Accelerated Lab 12: Joins and Data Visualization

Harris Coding Camp

Summer 2023

Today's lab has 4 parts.

1. a teaser about joins
2. ggplot pointers
3. base R plotting
4. exploring data on your own

Try to devote at least 20 minutes to exploring data on your own. When you make your own graphs you're sure to run into idiosyncratic issues where the TA assistance will be clutch.

joining together two data sets

Joining two data sets based on some shared column(s) is an important tool in the analysts toolbox. If we had one more lecture, we would cover joins. Chapter 20 of the text book covers the topic quite well!

We'll use a simple example to understand the bare bone basics

```
band_members
```

```
## # A tibble: 3 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

```
band_instruments
```

```
## # A tibble: 3 x 2
##   name  plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
```

```
band_instruments2
```

```
## # A tibble: 3 x 2
##   artist plays
##   <chr>  <chr>
## 1 John   guitar
## 2 Paul   bass
## 3 Keith  guitar
```

A useful concept is the **primary key**: column or columns that uniquely identify a row.

The primary key for both `band_members` and `band_insturments` is `name`. In `band_instruments2` the primary key is `artist`.

When we join to data sets, we need keys to match on.

```
band_members |> left_join(band_instruments, by = "name")
```

If there is no shared column name by is required – notice the syntax.[1]

```
band_members |> left_join(band_instruments2, by = c("name" = "artist"))
```

1. Run both of the `left_join`s above. What happens if you don't include the `by` term?
2. Notice that Mick has an `NA` for plays. Mick isn't in the `band_instruments` file! This leads to our second consideration, which data to keep – run the following code. Discuss with your partners and TA how these functions are similar and different.

```
band_members |> inner_join(band_instruments)
band_members |> left_join(band_instruments)
band_members |> right_join(band_instruments)
band_members |> full_join(band_instruments)
```

So far, we have joined "one-to-one" mappings between data sets.

3. What is "logically" the primary key in `origins`? Is this also a primary key in `band_members`?

```
origins <- tibble(band = c("Beatles", "Stones"),
                  origin = c("Liverpool", "London"))
```

4. Create the `origins` data set on your machine and combine it with `band_members`.[2] Notice that `band` is not a primary key in `band_members` – we're doing a "many-to-one" join.
5. Warning a "one-to-many" join come with risks. Look at the output and explain why this is a problem.

```
full_names <-
  tibble(name = c("John", "John", "Paul", "Mick"),
         surname = c("Lennon", "Bonham", "McCartney", "Jagger"))

band_members |>
  left_join(full_names)
```

[1] The lastest version of dplyr adds a helper function `join_by()` which allows you to use column names like so `join_by(name = artist)`. Check out the textbook for more information – there are neat additional features available.

[2] see helper-code-lab-12.R for copy-able code.

```
## Joining, by = "name"
```

```
## # A tibble: 4 x 3
##   name  band    surname
##   <chr> <chr>   <chr>
## 1 Mick  Stones  Jagger
## 2 John  Beatles Lennon
## 3 John  Beatles Bonham
## 4 Paul  Beatles McCartney
```

These are the *very basics* of joins. We pay attention to

a. which columns we're joining or merging on
b. whether the data are unique or not (i.e. are we doing a 1-to-1, many-to-1 or many-to-many join?)[3]
c. which data we want to keep left, right, all, or the intersection of the two.

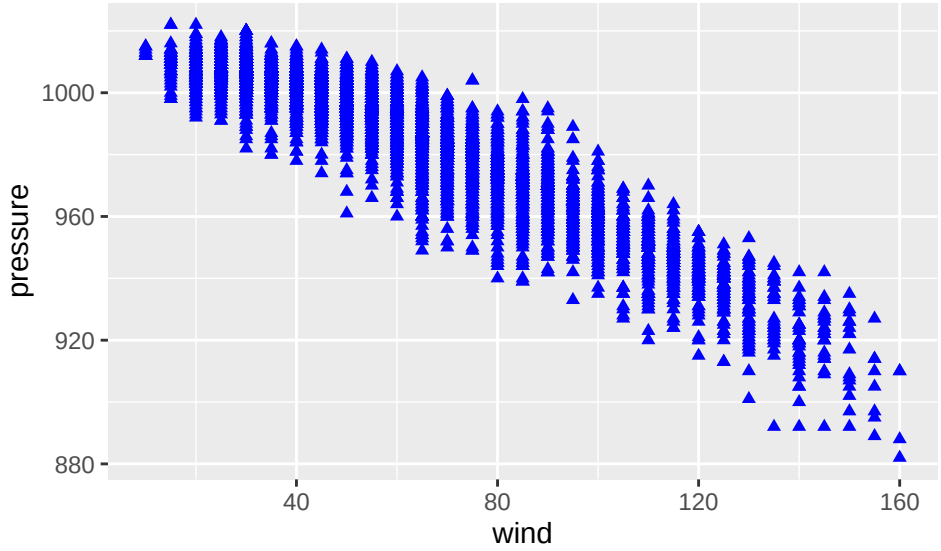
ggplot: some pointers and gotchas.

Wherein we provide some exposure to common points of confusion.

Declaring versus mapping aesthetics

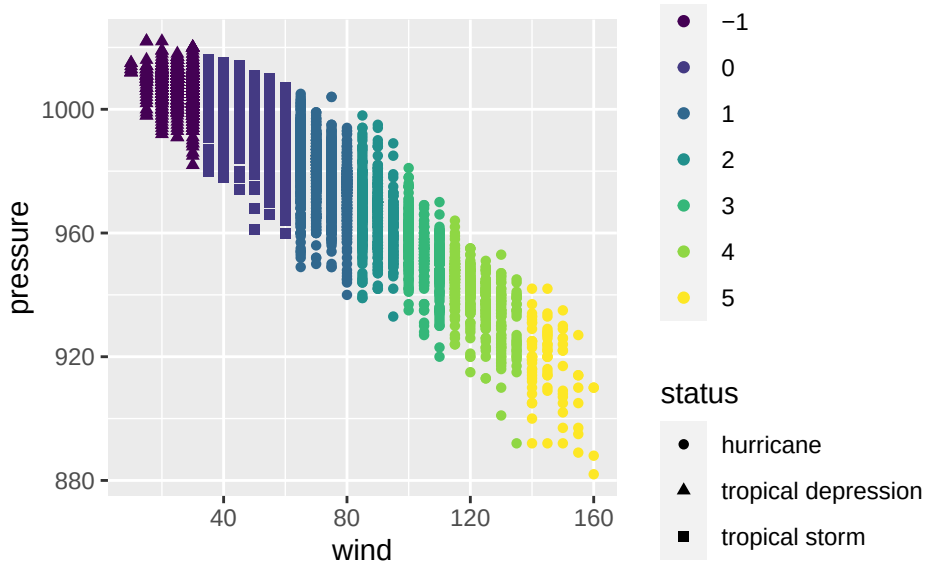
We can change the aesthetics of our plots by mapping data to aesthetics OR by declaring aesthetics.

```
#  declare that points should be red
ggplot(storms, aes(wind, pressure)) +
  # no need for a mapping = aes(...)
  geom_point(color = "blue", shape = "triangle")
```




```
ggplot(storms, aes(wind, pressure)) +
  geom_point(mapping = aes(color = category,
                           shape = status))
```

[3]Things can get even more hectic if you "many-to-many" joins. Moreover, "many-to-many" joins might create very large data sets that can cause R to break (temporarily). See bonus question at end of lab.

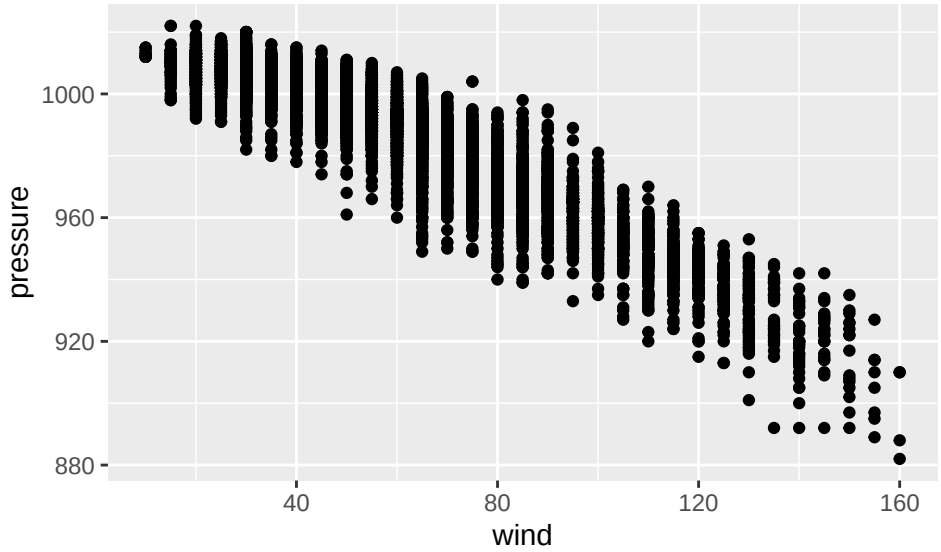


1. What happens if you mix it up? Run the following code and explain the errors or output.

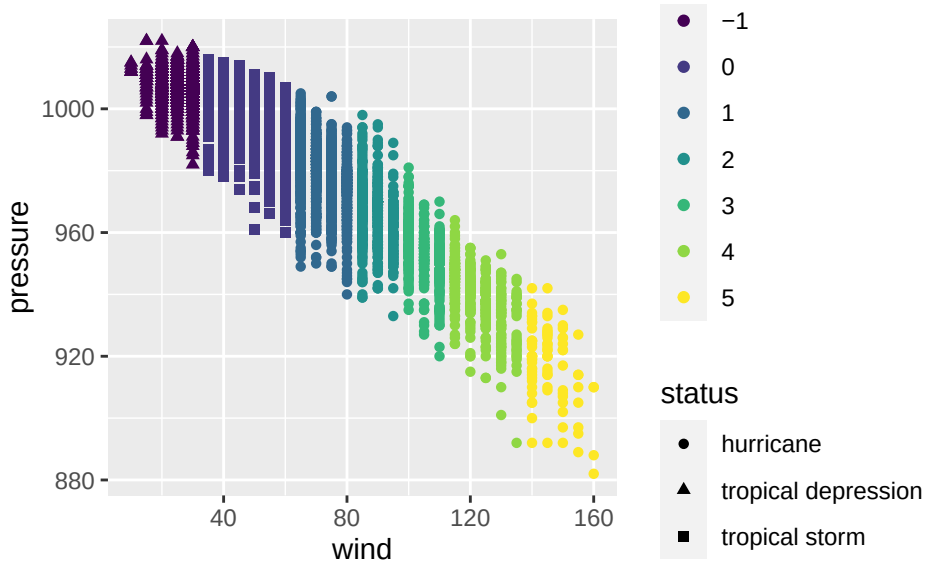
```
ggplot(storms, aes(wind, pressure)) +
  geom_point(color = category, shape = status)

ggplot(storms, aes(wind, pressure)) +
  geom_point(aes(color = "blue", shape = "triangle"))
```

2. Notice that declared aesthetics are not inherited while mappings (in `aes()`) are inherited.

```
ggplot(storms, aes(x = wind, y = pressure), color = "blue") +
  geom_point() # does not inherit declared aesthetics so points are black.
```



```
ggplot(storms, aes(wind, pressure,color = category,
                   shape = status)) +
  geom_point() # Inherits mappings from aes()
```



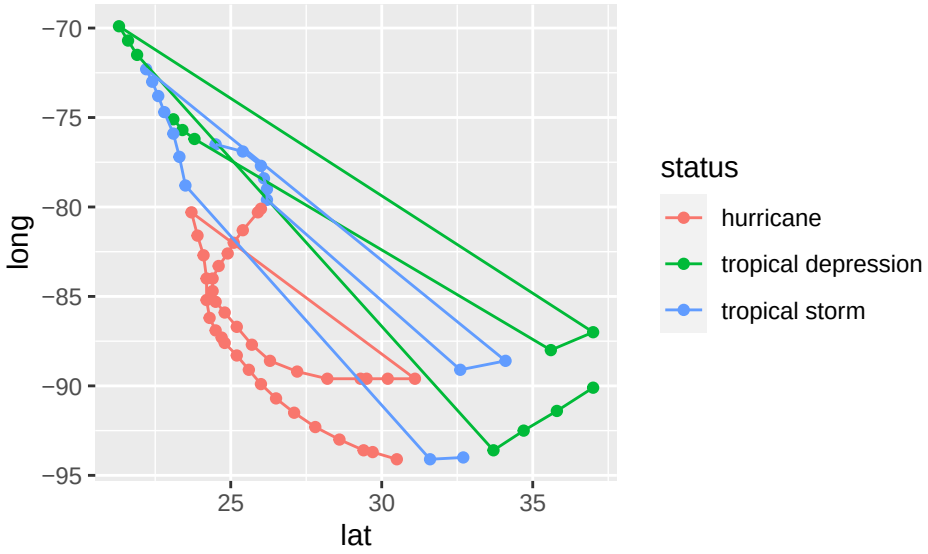
why use group?

A student asked, what is `group` for? Here's an example.

When we use an aesthetic like color, it automatically becomes the "group" for the plot. Some layers act on the groups directly, such as `geom_path()` or `geom_line()` which connect points within a group.

3. In the plot below, `geom_path()` is connecting points with the same status! But we want to see the paths of the storms. Adjust the aesthetic mapping so that we can see the paths of Rita and Katrina.

```
storms |>
  filter(name == "Rita" | (name == "Katrina" & year == 2005)) |>
  ggplot(aes(x = lat, y = long, color = status)) +
    geom_point() +
    geom_path()
```

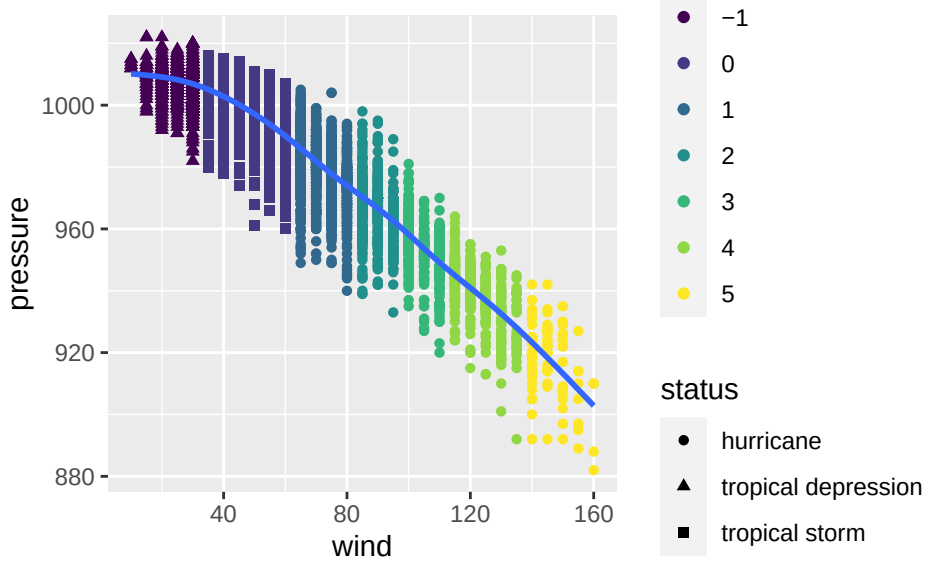


line of best fit

When you run the following code you see a message " . . . using method = `gam` . . . " which is a default chosen based on your data.

```
ggplot(storms, aes(wind, pressure)) +
  # here we map color and shape within geom_point b/c we don't want
  # those groups when we fit our model.
  geom_point(aes(color = category, shape = status)) +
  geom_smooth(method = "gam", se = FALSE)
```

```
## `geom_smooth()` using formula 'y ~ s(x, bs = "cs")'
```



4. If you want to learn the slope and intercepts of a "line of best fit" or "linear model", we use the lm function. Tell `geom_smooth` to use a straight line instead of the default "gam".

5. Notice we do the aesthetic mapping of color and shape within `geom_point()` instead of `ggplot()`. See

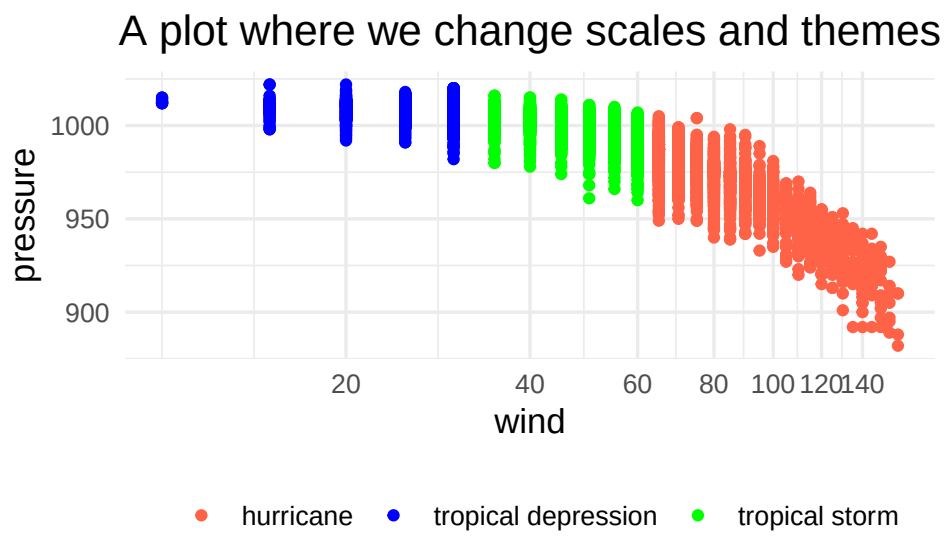
what happens if you do all the aesthetic mappings within `ggplot()` and explain to your partner what the error means.

changing colors and other scales

Most of the time the default scales for x and y, colors and other aesthetics are fine. But if you want to change them, there are a bevy of functions called `scale_<aesthetic>_<type>`. The overarching idea is that the "scale" can be adjusted or even transformed.

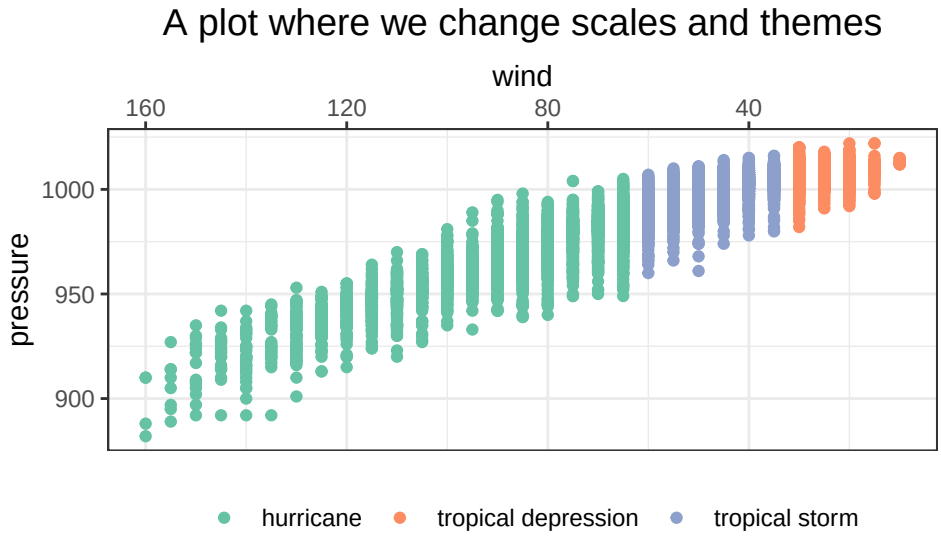
Here is an example where we change scales in probably not so useful ways. For good measure, we also adjust the theme in various ways. `theme_minimal()` makes a number of pre-set adjustments to how the plot is displayed.[4]

```
base_plot <- storms |>
  ggplot(aes(wind, pressure)) +
  geom_point(aes(color = status)) +
  labs(title = "A plot where we change scales and themes",
       color = "")

base_plot +
  scale_color_manual(values = c("tomato", "blue", "green")) +
  scale_y_continuous(breaks = c(900, 950, 1000)) +
  scale_x_log10(breaks = 1:7 * 20) +
  theme_minimal(base_size = 13) +
  theme(legend.position = "bottom",
        plot.title = element_text(hjust = .5))
```



6. Play with the code to make sure you understand what it's doing. Then, adjust the code to make the resulting plot. **See how close you can get before reading the explicit instructions (below).** Use `?` and google! (Don't worry about picking the right colors though.)

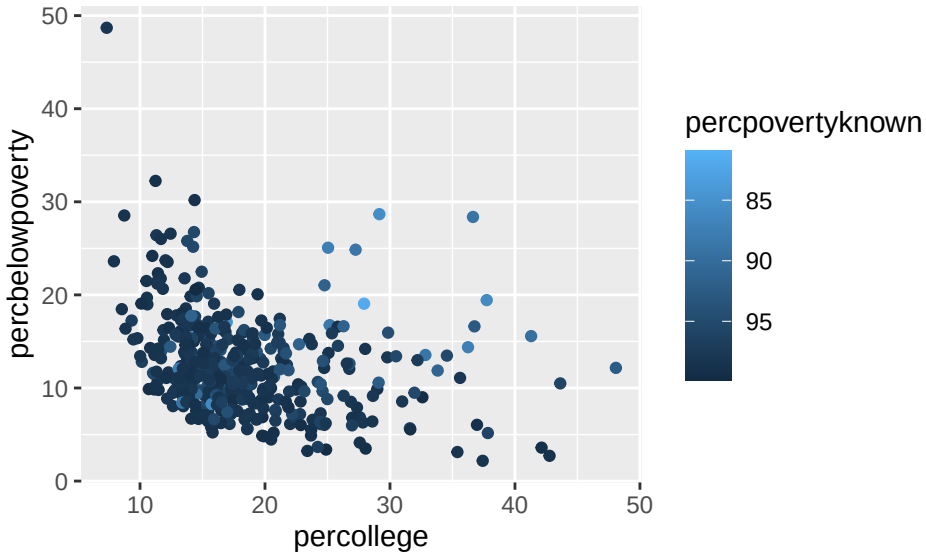
[4] There are a handful of built-in themes. You'll find the whole list using `?theme_minimal`.



a. Adjust the x axis scale so that it's reversed, has breaks at multiples of 40 and is located at the top of the plot.
b. Use `scale_color_brewer(palette = "Set2")` to pick the colors.
c. Remove the *label*[5] "status" from the legend.
d. Change the `base_size` of the text back to the default.
e. Make the plot.title `size = 14`.
f. Change the theme to `theme_bw()`.

7. In your own words, what's the difference between scale and theme?

8. Above we used `scale_x_reverse()` which is a shortcut for `scale_x_continuous(trans = "reverse")`. With that in mind, can you replicate this plot where we reverse the order of the color gradient here?



[5]Hint: You could probably do this using a scale function, but there's a more straight forward way using `labs()`.

base R plotting

There are several functions used to make plots in base R. Common ones include `plot()` for scatter plots and line charts,[6] `hist()` for histograms, `barplot()` for bar graphs, and `boxplot()` for boxplots. In practice, most graphs you make with `ggplot` can be built with base R plotting. However, the cohesion of the tidyverse, the support resources and the open-source community add-ons make `ggplot2` a go-to.[7]

In this lab, we'll provide a brief introduction to base R plotting and highlight sometimes where it is nicer to use relative to ggplot.

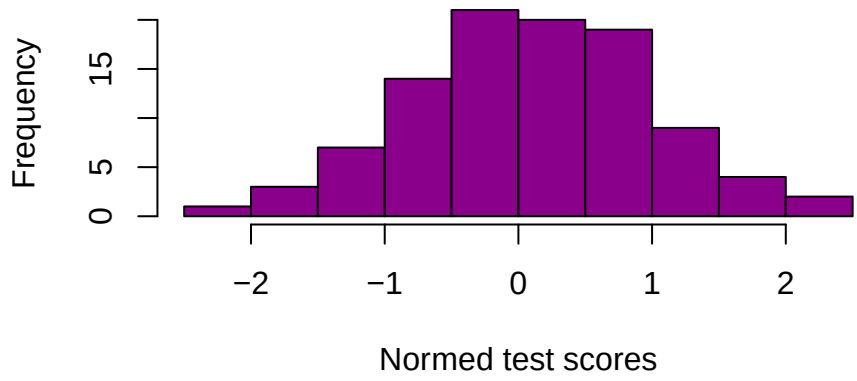
I will make plots in base R, your job is to produce a similar plot with ggplot.

1. Notice how the two methods have different defaults. Switching all the defaults can be tricky. We'll highlight important things to match. Try to match anything coded explicitly.

a. Here's a histogram produced in base R. Bin widths matter.

Tips: Recall that in `ggplot` the color of a bar can be adjusted with `fill` for the "inside" of the bar, while `color` adjusts the outline or "stroke" of the bar.

```
# create data
set.seed(1) # set.seed to get the same data as us!
data <- tibble(normed_test_scores = rnorm(100))

# make a histogram
hist(data$normed_test_scores, # takes a vector of data
     xlab = "Normed test scores",
     main = "", # What is main? try changing this and see what happens.
     col = "darkmagenta")
```

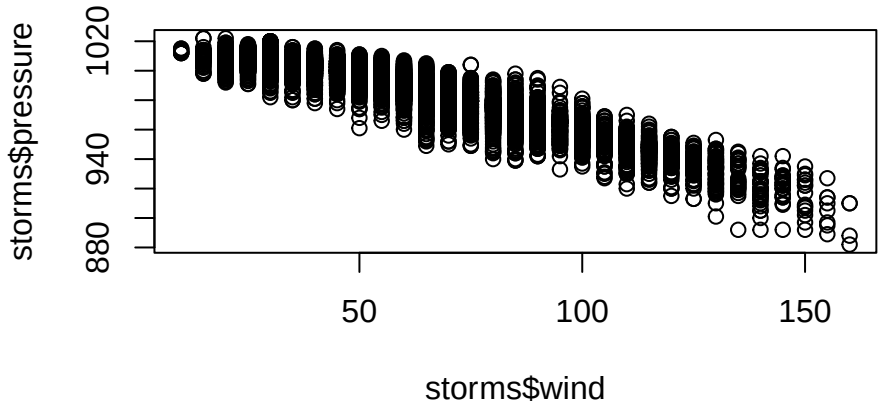


[6] `plot()` is the most flexible base R plotting function. Some packages have added custom behavior for `plot()`–which makes `plot()` seem like magic sometimes. In the background, the code checks for specific `class()` of input (e.g. is it a tibble or a geographic data (called `sf`) with lat/lon) and will call the specific versions of `plot()` based on the data type. This is beyond the scope of coding lab.

[7] For a contary POV, this person makes (an arguement in favor of base R)[https://flowingdata.com/2016/03/22/comparing-ggplot2-and-r-base-graphics/]

2. The `plot()` function makes "Generic X-Y plots"

```
# the most basic plot syntax is plot(x, y)
# where x and y are vectors of the same length
plot(storms$wind, storms$pressure)
```



If you want to avoid typing the name of the data twice we can use the following syntax

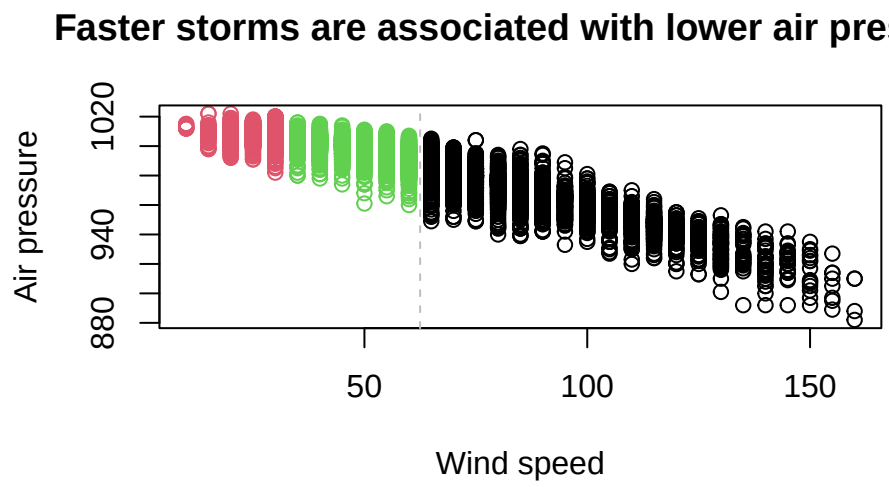
- Here we can refer to columns using "formula" objects `y ~ x`. You'll see formulas often when doing statistics and they pop up at times when there's a relationship between columns.[8]

```
plot(y ~ x,                    # plot y by x
     data = [df_name],         # data from [df_name]
     type = "[type]",          # type (points "p", lines "l", ...)
     ylab = "[y variable]",    # add y-axis label
     xlab = "[x variable]",    # add x-axis label
     main = "[title]"          # add title
     )
```

Here's a plot call in action.

```
plot(pressure~wind,
     data = storms,
     # type = "p", # this is the default plot type
     col = as.factor(storms$status),
     xlab = "Wind speed",
     ylab = "Air pressure",
     main = "Faster storms are associated with lower air pressure")

# let's also add a line
# (in a Rmd this whole chunck needs to be run simultaneously)
# (in the console running abline() after plot() will add the line to the plot)
abline(v = 62.5, col = "grey", lty = 2)
```

[8] formulas are also a convenient way to capture columns by name when the name isn't in the global environment.



Replicate the above plot.

- Approximate the colors.
- We can get an open circle by setting `shape = 1`.
- The vertical line uses `linetype = "dashed"`
- Don't worry about theme material, unless you want an added challenge.

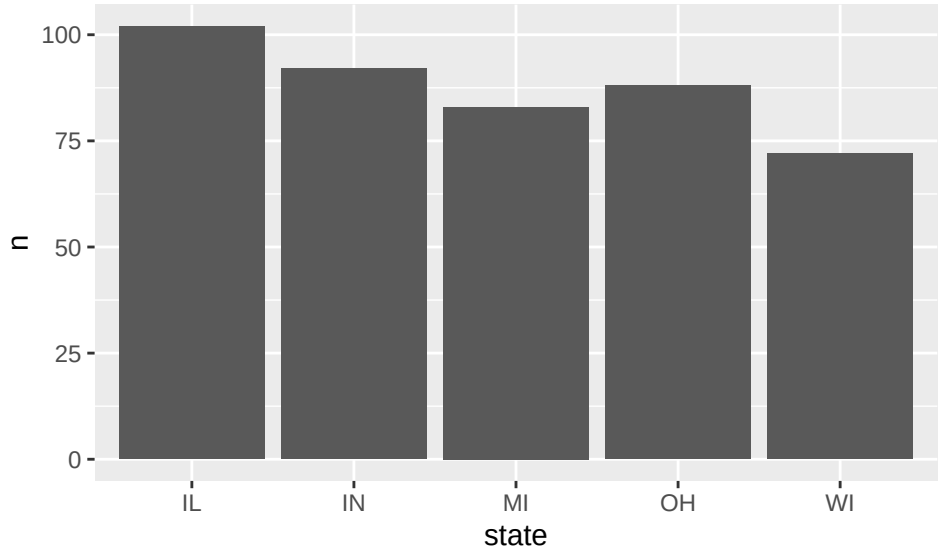
3. Bar graphs use the `barplot()` function. The function expects a table of bar heights.

```
# we can count the number of times a string shows up in a character vector
# using table.
(my_table <- table(midwest$state))
```

```
##
##  IL  IN  MI  OH  WI
## 102  92  83  88  72
```

a. Is the input to table a vector or a data frame?
b. Is the output a vector, a data frame or something distinct?
c. Now try `barplot(my_table)`.
d. Make a similar bar graph using `ggplot()` ignore aesthetic differences. Notice that your `geom_bar()` call calculates the number of counties per state automatically!
e. We can also feed `geom_bar()` counts using `geom_bar(stat = "identity")`. Run the code below. What does `count()` do? What does `geom_col()` do?

```
midwest |>
  count(state) |>
  ggplot(aes(x = state, y = n)) +
  geom_bar(stat = "identity") # instead try: geom_col()
```



4. Take a moment to look over the base R plotting slides.

Final project

Now's a good time to go back to the data you loaded for your final project. Start interrogating your data by making simple plots. At this point in the process you will generally rely on default formatting. The goal is to sharpen your questions by exploring the relationships in your data.

Joins Bonus question

```
a <- tibble(time = rep(1:5, each = 2),
            name = rep(c("a", "b"), 5),
            x = rnorm(10))
b <- tibble(obs_time = 1:5,
            name = rep(c("a"), 5),
            y = rnorm(5))

a
```

```
## # A tibble: 10 x 3
##     time name        x
##    <int> <chr>   <dbl>
##  1     1 a     -0.620
##  2     1 b      0.0421
##  3     2 a     -0.911
##  4     2 b      0.158
##  5     3 a     -0.655
##  6     3 b      1.77
##  7     4 a      0.717
##  8     4 b      0.910
##  9     5 a      0.384
## 10     5 b      1.68
```

```
b
```

```
## # A tibble: 5 x 3
##   obs_time name       y
##      <int> <chr>  <dbl>
## 1        1 a     -0.636
## 2        2 a     -0.462
## 3        3 a      1.43
## 4        4 a     -0.651
## 5        5 a     -0.207
```

How many rows will the resulting data set have? Once you think you know. Try running the code.

```
left_join(a, b)
left_join(a, b, by = c("time" = "obs_time", "name" = "name"))
```

Appendix:

Color

1. "What does this mean?" you might ask. ...

```
RColorBrewer::brewer.pal(3, "Set2")
```

```
## [1] "#66C2A5" "#FC8D62" "#8DA0CB"
```

We can refer to colors by name or by hexidecimal code (e.g. "#8DA0CB"). (r-charts.com/colors)[https://r-charts.com/colors/] has a mapping from color names to hexidecimal code.[9] Each color is determined by the amount of red, green and blue (e.g. green is c(0, 255, 0) or "#00ff00" – no red [0 out of 255], all green [255 out of 255], no blue [0 out of 255]).

a. What Hexidecimal should return blue?
b. To figure out `tomato` you'll have to visit the website linked above.
c. Replace the color names with hexidecimal to make sure you have the right numbers.

```
base_plot +
  scale_color_manual(values = c("tomato", "blue", "green"))
```

[9] `colors()` is a function that will tell you the name of the built-in colors.