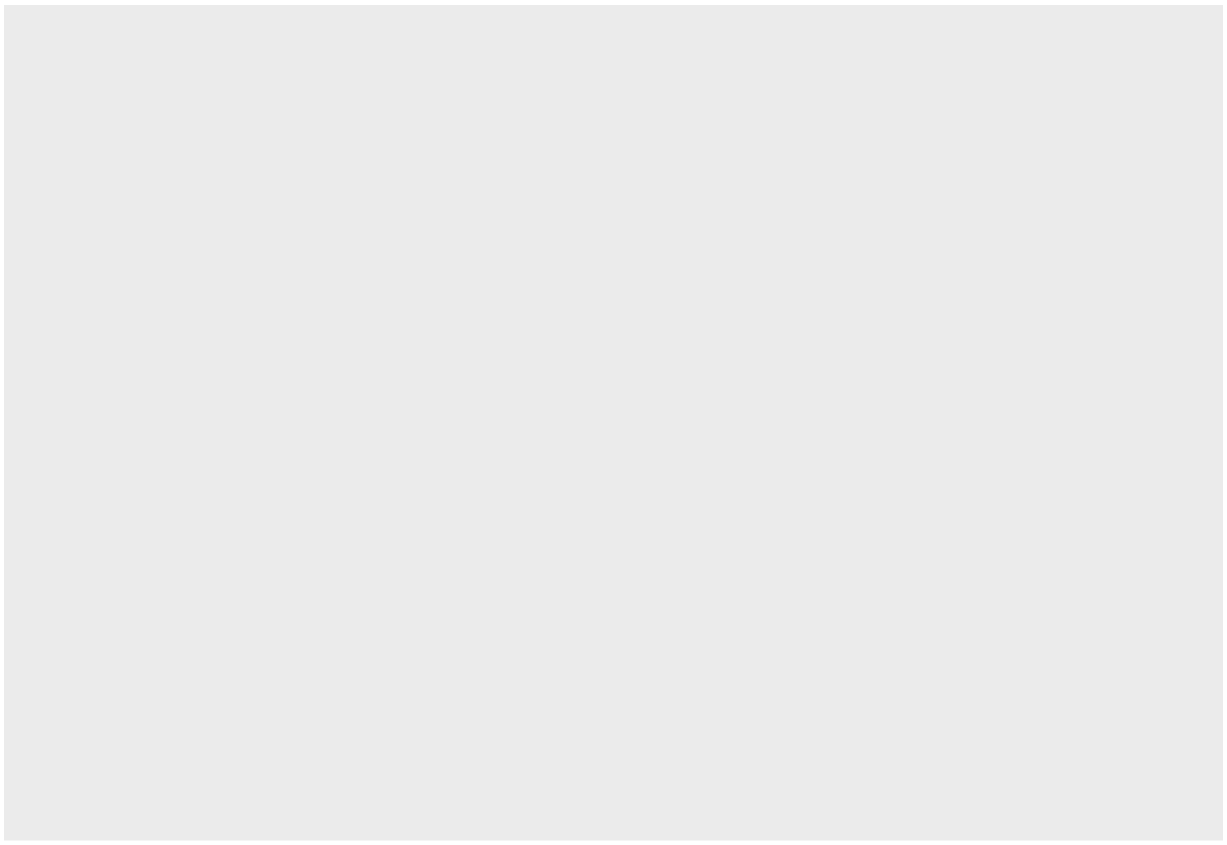
So again, the basic components we need are **data**, an **aesthetic mapping** (what data goes where), and a **geom** (how I should draw the data) that will take the following generic format:

`ggplot(data = <DATASET>) + <GEOM_FUNCTION>(mapping = aes(<AESTHETIC MAPPPINGS>))`

Let's build a plot in steps.

Try this:

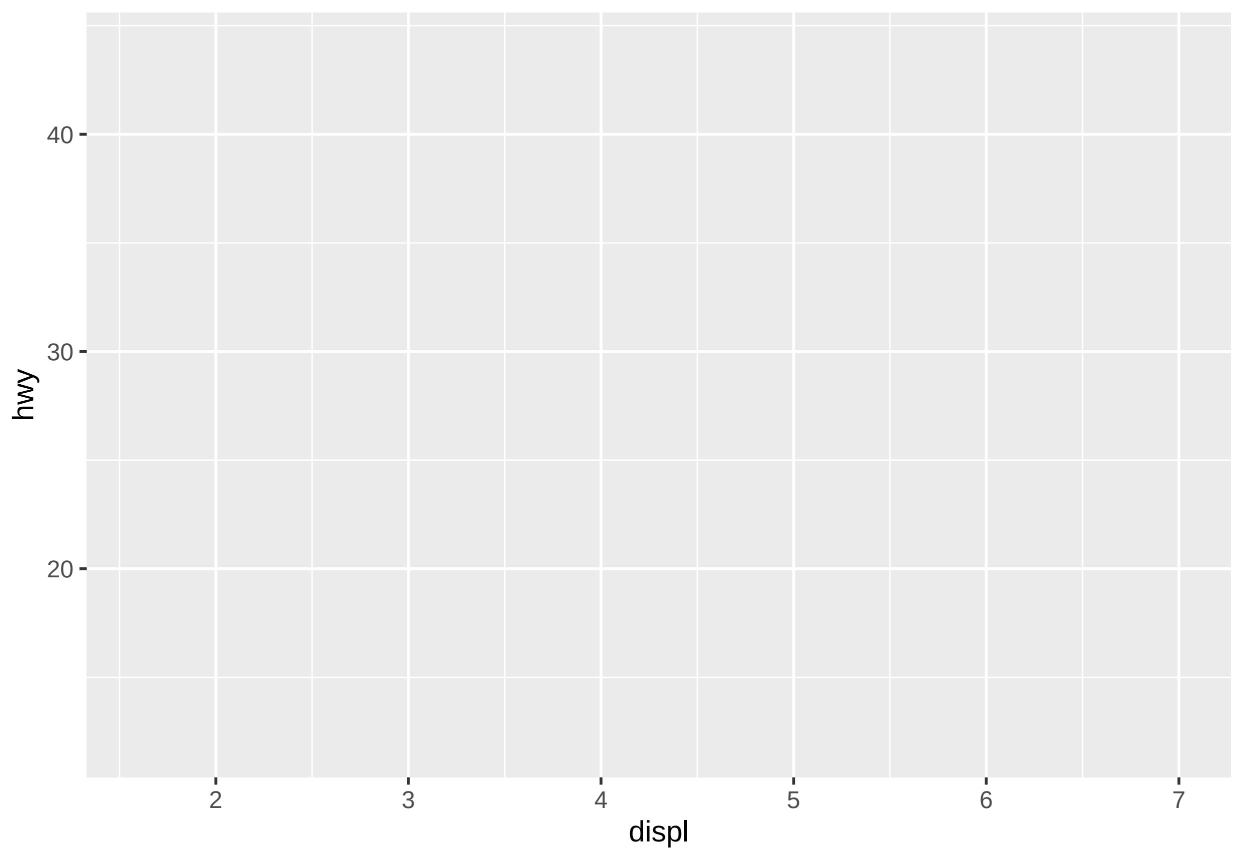
```r
ggplot(data = mpg)
```



We get a blank box. That's because ggplot has some data, but no indication as to what goes on which axis, nor any idea of how to draw that data.

Lets add the mapping

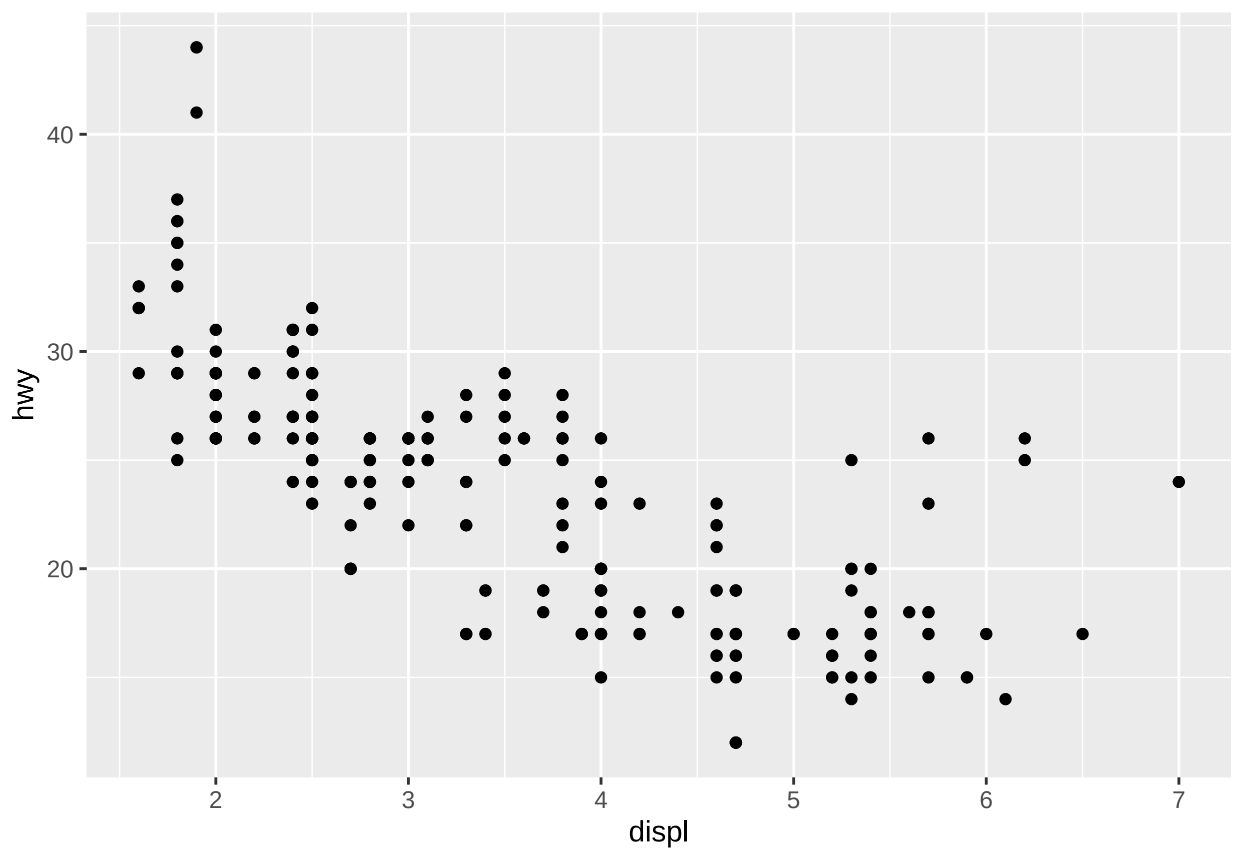
```r
ggplot(data = mpg, aes( x = displ, y = hwy))
```



Ok, now we have axes but nothing on the graph. We've told R what our axes are, but we haven't told R how or what to draw. We can do that using a `geom`.

Let's recreate the scatter plot with `geom_point`. We can do this by simply doing a + which adds a layer to our graph and then the function `geom_point()` and since R already has an X and Y mapping, it can figure out the rest.

```
ggplot(data = mpg, aes(x = displ, y = hwy)) + geom_point()
```

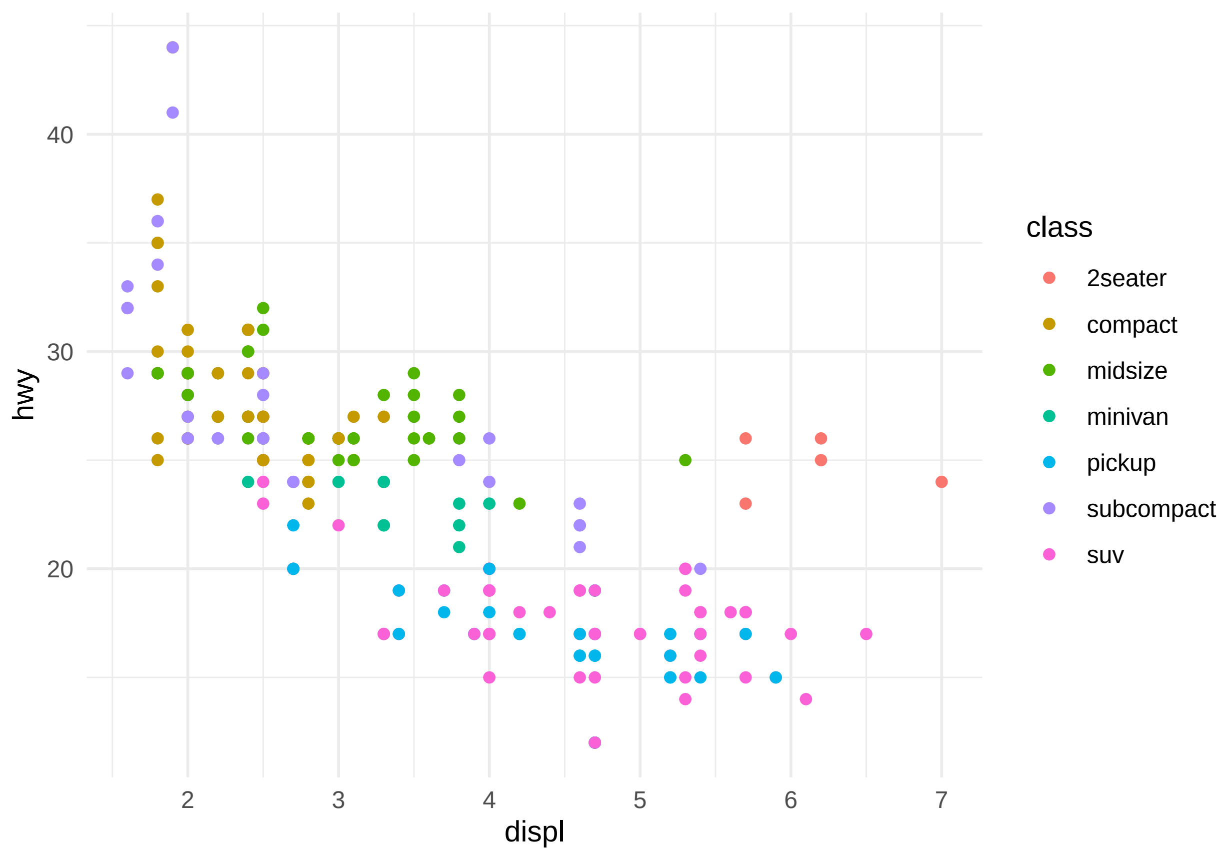


Cool, that looks like the plot we made with qplot a second ago, with a lot more work. What's the big deal?

Let's make this graph a little prettier using themes and colors.

Maybe we're interested in visualizing different classes of cars. We can color our points by class and make it pretty with the `theme_minimal()` layer.

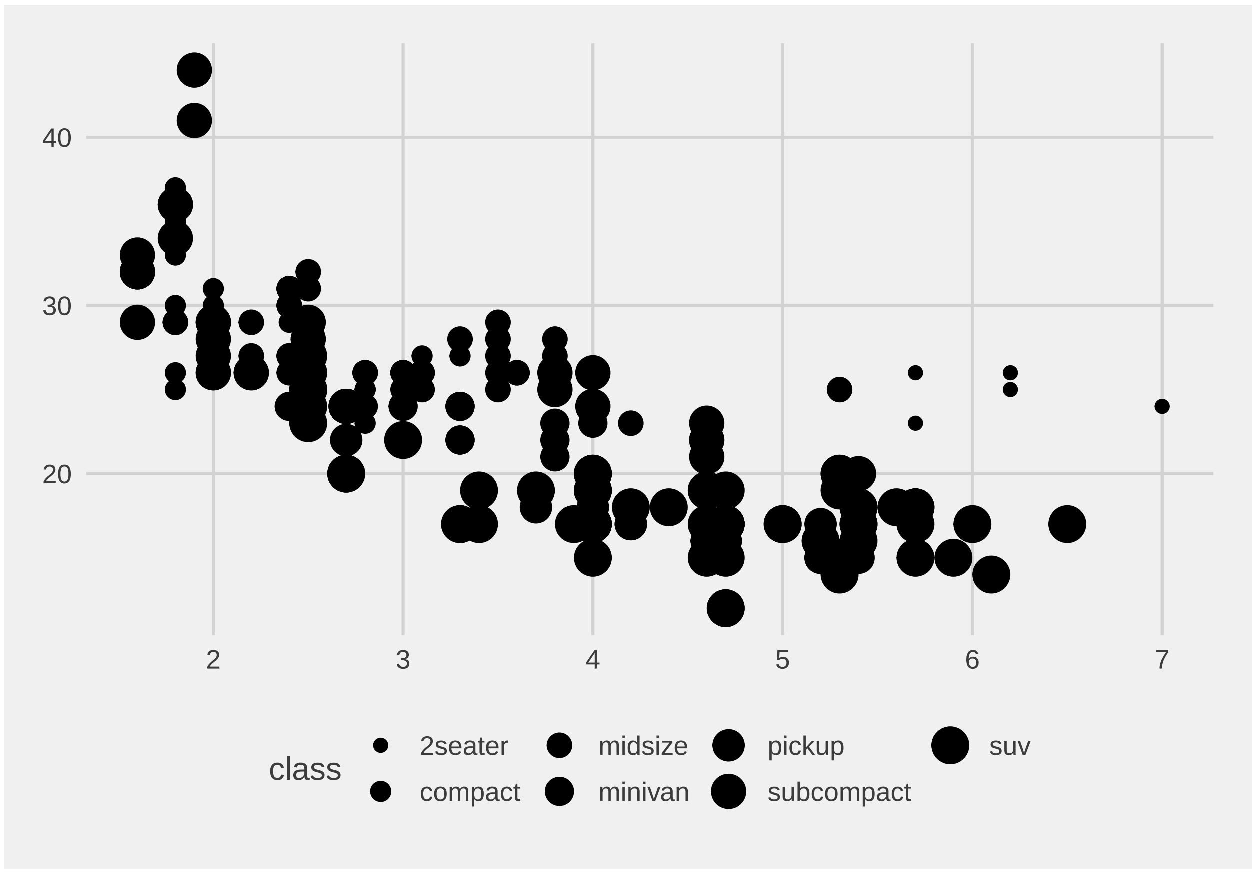
```
#We can add professional themes to our graphs by simply adding them as a layer! We can also map our aes
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class)) + theme_minimal()
```



Maybe you prefer to map your dots to size for a more color-blind friendly graph (although viridis has a library that will pick colors that are explicitly friendly to color blind individuals)

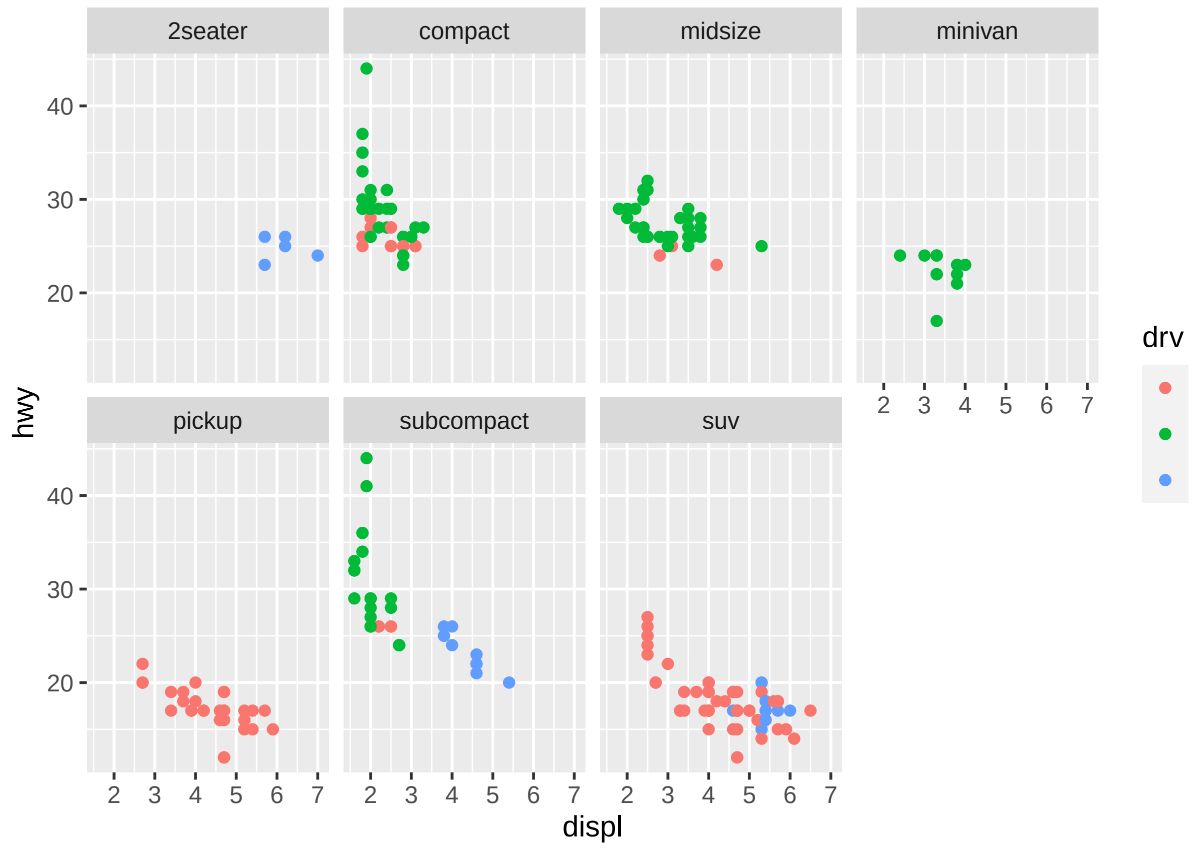
```
# can also size by variables, and use fivethirtyeight's theme! R will yell at me for using size on a di
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, size = class)) + theme_fivethirtyeight()
```

```
## Warning: Using size for a discrete variable is not advised.
```



We can also split a dataset up into a bunch of mini-graphs and plot them in a group using `facet_wrap()`

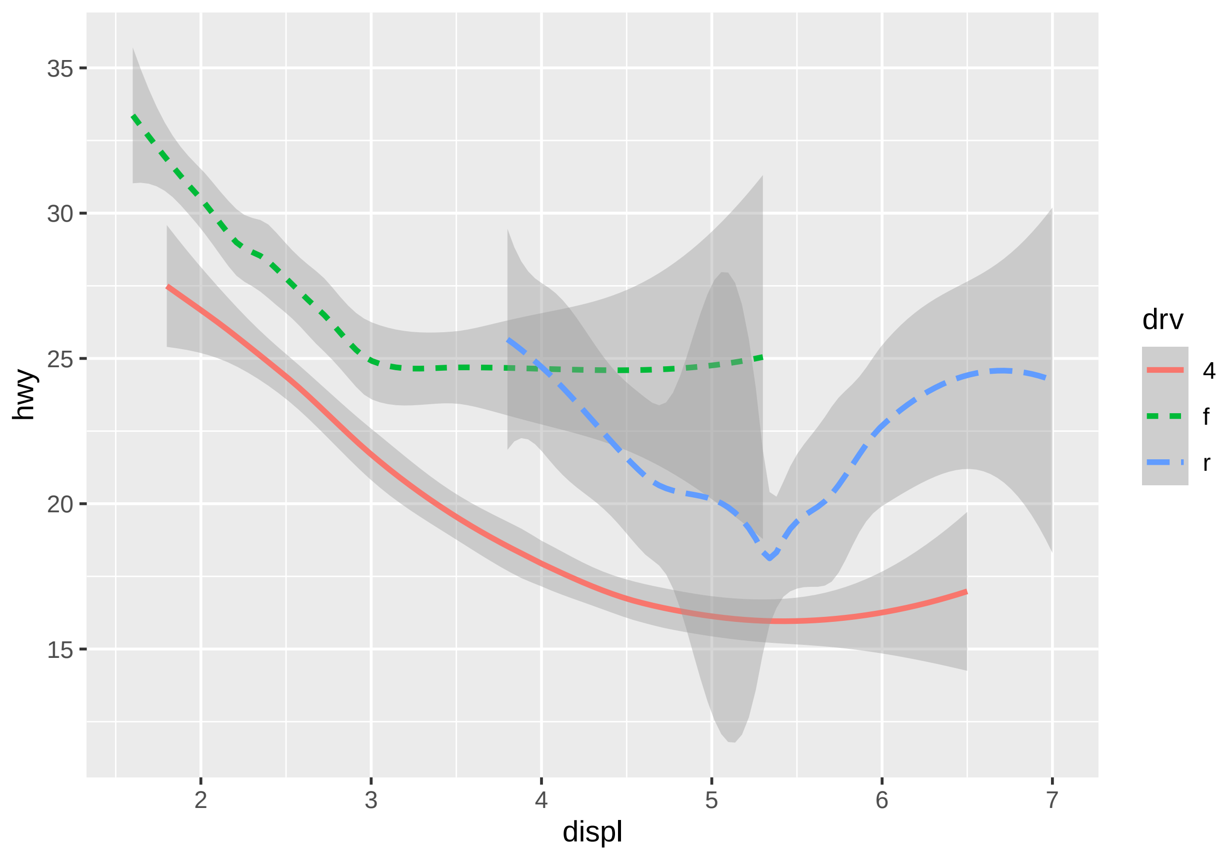
```r
#facet wrapping gives you grids
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = drv)) +
  facet_wrap(~ class, nrow = 2)
```



We can also do NON dot graphs by using a new geom. New geoms offer new ways to differentiate data, for instance, line graphs can have multiple types of line. We'll use `geom_smooth()` which fits a smooth line and gives std error ranges around said line.

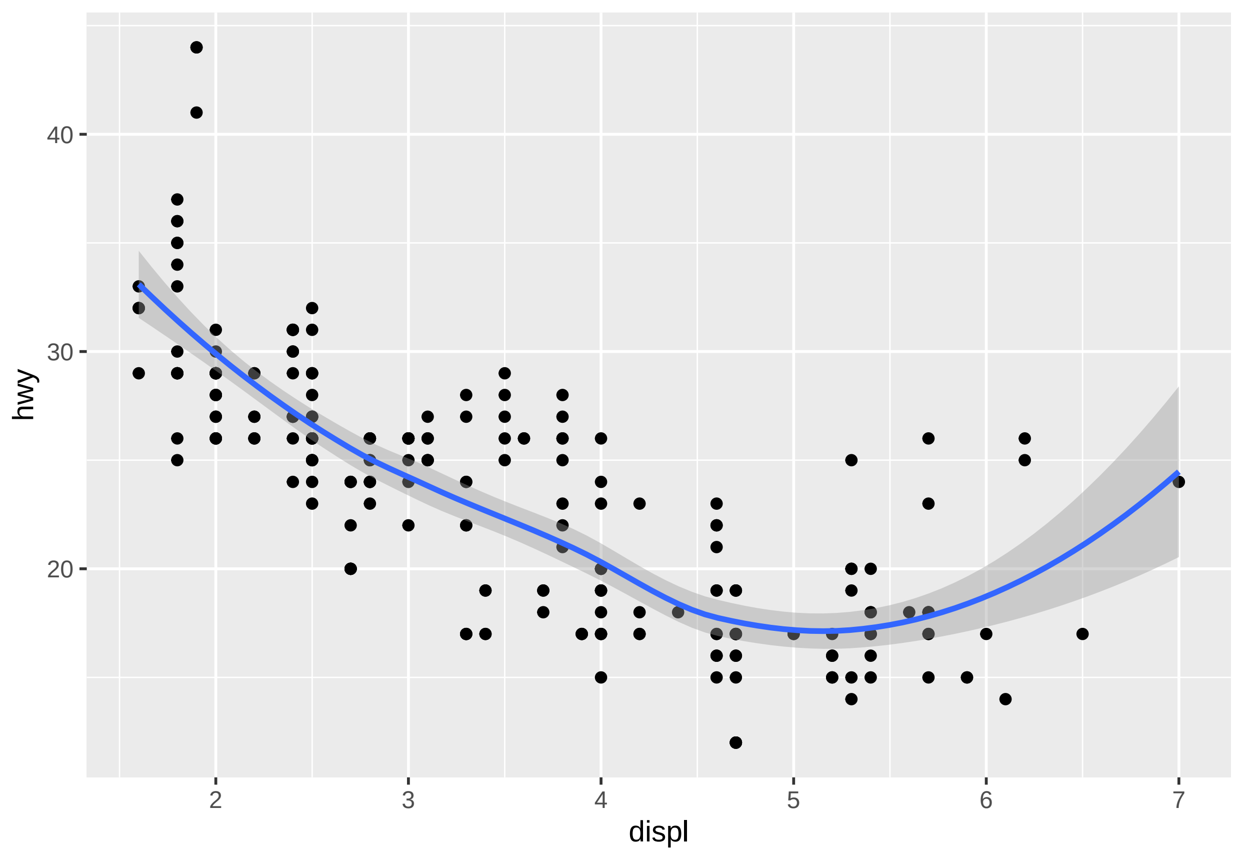
```r
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype = drv, color = drv))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



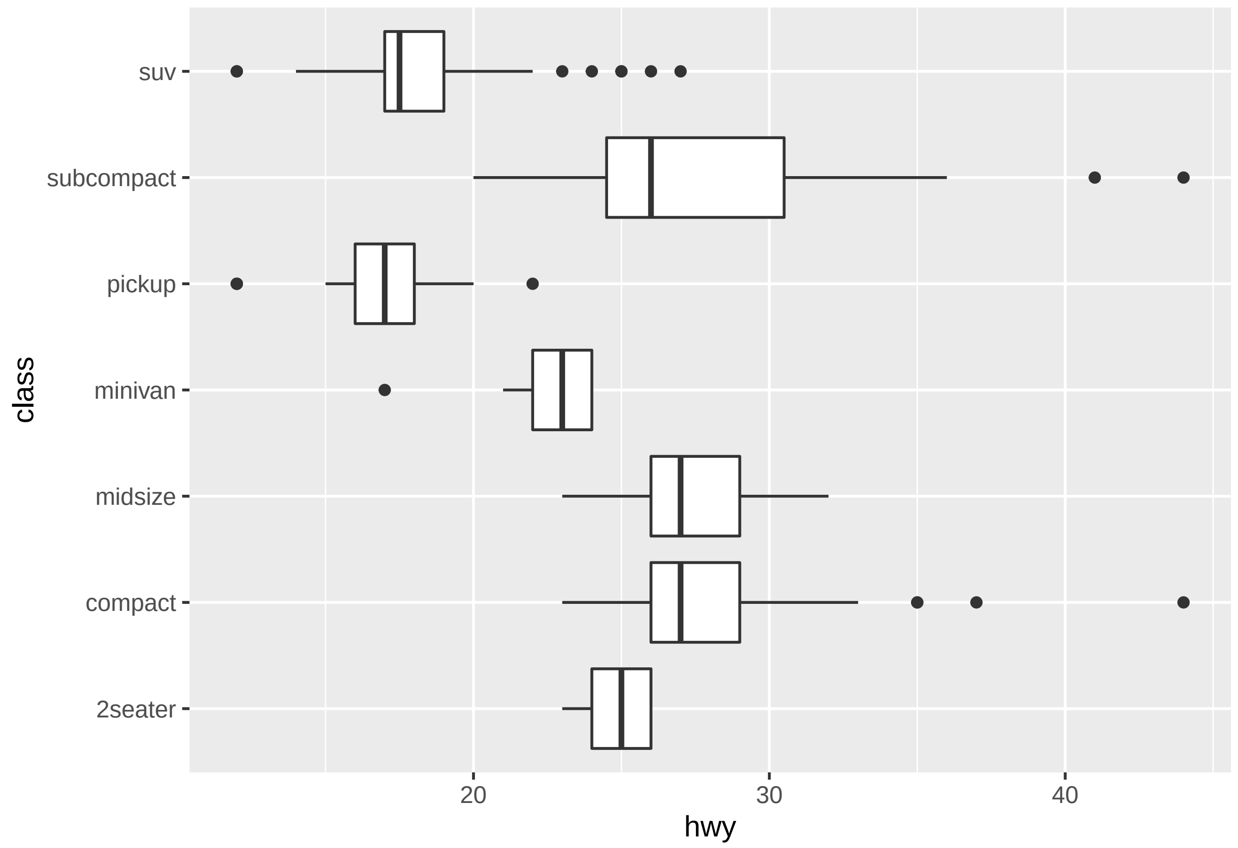
```r
#can combine different types of geoms
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Boxplots are also fairly easy to create!

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```



now that we have some basics down, we can make some plots with cool, relatively new datasets!

Tidy Tuesday is an excellent resource for R and GGplot users, both beginners and advanced. Every Tuesday, an easily accessible dataset is posted online for you to play around with to improve your skills.

You know what's cool about this? We don't even need to download the data to a csv format. We can just grab it directly from the repo in one line, that THEY wrote FOR you. You guys have the tidy tuesday files for this lab in a nice clean format posted to your course lab page, but I want to show you how to do this if you didn't want to download them to your computer, like this!

Let's read these in (bob_ross, ufo, and video_games) in…

```
#borrowed from the ufo repo
ufo_sightings <- read_csv("https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/20
```

```
## Parsed with column specification:
## cols(
##   date_time = col_character(),
##   city_area = col_character(),
##   state = col_character(),
##   country = col_character(),
##   ufo_shape = col_character(),
##   encounter_length = col_double(),
##   described_encounter_length = col_character(),
##   description = col_character(),
##   date_documented = col_character(),
##   latitude = col_double(),
##   longitude = col_double()
## )
```

```r
#borrowed from the video_games repo
video_games <- read_csv("https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2019/
```

```
## Parsed with column specification:
## cols(
##   number = col_double(),
##   game = col_character(),
##   release_date = col_character(),
##   price = col_double(),
##   owners = col_character(),
##   developer = col_character(),
##   publisher = col_character(),
##   average_playtime = col_double(),
##   median_playtime = col_double(),
##   metascore = col_double()
## )
```

```r
#borrowed from the bob_ross repo
bob_ross <- read_csv("https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2019/20
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   EPISODE = col_character(),
##   TITLE = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

However, the issue is that we need to use both of the tools we learned about today to - Get these datasets into a format we can work with - Visualize them in a way that's meaningful

Let's start with the bob ross one. This dataset is MUCH nicer in the canvas version, but I will show you how that data was produced. Just to give you an idea of what the data looks like:

```r
# Explore the dataset a bit first
head(bob_ross, 4)
```

```
## # A tibble: 4 x 69
##   EPISODE TITLE APPLE_FRAME AURORA_BOREALIS  BARN BEACH  BOAT BRIDGE BUILDING
##   <chr>   <chr>       <dbl>           <dbl> <dbl> <dbl> <dbl>  <dbl>    <dbl>
## 1 S01E01  "\"A~           0               0     0     0     0      0        0
## 2 S01E02  "\"M~           0               0     0     0     0      0        0
## 3 S01E03  "\"E~           0               0     0     0     0      0        0
## 4 S01E04  "\"W~           0               0     0     0     0      0        0
## # ... with 60 more variables: BUSHES <dbl>, CABIN <dbl>, CACTUS <dbl>,
## #   CIRCLE_FRAME <dbl>, CIRRUS <dbl>, CLIFF <dbl>, CLOUDS <dbl>, CONIFER <dbl>,
## #   CUMULUS <dbl>, DECIDUOUS <dbl>, DIANE_ANDRE <dbl>, DOCK <dbl>,
## #   DOUBLE_OVAL_FRAME <dbl>, FARM <dbl>, FENCE <dbl>, FIRE <dbl>,
## #   FLORIDA_FRAME <dbl>, FLOWERS <dbl>, FOG <dbl>, FRAMED <dbl>, GRASS <dbl>,
## #   GUEST <dbl>, HALF_CIRCLE_FRAME <dbl>, HALF_OVAL_FRAME <dbl>, HILLS <dbl>,
## #   LAKE <dbl>, LAKES <dbl>, LIGHTHOUSE <dbl>, MILL <dbl>, MOON <dbl>,
## #   MOUNTAIN <dbl>, MOUNTAINS <dbl>, NIGHT <dbl>, OCEAN <dbl>,
## #   OVAL_FRAME <dbl>, PALM_TREES <dbl>, PATH <dbl>, PERSON <dbl>,
## #   PORTRAIT <dbl>, RECTANGLE_3D_FRAME <dbl>, RECTANGULAR_FRAME <dbl>,
## #   RIVER <dbl>, ROCKS <dbl>, SEASHELL_FRAME <dbl>, SNOW <dbl>,
## #   SNOWY_MOUNTAIN <dbl>, SPLIT_FRAME <dbl>, STEVE_ROSS <dbl>, STRUCTURE <dbl>,
## #   SUN <dbl>, TOMB_FRAME <dbl>, TREE <dbl>, TREES <dbl>, TRIPLE_FRAME <dbl>,
## #   WATERFALL <dbl>, WAVES <dbl>, WINDMILL <dbl>, WINDOW_FRAME <dbl>,
## #   WINTER <dbl>, WOOD_FRAMED <dbl>
```

```r
names(bob_ross)
```

```
##  [1] "EPISODE"            "TITLE"              "APPLE_FRAME"
##  [4] "AURORA_BOREALIS"    "BARN"               "BEACH"
##  [7] "BOAT"               "BRIDGE"             "BUILDING"
## [10] "BUSHES"             "CABIN"              "CACTUS"
## [13] "CIRCLE_FRAME"       "CIRRUS"             "CLIFF"
## [16] "CLOUDS"             "CONIFER"            "CUMULUS"
## [19] "DECIDUOUS"          "DIANE_ANDRE"        "DOCK"
## [22] "DOUBLE_OVAL_FRAME"  "FARM"               "FENCE"
## [25] "FIRE"               "FLORIDA_FRAME"      "FLOWERS"
## [28] "FOG"                "FRAMED"             "GRASS"
## [31] "GUEST"              "HALF_CIRCLE_FRAME"  "HALF_OVAL_FRAME"
## [34] "HILLS"              "LAKE"               "LAKES"
## [37] "LIGHTHOUSE"         "MILL"               "MOON"
## [40] "MOUNTAIN"           "MOUNTAINS"          "NIGHT"
## [43] "OCEAN"              "OVAL_FRAME"         "PALM_TREES"
## [46] "PATH"               "PERSON"             "PORTRAIT"
## [49] "RECTANGLE_3D_FRAME" "RECTANGULAR_FRAME"  "RIVER"
## [52] "ROCKS"              "SEASHELL_FRAME"     "SNOW"
## [55] "SNOWY_MOUNTAIN"     "SPLIT_FRAME"        "STEVE_ROSS"
## [58] "STRUCTURE"          "SUN"                "TOMB_FRAME"
## [61] "TREE"               "TREES"              "TRIPLE_FRAME"
## [64] "WATERFALL"          "WAVES"              "WINDMILL"
## [67] "WINDOW_FRAME"       "WINTER"             "WOOD_FRAMED"
```

looks like we have a ton of dummy variables for whether or not items were present in a painting. To get to what you had in your course lab page, this is what needed to be done:

```r
#first, copy this very gross-looking code to make Fivethirtyeight's dataset a little easier to work wit

bob_ross_clean <- bob_ross %>% janitor::clean_names() %>%
  gather(element, present, -episode, -title) %>%
  filter(present == 1) %>%
  mutate(title = str_to_title(str_remove_all(title, '"')),
         element = str_to_title(str_replace(element, "_", " "))) %>%
  dplyr::select(-present) %>%
  separate(episode, into = c("season", "episode"), sep = "E") %>%
  mutate(season = str_extract(season, "[:digit:]+")) %>%
  mutate_at(vars(season, episode), as.integer) %>%
  arrange(season, episode)
```

Lets see what this turned our dataframe into:

```r
head(bob_ross_clean)
```

```
## # A tibble: 6 x 4
##   season episode title                element
##    <int>   <int> <chr>                <chr>
## 1      1       1 A Walk In The Woods Bushes
## 2      1       1 A Walk In The Woods Deciduous
## 3      1       1 A Walk In The Woods Grass
## 4      1       1 A Walk In The Woods River
## 5      1       1 A Walk In The Woods Tree
## 6      1       1 A Walk In The Woods Trees
```

Wow. That's nice. Let's build a dataset that COUNTS what elements were in each Bob Ross painting!

```r
# Top 10 items featured in Bob Ross paintings
counts <- bob_ross_clean %>% count(element, sort = TRUE) %>%
  arrange(desc(n))

head(counts, 10)
```

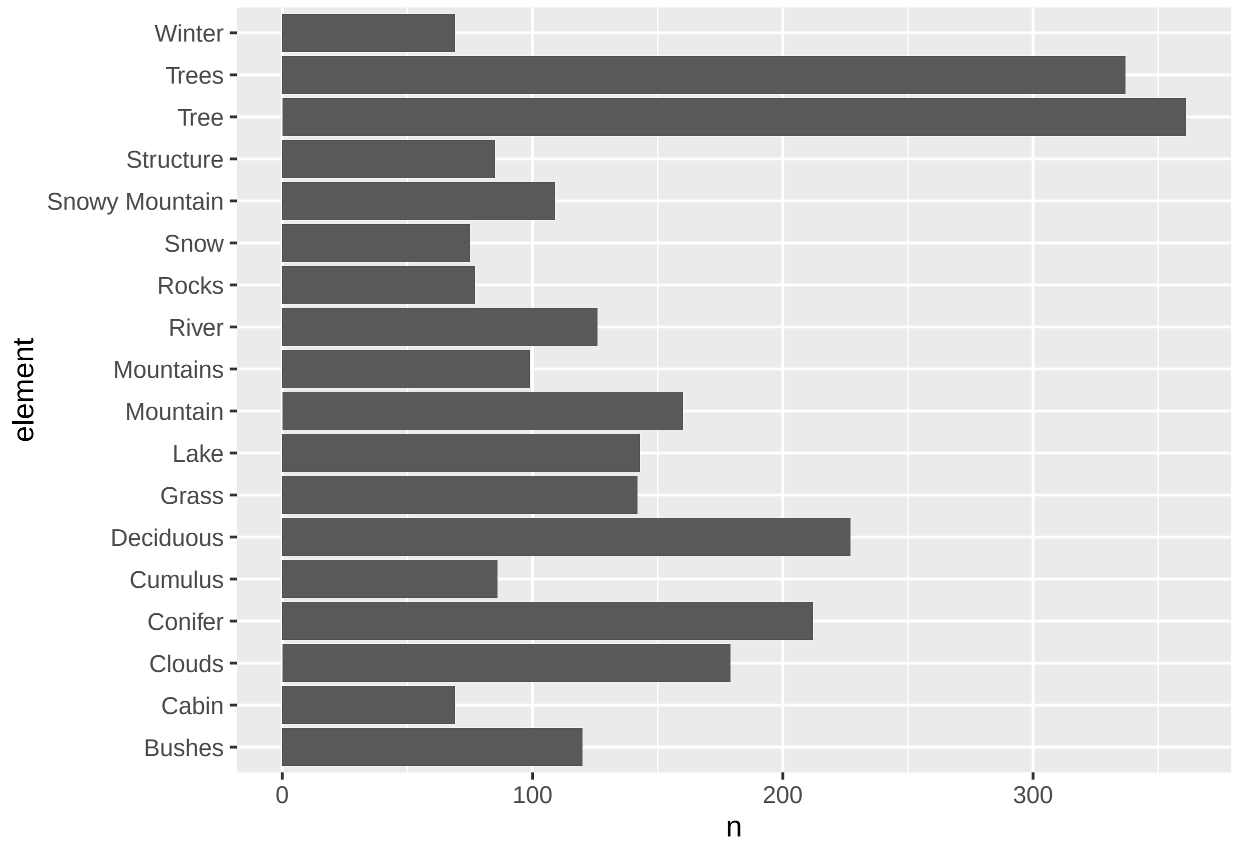
```
## # A tibble: 10 x 2
##    element       n
##    <chr>     <int>
##  1 Tree        361
##  2 Trees       337
##  3 Deciduous   227
##  4 Conifer     212
##  5 Clouds      179
##  6 Mountain    160
##  7 Lake        143
##  8 Grass       142
##  9 River       126
## 10 Bushes      120
```

Lets plot the top 18 items in bob ross paintings. We're going to, this time, assign our ggplot object to a name, the ephemerally named plot1

```r
plot1 <- counts %>% head(18) %>%
  ggplot(aes(element, n)) + geom_col() + coord_flip()
```

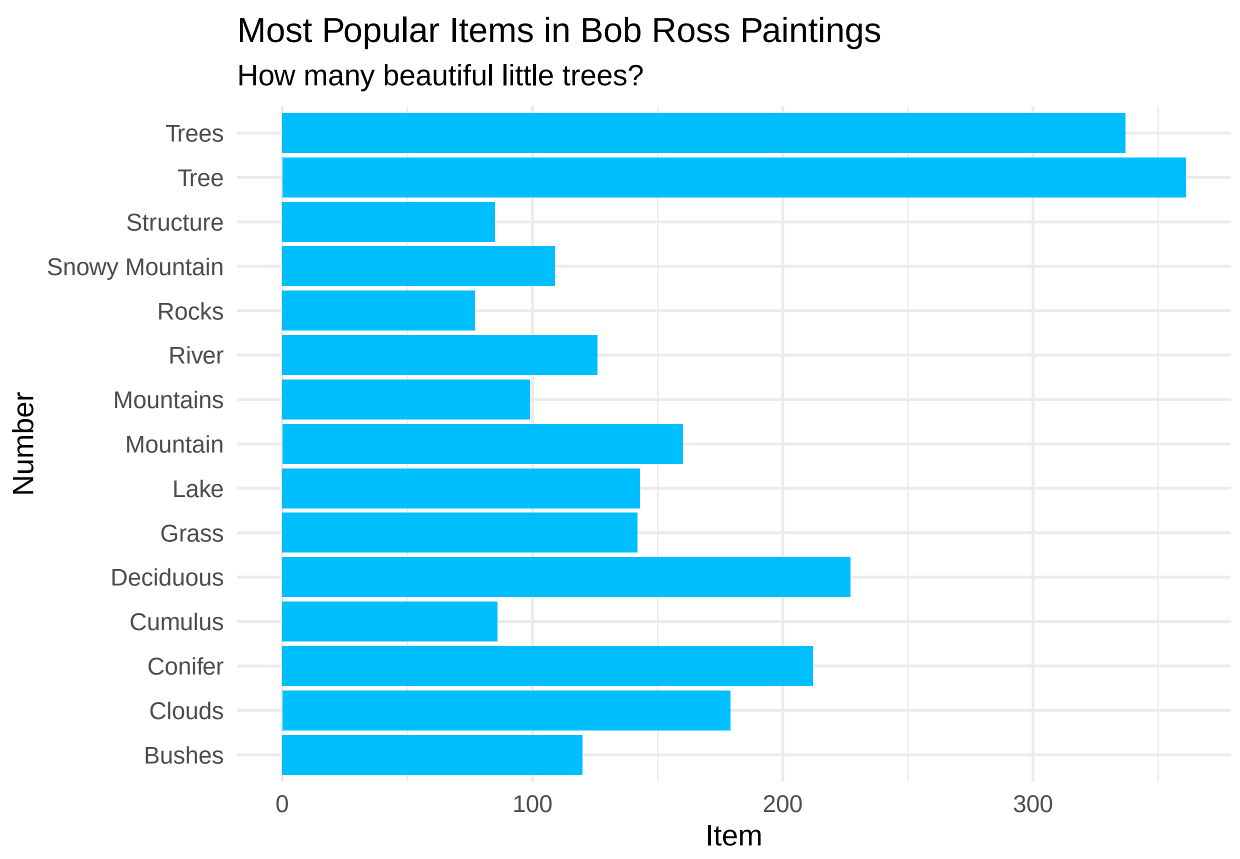
Nothing happens. Well, something happened, but we need to call the plot's name to show it.

```r
#you can call object names by typing them out
plot1
```



And different TYPES of trees. In honor of the man, let's make this more colorful and add some better axes labels. R has a TON of different colors in its pallette: you can go here to see all of them.

```r
#make it pretty by building an object with the deepbluesky color label
my_happy_little_palette = 'deepskyblue'

plot1 <- counts %>% head(15) %>%
  ggplot(aes(element, n)) + geom_col(fill = my_happy_little_palette) + coord_flip() +
  theme_minimal() +
  labs(title = "Most Popular Items in Bob Ross Paintings",
       subtitle = "How many beautiful little trees?",
       x = "Number",
       y = "Item")

#display our new improved plot
plot1
```



Ok, that was fun. Now let's play around with the video games data we loaded (if you were coding along, it should be in a df called video_games())

```r
#look at it first
head(video_games, 10)
```

```
## # A tibble: 10 x 10
##    number game  release_date price owners developer publisher average_playtime
##     <dbl> <chr> <chr>        <dbl> <chr>  <chr>     <chr>                <dbl>
##  1      1 Half~ Nov 16, 2004  9.99 10,00~ Valve     Valve                  110
##  2      3 Coun~ Nov 1, 2004   9.99 10,00~ Valve     Valve                  236
##  3     21 Coun~ Mar 1, 2004   9.99 10,00~ Valve     Valve                   10
##  4     47 Half~ Nov 1, 2004   4.99 5,000~ Valve     Valve                    0
##  5     36 Half~ Jun 1, 2004   9.99 2,000~ Valve     Valve                    0
##  6     52 CS2D  Dec 24, 2004 NA    1,000~ Unreal S~ Unreal S~               16
##  7      2 Unre~ Mar 16, 2004 15.0  500,0~ Epic Gam~ Epic Gam~                0
##  8      4 DOOM~ Aug 3, 2004   4.99 500,0~ id Softw~ id Softw~                0
##  9     14 Beyo~ Apr 27, 2004  5.99 500,0~ Larian S~ Larian S~                0
## 10     40 Hitm~ Apr 20, 2004  8.99 500,0~ Io-Inter~ Io-Inter~                0
## # ... with 2 more variables: median_playtime <dbl>, metascore <dbl>
```

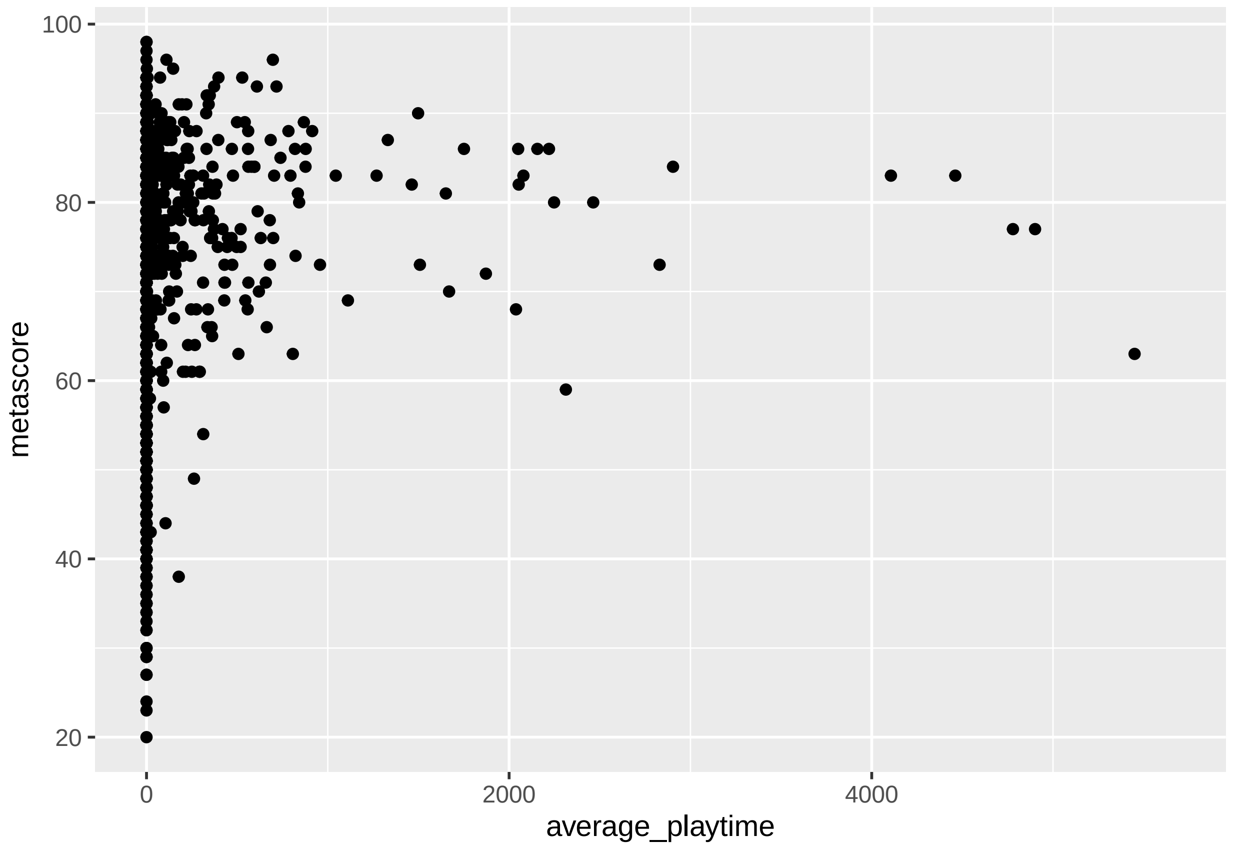
```r
names(video_games)
```

```
##  [1] "number"           "game"             "release_date"     "price"
##  [5] "owners"           "developer"        "publisher"        "average_playtime"
##  [9] "median_playtime"  "metascore"
```

Neat. Only issue I see here is that the price variable appears to be potentially incomplete (those NA values mean it was missing). Let's create a scatterplot of playtime and ratings

```r
#make a scatterplot of average playtime and ratings
video_games %>% ggplot(aes(x = average_playtime, y = metascore)) + geom_point()
```

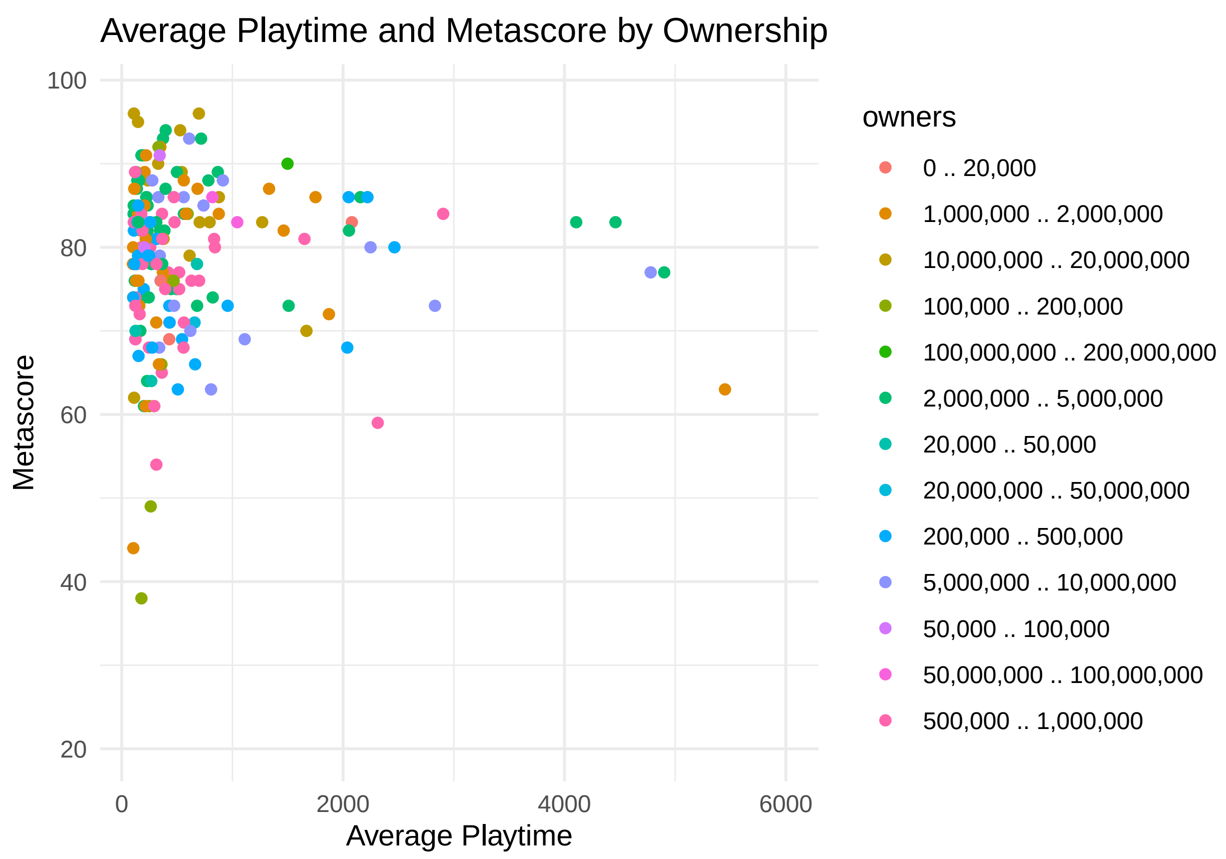
```
## Warning: Removed 23840 rows containing missing values (geom_point).
```



oof, that's really hard to read. Luckily, ggplot lets us "censor" our points by providing cutoffs to our x or y axis. Here's how to do that:

```r
#lets also add a mapping from owner count (these are discrete buckets) to colors and some new labels
video_games  %>% ggplot(aes(x = average_playtime, y = metascore, color = owners)) +
  geom_point() +
  xlim(100, 6000) + theme_minimal() +
  labs(x = "Average Playtime",
       y = "Metascore",
       title = "Average Playtime and Metascore by Ownership")
```

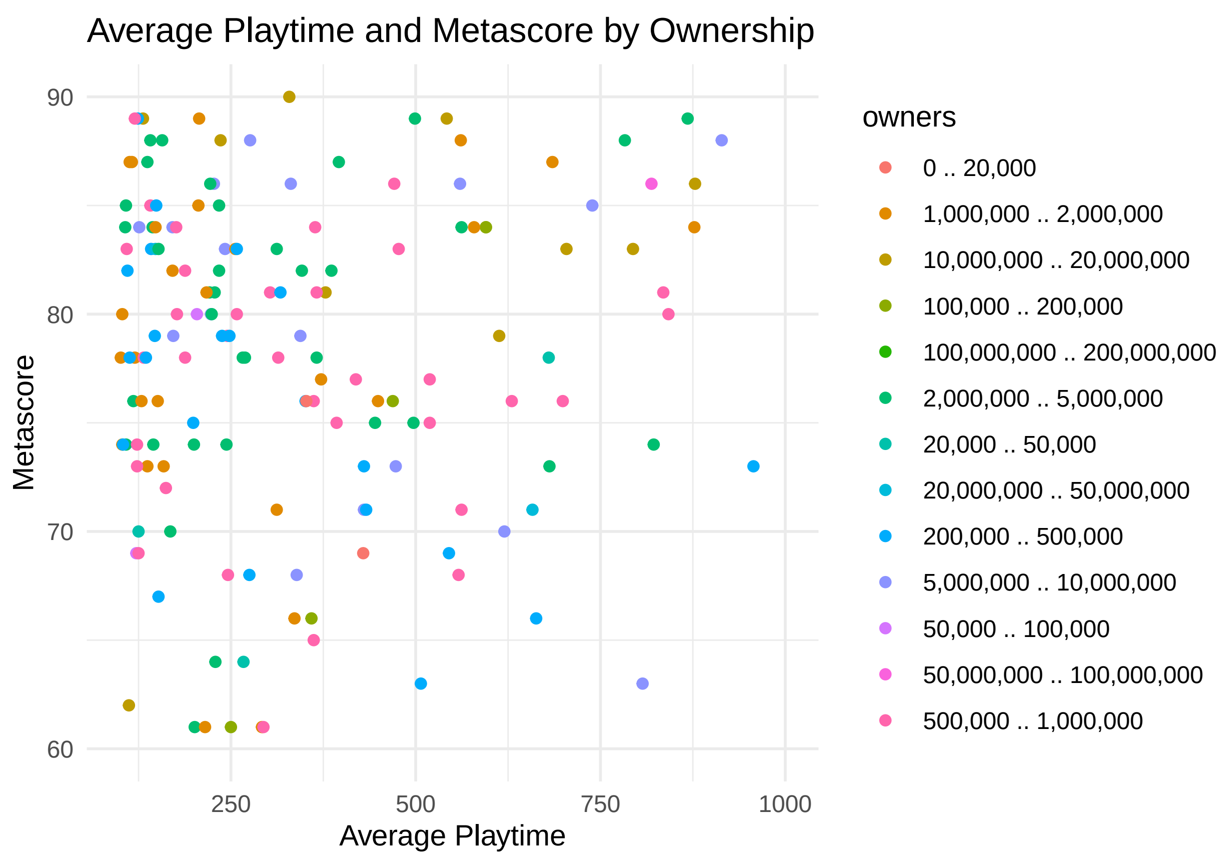
```
## Warning: Removed 26482 rows containing missing values (geom_point).
```



Cool! We can also zoom in on that big chunk of data if we want to check for any obscured trends.

```
video_games  %>% ggplot(aes(x = average_playtime, y = metascore, color = owners)) +
  geom_point() +
  xlim(100, 1000) + ylim(60,90) + theme_minimal() +
  labs(x = "Average Playtime",
       y = "Metascore",
       title = "Average Playtime and Metascore by Ownership")
```

```
## Warning: Removed 26528 rows containing missing values (geom_point).
```



Kinda cool, huh?

Lets use the tidyverse to find out what the cost of buying the library of games for all publishers would be. Can you figure out what each part of the code below does?

```
game_publishers <- video_games %>%
  group_by(publisher) %>% summarize(revenue = sum(price, na.rm = T)) %>% ungroup() %>%
  filter(publisher != "") %>%
  arrange(desc(revenue))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

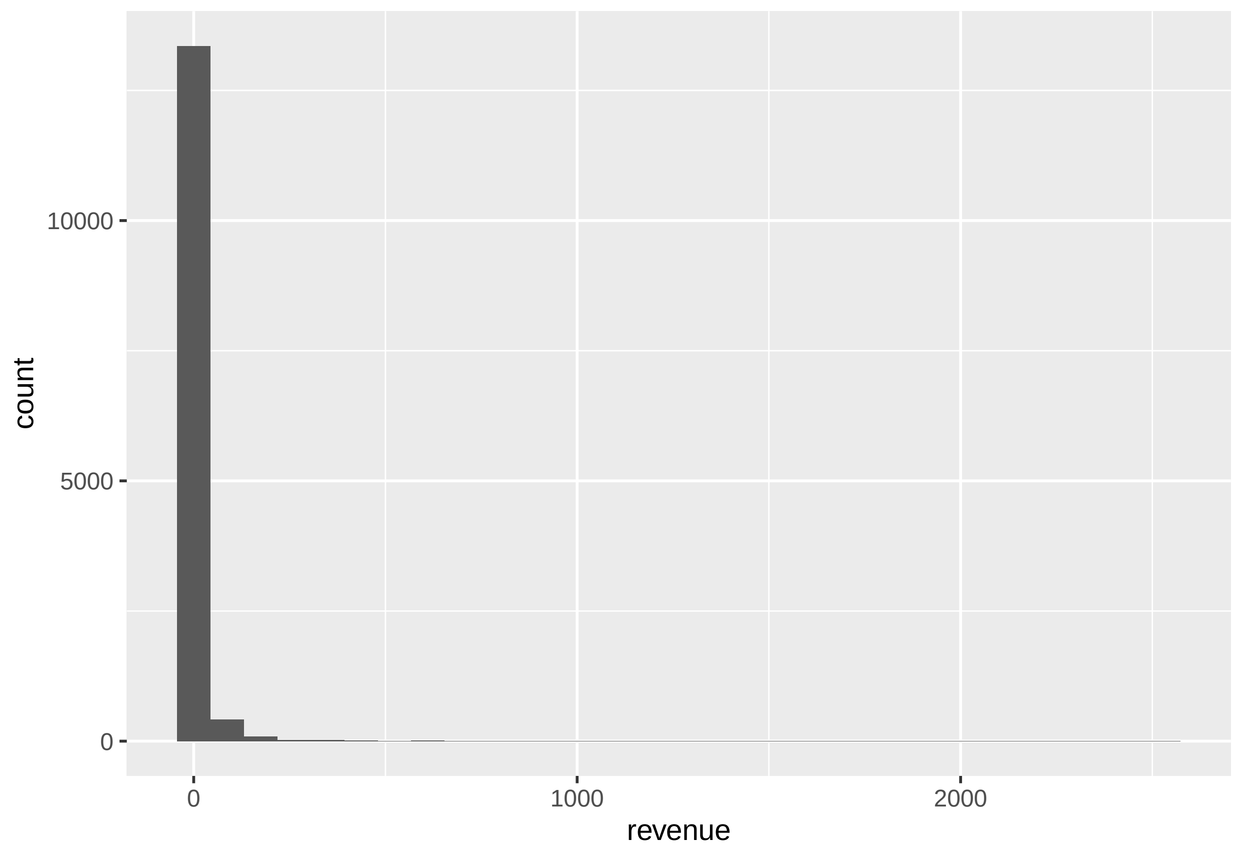
```
head(game_publishers)
```

```
## # A tibble: 6 x 2
##   publisher                     revenue
##   <chr>                           <dbl>
## 1 Big Fish Games                  2530.
## 2 KOEI TECMO GAMES CO., LTD.      2397.
## 3 Ubisoft                         2337.
## 4 Slitherine Ltd.                 2135.
## 5 MAGIX Software GmbH             1826.
## 6 BANDAI NAMCO Entertainment      1583.
```

Lets do a histogram this time. What is the Y axis on a histogram, by the way?

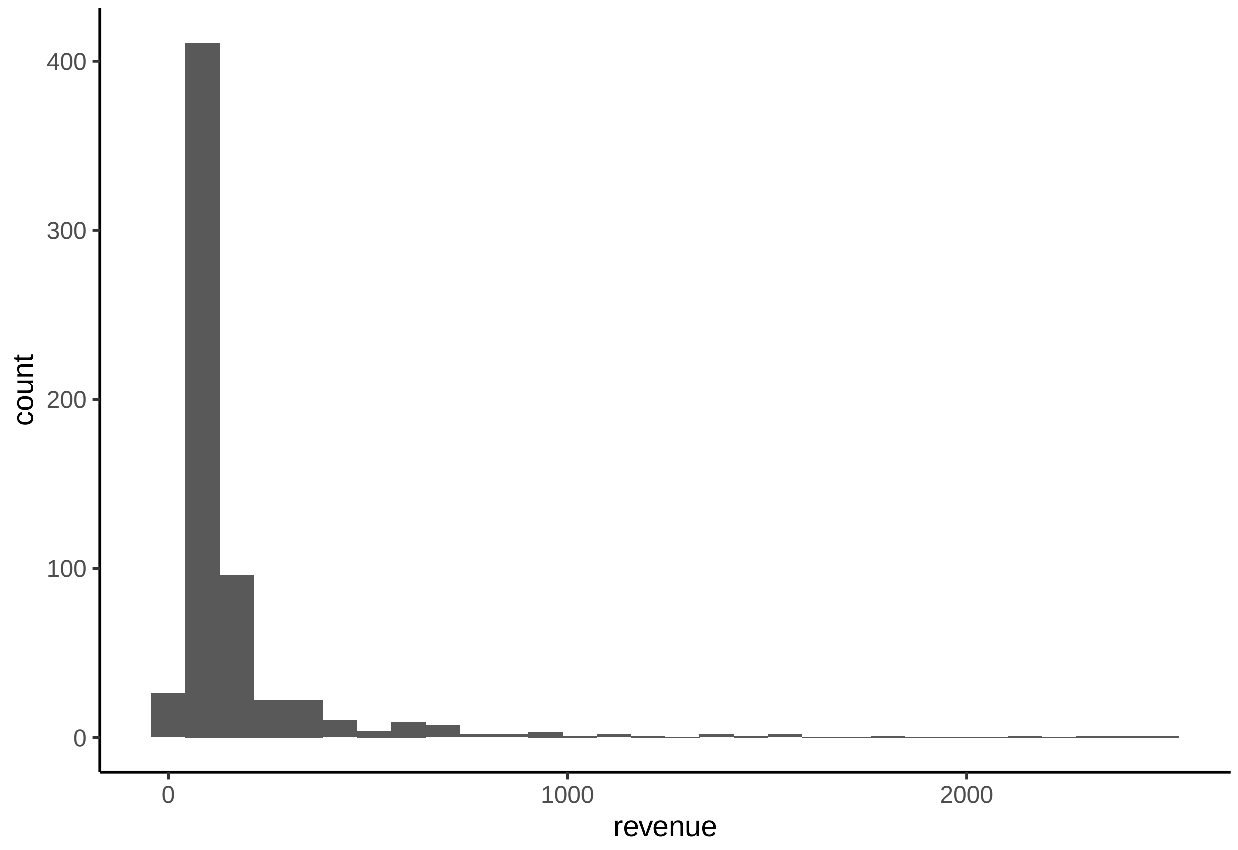
```r
#histogram
ggplot(data = game_publishers) + geom_histogram(aes(x = revenue))
```

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



```r
#we can also use the tidyverse functions right inside of the ggplot function
ggplot(data = game_publishers %>% filter(revenue > 40)) + geom_histogram(aes(x = revenue)) +
  theme_classic()
```

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



cool, moving on to some ufo sightings. Lets see what we can learn from this dataset. First, let's poke at it a bit

```
head(ufo_sightings, 5)
```

```
## # A tibble: 5 x 11
##   date_time city_area state country ufo_shape encounter_length described_encou~
##   <chr>     <chr>     <chr> <chr>   <chr>                <dbl> <chr>
## 1 10/10/19~ san marc~ tx    us      cylinder              2700 45 minutes
## 2 10/10/19~ lackland~ tx    <NA>    light                 7200 1-2 hrs
## 3 10/10/19~ chester ~ <NA>  gb      circle                  20 20 seconds
## 4 10/10/19~ edna      tx    us      circle                  20 1/2 hour
## 5 10/10/19~ kaneohe   hi    us      light                  900 15 minutes
## # ... with 4 more variables: description <chr>, date_documented <chr>,
## #   latitude <dbl>, longitude <dbl>
```

```
names(ufo_sightings)
```

```
##  [1] "date_time"                  "city_area"
##  [3] "state"                      "country"
##  [5] "ufo_shape"                  "encounter_length"
##  [7] "described_encounter_length" "description"
##  [9] "date_documented"            "latitude"
## [11] "longitude"
```

Let's take a closer look at this date column:

```
class(ufo_sightings$date_time)
```

```
## [1] "character"
```

Okay, it's a character. But wouldn't it be nice if we could use these dates/times in our plots? Like for a time series? We can.

Let's convert it using the `lubridate` package, and get rid of any NA values for country. Lubridate will let us transform a string like this into a different kind of object that lets us access all sorts of different slices of information.

```
library(lubridate)
```

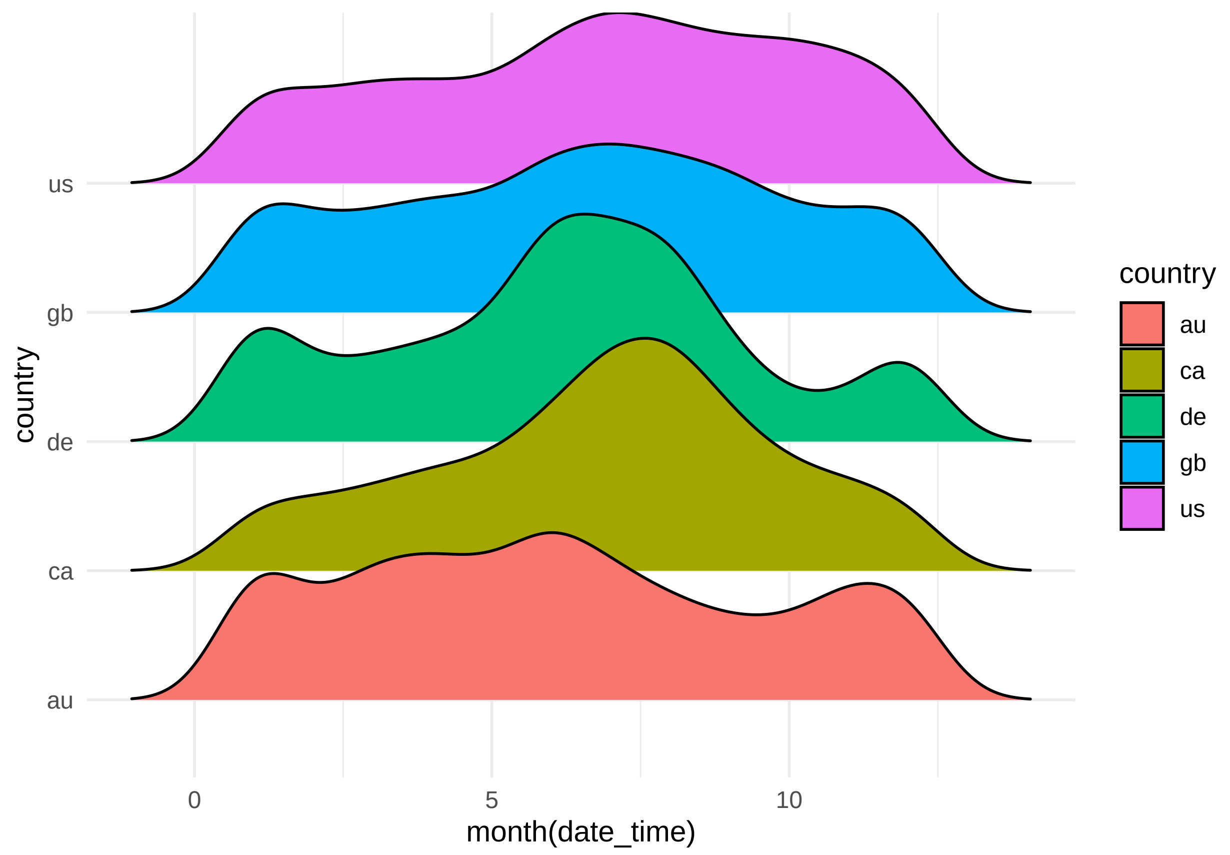
```
##
## Attaching package: 'lubridate'

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

```
ufo <- ufo_sightings %>%
  mutate(date_time = parse_date_time(date_time, 'mdy_HM')) %>%
  filter(country != "NA")
```

let's look at what month these ufo sightings are happening and break it down by country.

```
library(ggridges)
ufo %>% ggplot(aes(x = month(date_time), y = country, fill = country)) + geom_density_ridges() +
  theme_minimal()
```

```
## Picking joint bandwidth of 0.685
```

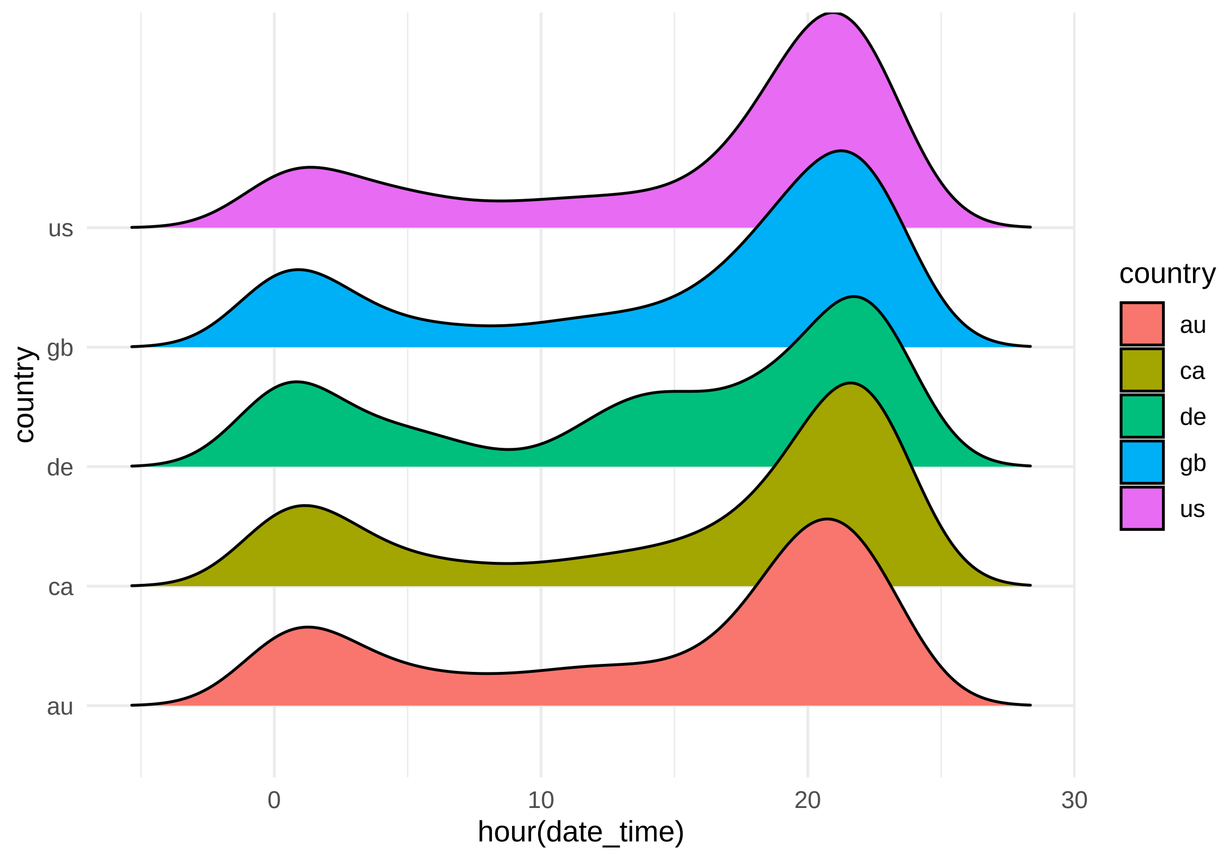


It looks like Aliens prefer the late summer! (think about when summer is in AU)

I wonder what time of day these sightings are happening?

```
ufo %>% ggplot(aes(x = hour(date_time), y = country, fill = country)) + geom_density_ridges() +
  theme_minimal()
```

```
## Picking joint bandwidth of 1.78
```



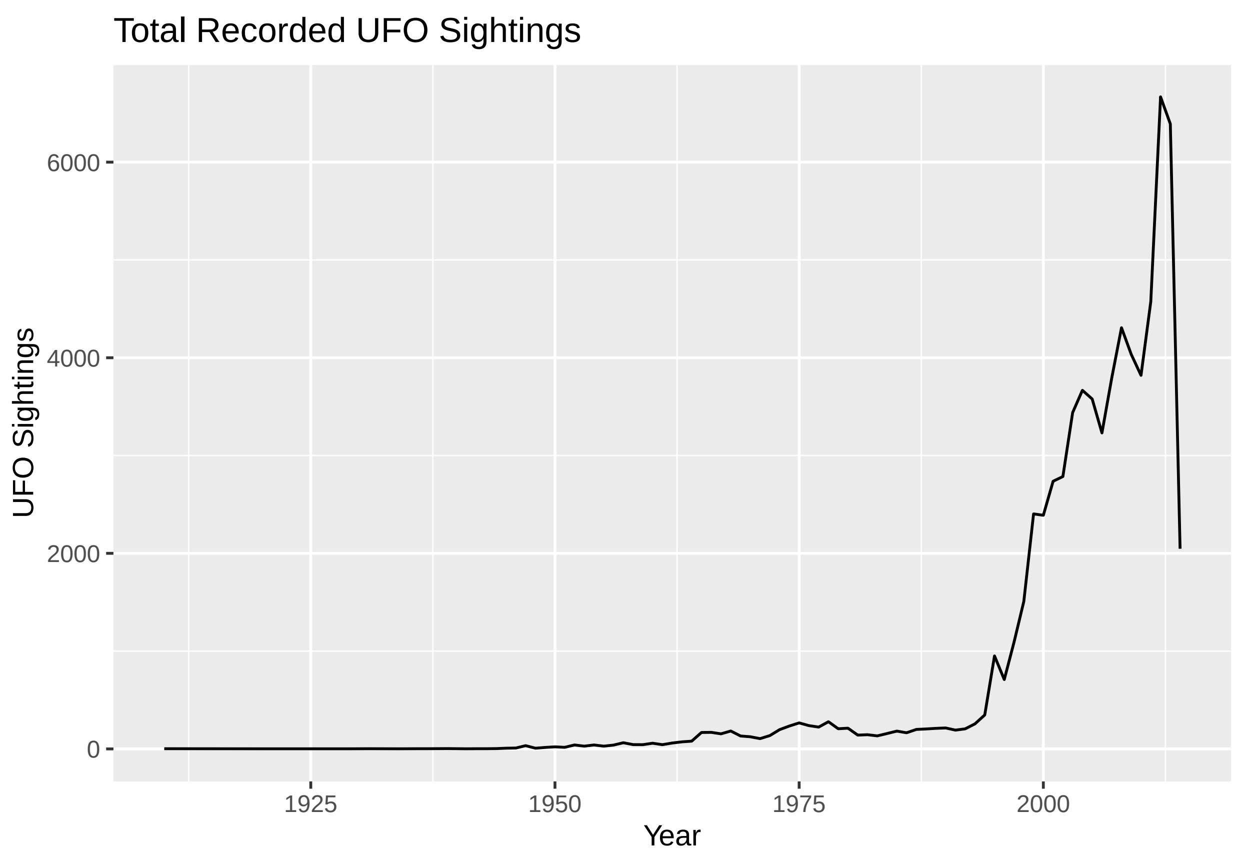
Mostly at night! That makes sense!

Now let's look at total ufo sightings per year

```
ufo_total <- ufo %>% group_by(year(date_time)) %>% summarize(total = n())
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
names(ufo_total) <- c("year", "total")

ggplot(aes(x = year, y = total), data = ufo_total) + geom_line() +
  labs(x = "Year",
       y = "UFO Sightings",
       title = "Total Recorded UFO Sightings")
```



Awesome! We seem to be getting a big spike in alien sightings sometime in the 90s and mid 2000s.

Lesson 1: Functions, loops continued

```r
if (!require("pacman")) install.packages("pacman")
pacman::p_load(tidyverse)
```

Quick review

We have already seen and used a multitude of functions in R. Some of these functions come pre-packaged with base R (e.g. `mean()`), while others are from external packages (e.g. `dplyr::filter()`). Regardless of where they come from, functions in R all adopt the same basic syntax:

```r
function_name(ARGUMENTS)
```

For much of the time, we will rely on functions that other people have written for us. However, you can — and should! — write your own functions too. This is easy to do with the generic **`function()`** function.[1] The syntax will again look familiar to you:

[1]Yes, it's a function that let's you write functions. Very meta.

```
function(ARGUMENTS) {
  OPERATIONS
  return(VALUE)
}
```

While it's possible and reasonably common to write anonymous functions like the above, we typically write functions because we want to reuse code. For this typical use-case it makes sense to name our functions.[2]

```
my_func <-
  function(ARGUMENTS) {
    OPERATIONS
    return(VALUE)
  }
```

For some short functions, you don't need to invoke the curly brackets or assign an explicit return object (more on this below). In these cases, you can just write your function on a single line:

```
my_short_func <- function(ARGUMENTS) OPERATION
```

Try to give your functions short, pithy names that are informative to both you and anyone else reading your code. This is harder than it sounds, but will pay off down the road.

A simple example

Let's write out a simple example function, which gives the square of an input number.

```
square <-       ## Our function name
  function(x) { ## The argument(s) that our function takes as an input
    x^2         ## The operation(s) that our function performs
  }
```

Test it.

```
square(3)
```

```
## [1] 9
```

Great, it works. Note that for this simple example we could have written everything on a single line; i.e. `square <- function(x) x^2` would work just as well. (Confirm this for yourself.) However, we're about to add some extra conditions and options to our function, which will strongly favour the multi-line format.

Aside: I want to stress that our new `square()` function is not particularly exciting… or, indeed, useful. R's built-in arithmetic functions already take care of (vectorised) exponentiation and do so very efficiently. (See `?Arithmetic`.) However, we're going to continue with this conceptually simple example, since it will provide a clear framework for demonstrating some general principles about functions in R.

[2]Remember: "In R, everything is an object and everything has a name."

Specifying return values

Notice that we didn't specify a return value for our function. This will work in many cases because R's default behaviour is to automatically return the final object that you created within the function. However, this won't always be the case. I thus advise that you get into the habit of assigning the return object(s) explicitly. Let's modify our function to do exactly that.

```
square <-
  function(x) {
    x_sq <- x^2   ## Create an intermediary object (that will be returned)
    return(x_sq)  ## The value(s) or object(s) that we want returned.
  }
```

Again, test that it works.

```
square(5)
```

```
## [1] 25
```

Specifying an explicit return value is also helpful when we want to return more than one object. For example, let's say that we want to remind our user what variable they used as an argument in our function:

```
square <-
  function(x) { ## The argument(s) that our function takes as an input
    x_sq <- x^2 ## The operation(s) that our function performs
    return(list(value=x, value_squared=x_sq)) ## The list of object(s) that we want returned.
  }
```

```
square(3)
```

```
## $value
## [1] 3
##
## $value_squared
## [1] 9
```

Note that multiple return objects have to be combined in a list. I didn't have to name these separate list elements — i.e. "value" and "value_squared" — but it will be helpful for users of our function. Nevertheless, remember that many objects in R contain multiple elements (vectors, data frames, and lists are all good examples of this). So we can also specify one of these "array"-type objects within the function itself if that provides a more convenient form of output. For example, we could combine the input and output values into a data frame:

```
square <-
  function(x) {
    x_sq <- x^2
    ## tibble is a syntax for creating simple data frames.
    ## Provides a 'tbl_df' class (the 'tibble') that provides stricter checking and better formatting t
    df <- tibble(value=x, value_squared=x_sq) ## Bundle up our input and output values into a convenien
    return(df)
  }
```

Test.

```r
square(12)
```

```
## # A tibble: 1 x 2
##   value value_squared
##   <dbl>         <dbl>
## 1    12           144
```

Specifying default argument values

Another thing worth noting about R functions is that you can assign default argument values. You have already encountered some examples of this in action.[3] We can add a default option to our own function pretty easily.

```r
square <-
  function(x = 1) { ## Setting the default argument value
    x_sq <- x^2
    df <- tibble(value=x, value_squared=x_sq)
    return(df)
  }
```

```r
square() ## Will take the default value of 1 since we didn't provide an alternative.
```

```
## # A tibble: 1 x 2
##   value value_squared
##   <dbl>         <dbl>
## 1     1             1
```

```r
square(2) ## Now takes the explicit value that we give it.
```

```
## # A tibble: 1 x 2
##   value value_squared
##   <dbl>         <dbl>
## 1     2             4
```

We'll return the issue of specifying default values (and handling invalid inputs) in the next lecture on function debugging.

Before continuing, I want to highlight the fact that none of the intermediate objects that we created within the above functions (`x_sq`, `df`, etc.) have made their way into our global environment. Take a moment to confirm this for yourself by looking in the "Environment" pane of your RStudio session.

Iteration

The most important early programming skill to master is iteration. In particular, we want to write functions that can iterate — or *map* — over a set of inputs.[4] By far the most common way to iterate across different programming languages is *for* loops. Indeed, we already saw some examples of *for* loops back in the shell lecture (see here). However, while R certainly accepts standard *for* loops, I'm going to advocate that you adopt what is known as a "functional programming" approach to writing loops. Let's dive into the reasons why and how these approaches differ.

[3] E.g. Type `?rnorm` and see that it provides a default mean and standard deviation of 0 and 1, respectively.

[4] Our focus today will only be on sequential iteration, but we'll return to parallel iteration in the lecture after next.

Vectorisation

The first question you need to ask is: "Do I need to iterate at all?" You may remember from a previous lecture that I spoke about R being *vectorised*. Which is to say that you can apply a function to every element of a vector at once, rather than one at a time. Let's demonstrate this property with our `square` function:

```r
square(1:5)
```

```
## # A tibble: 5 x 2
##   value value_squared
##   <int>         <dbl>
## 1     1             1
## 2     2             4
## 3     3             9
## 4     4            16
## 5     5            25
```

```r
square(c(2, 4))
```

```
## # A tibble: 2 x 2
##   value value_squared
##   <dbl>         <dbl>
## 1     2             4
## 2     4            16
```

So you may not need to worry about explicit iteration at all. That being said, there are certainly cases where you will need to worry about it. Let's explore with some simple examples (some of which are already vectorised) that provide a mental springboard for thinking about more complex cases.

for loops. Simple, but limited (and sometimes dangerous)

In R, standard *for* loops take a pretty intuitive form. For example:

```r
for(i in 1:10) print(LETTERS[i])
```

```
## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
## [1] "E"
## [1] "F"
## [1] "G"
## [1] "H"
## [1] "I"
## [1] "J"
```

Note that in cases where we want to "grow" an object via a *for* loop, we first have to create an empty (or NULL) object.

```r
kelvin <- 300:305
fahrenheit <- NULL
# fahrenheit <- vector("double", length(kelvin)) ## Better than the above. Why?
for(k in 1:length(kelvin)) {
  fahrenheit[k] <- kelvin[k] * 9/5 - 459.67
}
fahrenheit
```

```
## [1] 80.33 82.13 83.93 85.73 87.53 89.33
```

Unfortunately, basic *for* loops in R also come with some downsides. Historically, they used to be significantly slower and memory consumptive than alternative methods (see below). This has largely been resolved, but I've still run into cases where an inconspicuous *for* loop has brought an entire analysis crashing to its knees.[5] The bigger problem with *for* loops, however, is that they deviate from the norms and best practices of **functional programming**.

Functional programming

The concept of functional programming is arguably the most important thing you can take away from today's lecture. Thus, while it can certainly be applied to iteration, I'm going to cover it in its own section.

FP defined

Here is Hadley Wickham explaining the key idea:

> R, at its heart, is a functional programming (FP) language. This means that it provides many tools for the creation and manipulation of functions. In particular, R has what's known as first class functions. You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

That may seem a little abstract, so here is video of Hadley giving a much more intuitive explanation through a series of examples:

Summary: *for* loops tend to emphasise the *objects* that we're working with (say, a vector of numbers) rather than the *operations* that we want to apply to them (say, get the mean or median or whatever). This is inefficient because it requires us to continually write out the *for* loops by hand rather than getting an R function to create the for-loop for us.

As a corollary, *for* loops also pollute our global environment with the variables that are used as counting variables. Take a look at your "Environment" pane in RStudio. What do you see? In addition to the `kelvin` and `fahrenheit` vectors that we created, we also see two variables `i` and `k` (equal to the last value of their respective loops). Creating these auxilliary variables is almost certainly not an intended outcome when your write a for-loop.[6] More worringly, they can cause programming errors when we inadvertently refer to a similarly-named variable elsewhere in our script. So we best remove them manually as soon as we're finished with a loop.

[5] Exhibit A. Trust me: debugging these cases is not much fun.

[6] The best case I can think of is when you are trying to keep track of the number of loops, but even then there are much better ways of doing this.

```r
rm(i,k)
```

Another annoyance arrived in cases where we want to "grow" an object as we iterate over it (e.g. the `fahrenheit` object in our second example). In order to do this with a *for* loop, we had to go through the rigmarole of creating an empty object first.

FP allows to avoid the explicit use of loop constructs and its associated downsides by `*apply` family of functions in base R.

Let's explore these in more depth.

lapply and co.

Base R contains a very useful family of `*apply` functions. I won't go through all of these here — see `?apply` or this blog post among numerous excellent resources — but they all follow a similar philosophy and syntax. The good news is that this syntax very closely mimics the syntax of basic for-loops. For example, consider the code below, which is analgous to our first *for* loop above, but now invokes a `base::lapply()` call instead.

```r
# for(i in 1:10) print(LETTERS[i]) ## Our original for loop (for comparison)
lapply(1:10, function(i) LETTERS[i])
```

```
## [[1]]
## [1] "A"
##
## [[2]]
## [1] "B"
##
## [[3]]
## [1] "C"
##
## [[4]]
## [1] "D"
##
## [[5]]
## [1] "E"
##
## [[6]]
## [1] "F"
##
## [[7]]
## [1] "G"
##
## [[8]]
## [1] "H"
##
## [[9]]
## [1] "I"
##
## [[10]]
## [1] "J"
```

A couple of things to notice.

First, check your "Environment" pane in RStudio. Do you see an object called "i" in the Global Environment? (The answer should be"no".) Again, this is because of R's lexical scoping rules, which mean that any object created and invoked by a function is evaluated in a sandboxed environment outside of your global environment.

Second, notice how little the basic syntax changed when switching over from `for()` to `lapply()`. Yes, there are some differences, but the essential structure remains the same: We first provide the iteration list (`1:10`) and then specify the desired function or operation (`LETTERS[i]`).

Third, notice that the returned object is a *list*. The `lapply()` function can take various input types as arguments — vectors, data frames, lists — but always returns a list, where each element of the returned list is the result from one iteration of the loop. (So now you know where the "l" in "**l**apply" comes from.)

Okay, but what if you don't want the output in list form? There several options here.[7] However, the method that I use most commonly is to bind the different list elements into a single data frame with `dplyr::bind_rows()`. For example, here's a a slightly modified version of our function that now yields a data frame:

```
# library(tidyverse) ## Already loaded

lapply(1:10, function(i) {
  df <- tibble(num = i, let = LETTERS[i])
  return(df)
  }) %>%
  bind_rows()
```

```
## # A tibble: 10 x 2
##      num let
##    <int> <chr>
##  1     1 A
##  2     2 B
##  3     3 C
##  4     4 D
##  5     5 E
##  6     6 F
##  7     7 G
##  8     8 H
##  9     9 I
## 10    10 J
```

Taking a step back, while the default list-return behaviour may not sound ideal at first, I've found that I use `lapply()` more frequently than any of the other `apply` family members. A key reason is that my functions normally return multiple objects of different type (which makes lists the only sensible format)… or a single data frame (which is where bind `dplyr::bind_rows()` comes in).

Aside: quick look at sapply() Another option that would work well in the this particular case is `sapply()`, which stands for "**s**implify apply". This is essentially a wrapper around `lapply` that tries to return simplified output that matches the input type. If you feed the function a vector, it will try to return a vector, etc.

[7]For example, we could pipe the output to `unlist()` if you wanted a vector. Or you could use use `sapply()` instead, which I'll cover shortly.

```r
sapply(1:10, function(i) LETTERS[i])
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

Create and iterate over named functions

As you may have guessed already, we can split the function and the iteration (and binding) into separate steps. This is generally a good idea, since you typically create (named) functions with the goal of reusing them.

```r
## Create a named function
num_to_alpha <-
  function(i) {
  df <- tibble(num = i, let = LETTERS[i])
  return(df)
  }
```

Now, we can easily iterate over our function using different input values. For example,

```r
lapply(1:10, num_to_alpha) %>% bind_rows()
```

```
## # A tibble: 10 x 2
##      num let
##    <int> <chr>
##  1     1 A
##  2     2 B
##  3     3 C
##  4     4 D
##  5     5 E
##  6     6 F
##  7     7 G
##  8     8 H
##  9     9 I
## 10    10 J
```

Or,

```r
lapply(c(1, 5, 26, 3), num_to_alpha) %>% bind_rows()
```

```
## # A tibble: 4 x 2
##     num let
##   <dbl> <chr>
## 1     1 A
## 2     5 E
## 3    26 Z
## 4     3 C
```

Further resources

In the next two lectures, we'll dive into more advanced programming and function topics (debugging, parallel implementation, etc.). However, I hope that today has given you solid grasp of the fundamentals. I highly encourage you to start writing some of your own functions. You will be doing this a *lot* as your career progresses. Establishing an early mastery of function writing will put you on the road to awesome data science successTM. Here are some additional resources for both inspiration and reference:

- Garrett Grolemund and Hadley Wickham's *R for Data Science* book — esp. chapters 19 ("Functions)") and 21 ("Iteration)") — covers much of the same ground as we have here, with particular emphasis on the **purrr** package for iteration.
- If you're looking for an in-depth treatment, then I can highly recommend Hadley's *Advanced R* (2nd ed.) He provides a detailed yet readable overview of all the concepts that we touched on today, including more on his (and R's) philospohy regarding functional programming (see Section ||).
- If you're in the market for a more concise overview of the different `*apply()` functions, then I recommend this blog post by Neil Saunders.
- On the other end of the scale, Jenny Bryan (all hail) has created a fairly epic purrr tutorial mini-website. (Bonus: She goes into more depth about working with lists and list columns.)