R Visualization and Data Manipulation

RICE

Fondren Library
Digital Scholarship Services

ABOUT THE TIDYVERSE

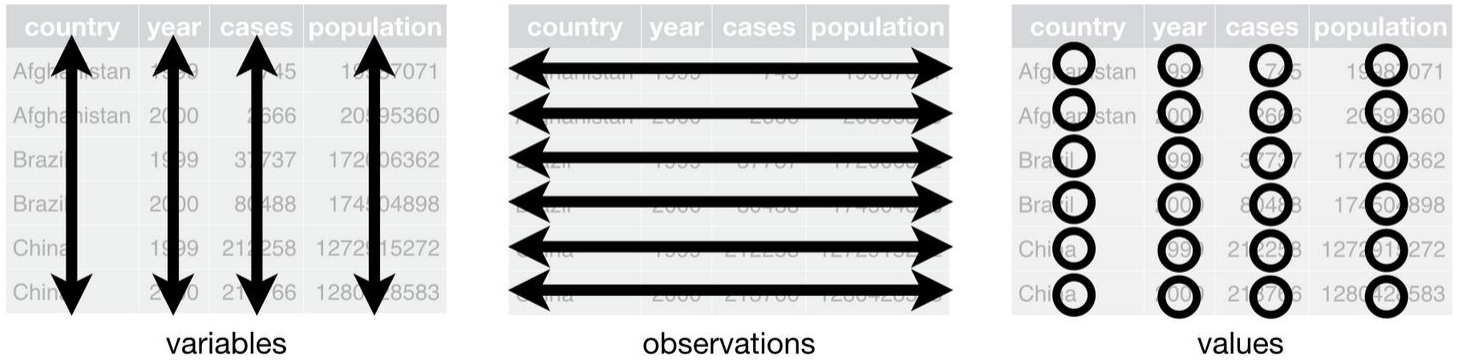
Simple and attractive packages

WHAT IS THE TIDYVERSE?

- Hadley Wickham's brainchild
- The most important and influential development in R in the past decade
- An ecosystem of packages that hold a similar philosophy about tidy data

SO, WHAT IS TIDY DATA?

- Two rules:
  - Row is an observation
  - Column is a variable



EXAMPLES OF UNTIDY DATA

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

TIDYING UP

- `spread`
  - If there are two columns worth of data in one column
  - `spread(table, TYPE OF OBSERVATION, VALUE)`
- `gather`
  - If one column worth of data is spread across many columns
  - `gather(table, firstcol, secondcol, columnname1, columnname2)`

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

key value

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

spread(table2, type, count)

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

key value

gather(table4a, `1999`, `2000`, key = "year", value = "cases")

DATA STORAGE WITH TIBBLES

A modernized approach to data frames

WHY TIBBLES?

- Fixes many annoyances with base R
  - stringsAsFactors
  - Partial column name matching
  - Unhelpful printing
- Very easy to use with all the packages we are going to talk about today!

CONVERTING TO TIBBLES

- Convert an existing `data.frame` to a `tibble`:
  - `as_tibble(df)`
- Convert an existing `tibble` to a `data.frame`:
  - `as.data.frame(tbl)`
- Typical asymmetry
  - Base R uses periods a lot
  - Tidyverse prefers underscores

MAKING TIBBLES

- Column approach
  - `tibble(colname1=c(…), colname2=c(…))`
- Row approach
  -
```
tribble(
   ~colname1,      ~colname2,
   val1,           val2,
   val3,           val4,
)
```

ACCESSING TIBBLES

- Columns:
  - `tbl$colname`
  - `select(tbl, colnames)`: returns a tibble
  - `pull(tbl, colname)`: returns a vector
- Rows:
  - `tbl[index, ]`
  - `slice(tbl, index)`
- Note: the first options are inherited from base R, the other options are from the `dplyr` package of the tidyverse

DATA ACQUISITION WITH READR

Fast, simple, and instantly in the tidyverse!

READING DATA, TIDY-STYLE!

- `read_csv("filename")`
  - Also, csv2 (;), tsv (tabs), and more
- `read_fwf("filename", fwf_widths(c(3, 5, …)))`
- Other libraries for reading xlsx, json, xml and more!
  - readxl, tidyjson, XML
- Base R: `read.csv`, `read.fwf`, etc.
  - Much slower!
  - `data.frame`, not `tibble`

VECTOR MANIPULATION WITH PURRR AND FURRR

Functional and expressive operations

THE FUNCTIONAL APPROACH

- Focus is on the operation, not iteration
- More concise
- purrr vs. for loops:
  - purrr is extremely easy to parallelize, much cleaner and more expressive

MAP

- Apply an operation to everything in a vector (column)
- Function can be passed as either:
  - Any function that takes one argument
    - e.g. `tolower(x)`, which converts a string of characters to lowercase
  - Any function can be passed as a formula
    - Lambda / Anonymous function
    - Start with ~ to indicate formula
    - `.x` is the name of the parameter of the function
    - `~ 3 + 5 * .x`
- `map(a, ~ 3 + 5 * .x)` will take each value in a and multiply it by 5, then add 3!

MAP2

- Apply an operation to everything in two vectors (column)
- Function can be passed as either:
  - Any function that takes two arguments
    - e.g. `max(x, y)`, returns the larger
  - Any function passed as a formula
    - Lambda / Anonymous function
    - Start with ~ to indicate formula
    - `.x and .y`
    - `~ 3 * .y + 5 * .x`
- `map2(firstname, lastname, ~ paste(.x, .y))` will concatenate corresponding strings!

DELETE AND KEEP

- Only get the data you really want
- Formulas again!
  - `delete(gpas, ~ .x < 2.0)`
  - `keep(gpas, ~ .x > 3.8)`

EVERY AND SOME

- Summarize a list
  - Is a condition always true?
  - Is a condition ever true?
- What do you know? Formulas again!
  - `every(gpas, ~ .x < 2.0)`
  - `some(gpas, ~ .x > 3.8)`

MODIFY IF

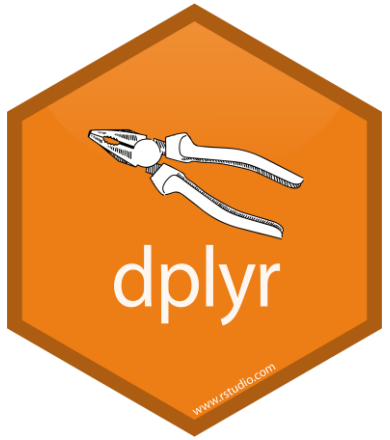
- Only change certain values
- `modify_if(vector, condition, operation)`
  - `modify_if(stateareas, ~ .x > 50000, ~ .x / 2)`

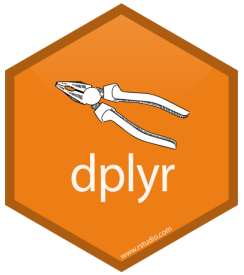
EASY PARALLELIZATION

- Run many tasks at the same time
- Considerable speedup!
- furrr: leverage parallel computation for purr
  - Add `plan(multiprocess)` to your script
  - Just use `future_map` over `map`, and similarly for other functions!
- e.g.:
  - `future_map(names, tolower)`

DATA MANIPULATION WITH DPLYR

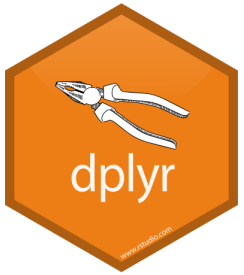
A vocabulary of common actions





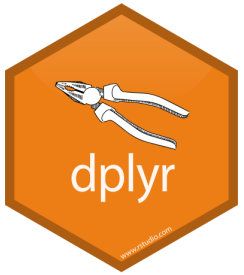
FILTER

- Subsetting a data frame
  - `filter(tbl, condition)`
  - Use column names as-is
    - `filter(states, Area > 50000)`
  - Can use multiple column names!
- `drop_na()`: a special case for deleting any row that has a missing value



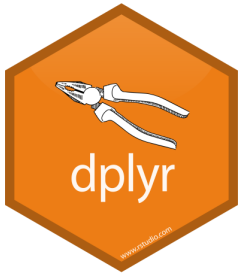
MUTATE AND TRANSMUTE

- Create new columns
  - Similar to purrr's mapping functions
- `mutate`: add column
- `transmute`: delete everything but column
- `mutate(df, newcol = oldcol * 2)`



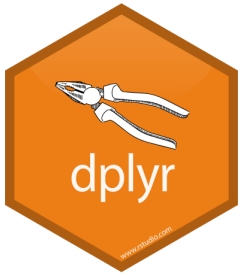
USING PURRR WITH TIBBLES

- purrr can be extremely useful when you're dealing with data tables as well!
  - `statedata$Area <- modify_if(statedata$Area, ~ .x > 50000, ~ .x + 20)`



ARRANGE

- Sort rows by some criterion
  - Specify as column name
- `arrange(states, Area)`
- Use `desc()` to switch order!
  - `arrange(states, desc(Area))`



JOIN / MERGE

- Combine data from two different tibbles!
  - Must share one column name. Specify this as the “by” argument.
  - Inner: keep only rows that are in both
  - Left: keep all rows in first tibble, even if they don’t have a match
  - Right: keep all rows in second tibble, even if they don’t have a match
  - Full: keep all rows
  - Note: if there is no match, `NA` is the default!
- `inner_join, left_join, right_join, outer_join`
- `*_join(tbl_1, tbl_2, by=“common column name”)`

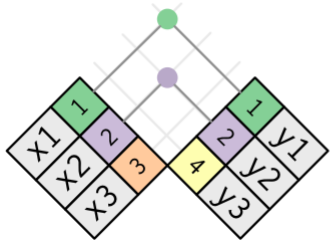
JOIN / MERGE: VISUALLY

x

1	x1
2	x2
3	x3

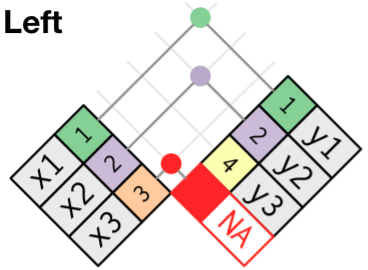
y

1	y1
2	y2
4	y3



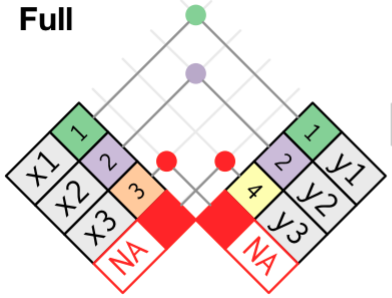
key	val_x	val_y
1	x1	y1
2	x2	y2

Left



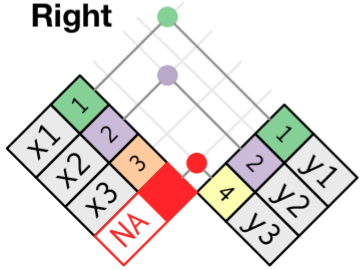
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

Full

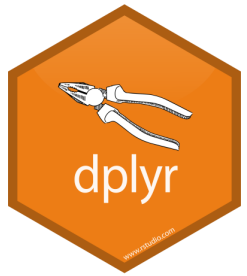


key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

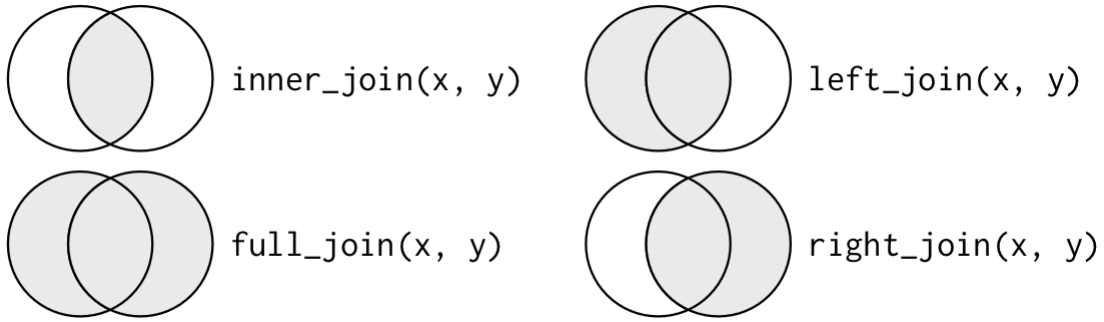
Right

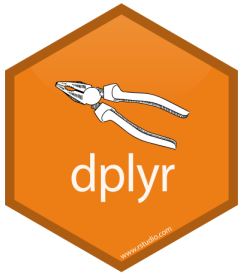


key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3



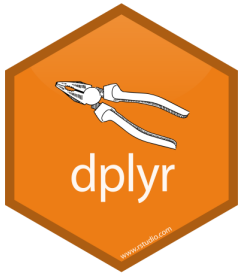
JOIN / MERGE: VENN DIAGRAMS





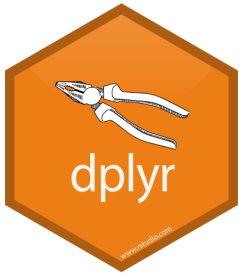
GROUP BY

- Group by a categorical variable
  - Won't immediately change the tibble
- `group_by(students, major)`
- Use `count()` to make sure the `group_by` worked
- Then, add a different summary function!



SUMMARIZE

- Calculate a measurement of a vector
  - mean, median, range, etc
  - `summarize(tbl, colname=mean(column))`
- `summarize_all(tbl, summary)`: apply to all columns
  - `summarize_all(states, mean)`
- Common grouping operations:
  - `n_distinct` -> number of unique rows
  - `sum`
  - `min, max`

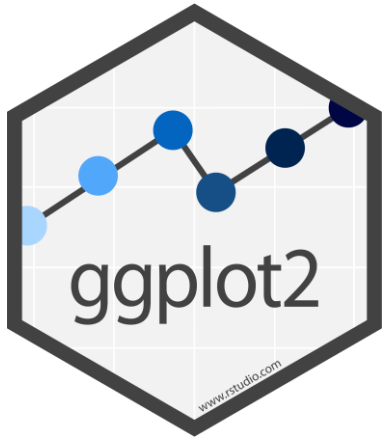


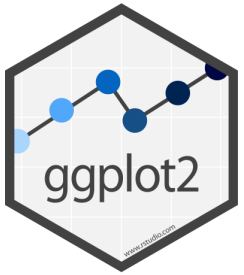
THE PIPE

- Easily combine many operations
  - `%>%` at the end of a line (not the start of the next one!)
  - `magrittr` package
  - Chains operations together by passing along the first argument
- Example:
  - ```
    states %>%
      filter(Area > 50000) %>%
      mutate(density = Population / Area) %>%
      summarize(avg=mean(density))
    ```
  - Instead of
  - ```
    summarize(mutate(filter(states, Area > 50000),
    density=Population / Area), avg = mean(density))
    ```

VISUALIZING DATA WITH GGPLOT2

Versatile and customizable plotting



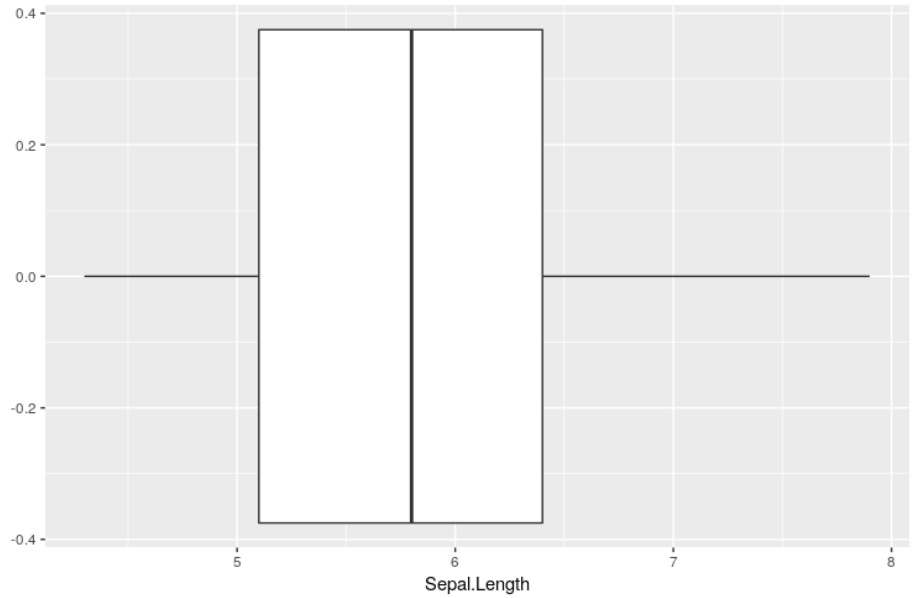


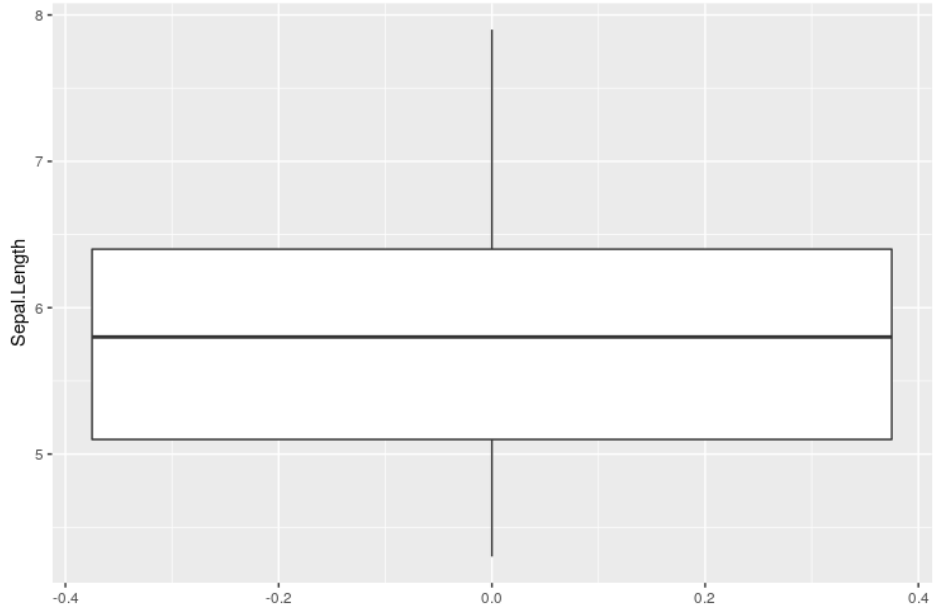
THE LAYERED PLOTTING MODEL

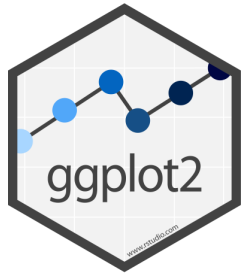
- Add aspects of the plot, one-by-one
  - Most plotting programs make you decide on everything at once
- Coordinate system, type of plot, labels, facets….

COORDINATES

- Created when you call `ggplot(data = …)`
- Changed by `coord_flip()` and `coord_polar()`, for example



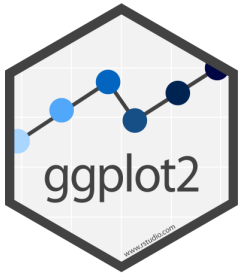




MAPPINGS

- How to tell `ggplot` what you want to use your data as
- Usually, `x = …,` and `y = …, color = …`

- `ggplot(data = iris, mapping = aes(x = Petal.Length, y = Petal.Width, color = Species))`
  - This means we've assigned the x value the lengths of the irises' petals, the y value the widths of the irises' petals, and the color to the species.
  - Note this doesn't yet make a plot! We have to specify what to do with our mapping.



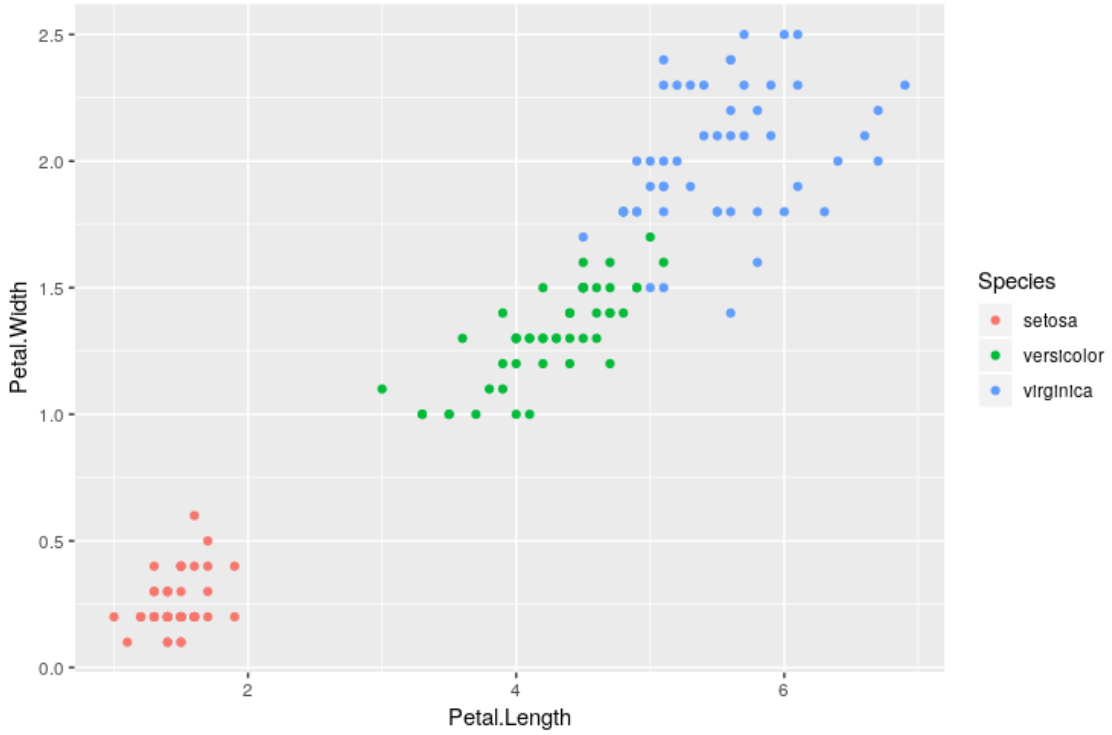
PLOTS (GEOMS)

- All plotting functions in `ggplot` start with `geom_`, to indicate they contain some geometric information in them
- Examples: scatterplots (`geom_point`), boxplots (`geom_boxplot`), histograms (`geom_histogram`)
- e.g.
  - ```
    ggplot(data = iris) +
      geom_point(mapping = aes(x = Petal.Length, y = Petal.Width))
    ```
    ```

PLOTS (GEOMS): SCATTERPLOTS

- Scatterplots are made with the `geom_point()` function
- Needs aesthetic mapping for `x` and `y`. Can use `color` as well!

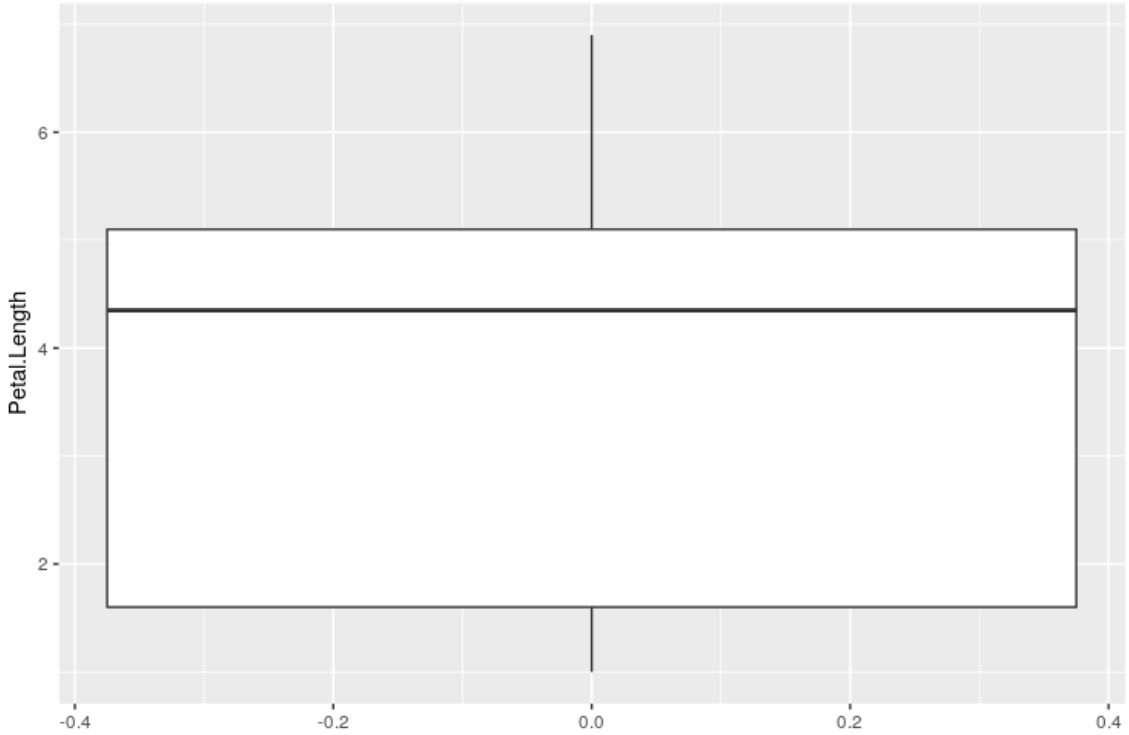
```
ggplot(data = iris) +
  geom_point(mapping = aes(
      x = Petal.Length,
      y = Petal.Width,
      color = Species))
```



PLOTS (GEOMS): BOXPLOTS

- Boxplots are made with the `geom_boxplot()` function
- Only needs `y` mapping.

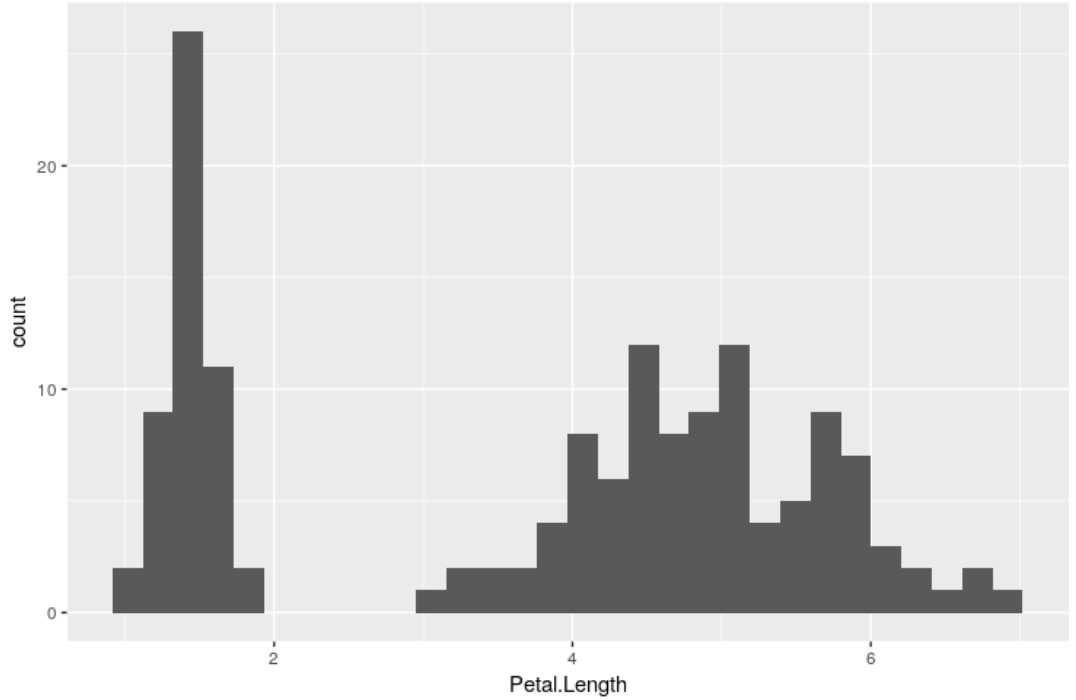
```
ggplot(data = iris) +
  geom_boxplot(
    mapping = aes(
      y = Petal.Length
    )
)
```



PLOTS (GEOMS): HISTOGRAMS

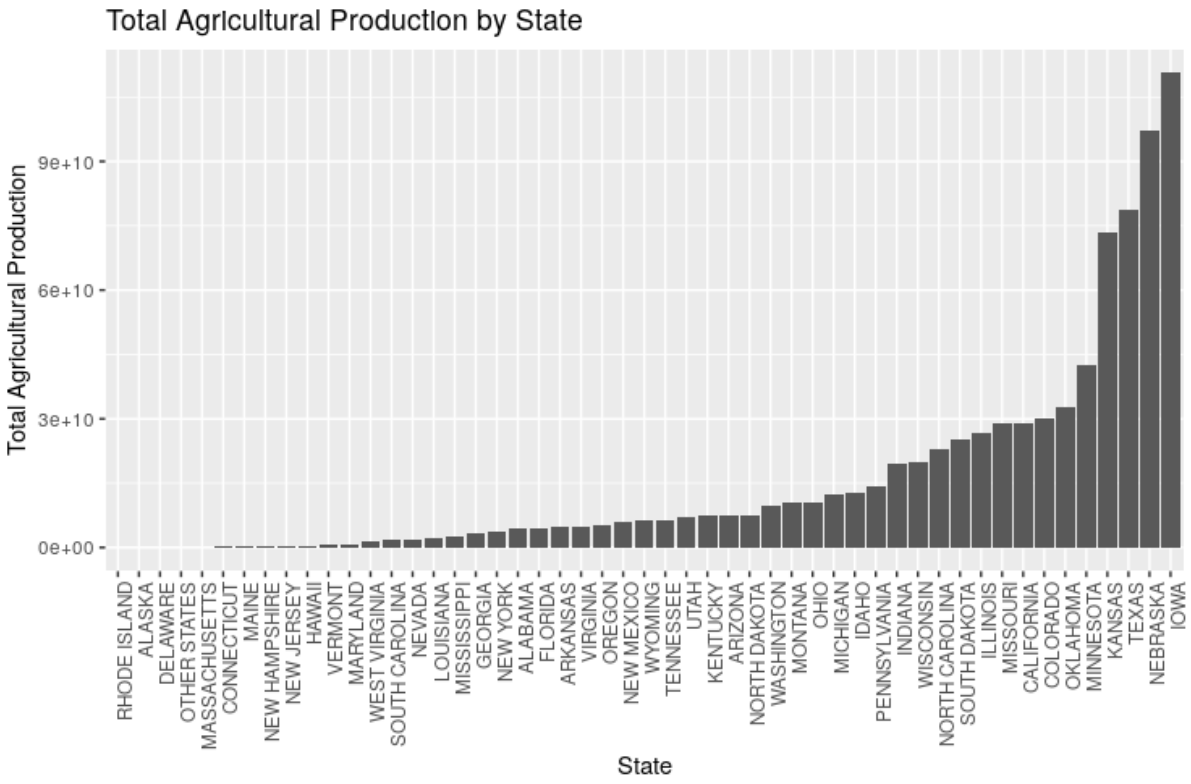
- Histograms are made with the `geom_histogram()` function
- Only needs `x` mapping.

```
ggplot(data = iris) +
  geom_histogram(
    mapping = aes(
      x = Petal.Length
  )
)
```



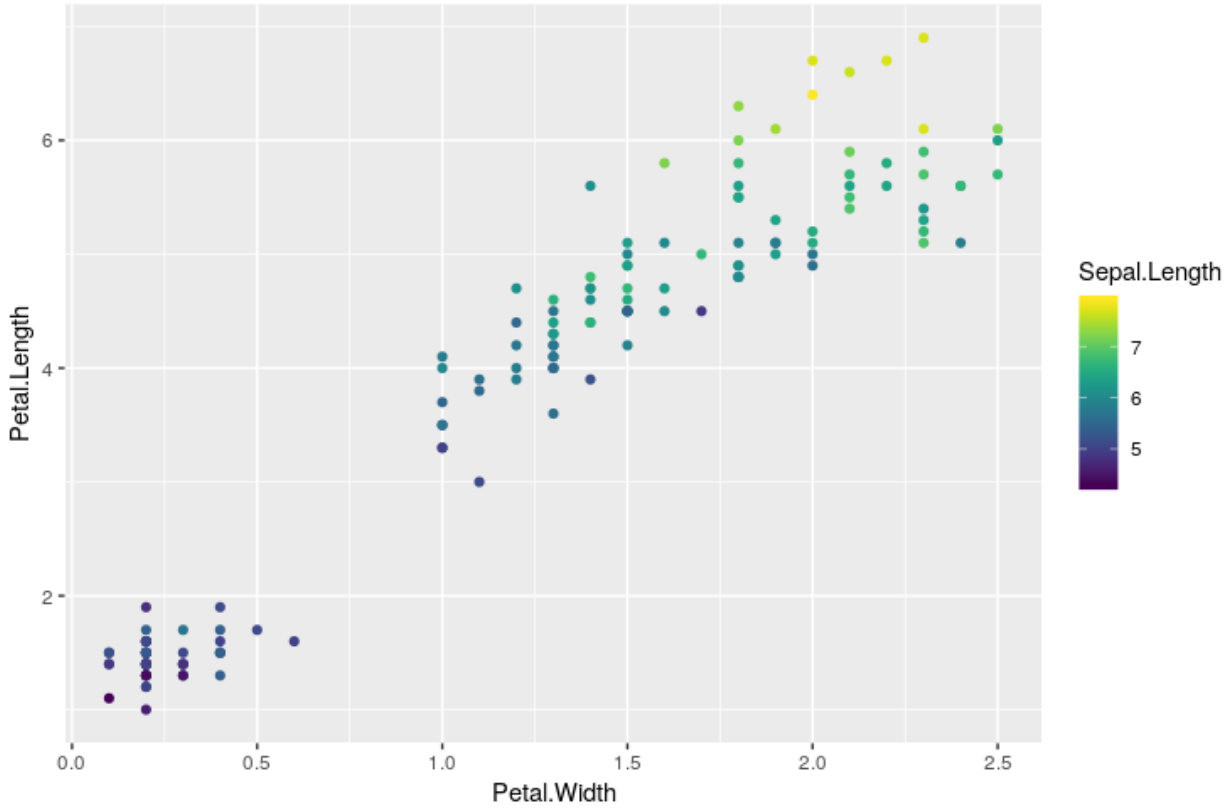
PLOTS (GEOMS): BARCHARTS

- Bar charts have categorical data on the x-axis, and some value on the y-axis.
- Use `geom_col()` with appropriate `aes` mapping
- For discrete data, especially for bar charts, you might want to rotate your axis labels!
  - `theme(axis.text.x = element_text(angle = 90, hjust = 1))`



COLORS

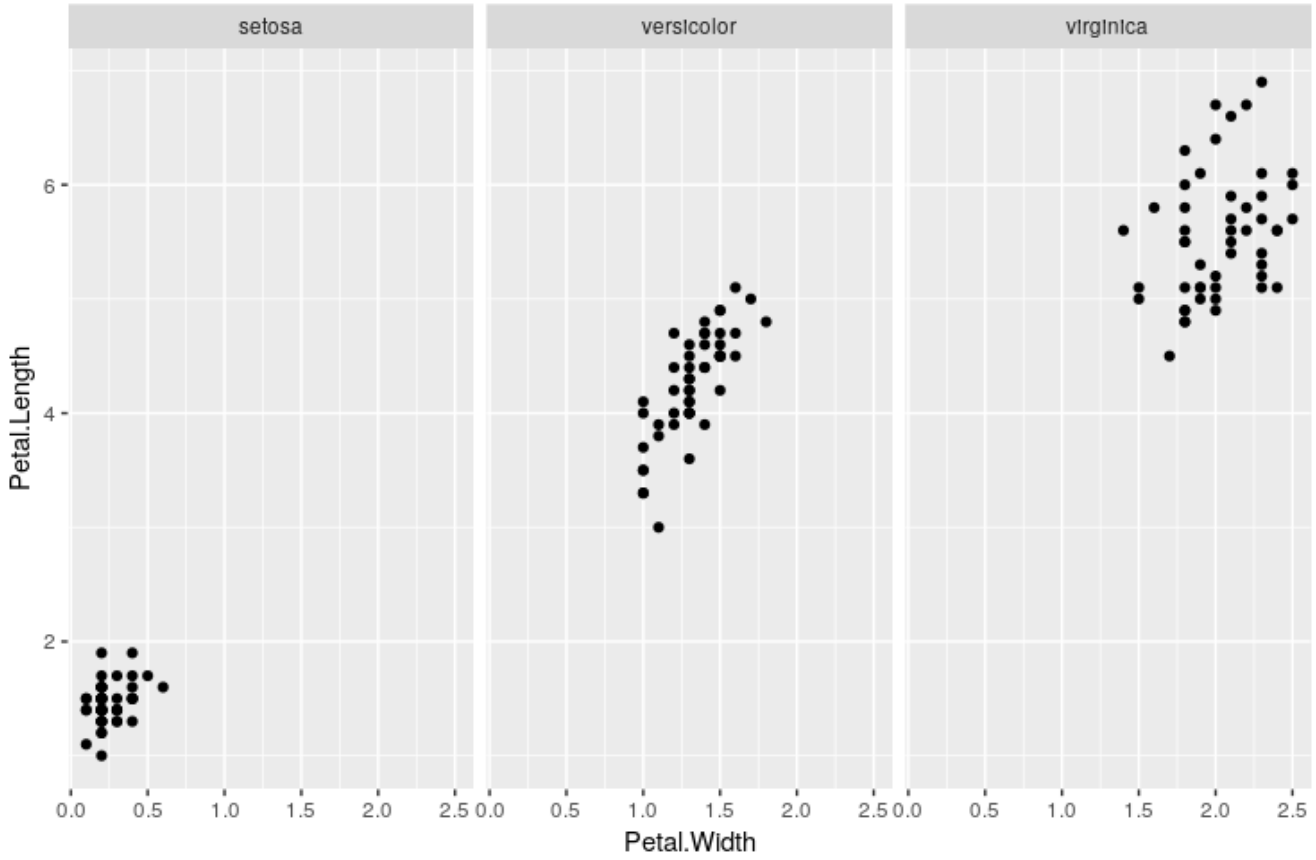
- You can color points two ways to get a ‘third dimension’ on your plots:
  - Continuous: `color = column`, then add a color scale with `scale_color_viridis()` or something similar
  - Discrete: `color = column` in `aes()`



FACETS

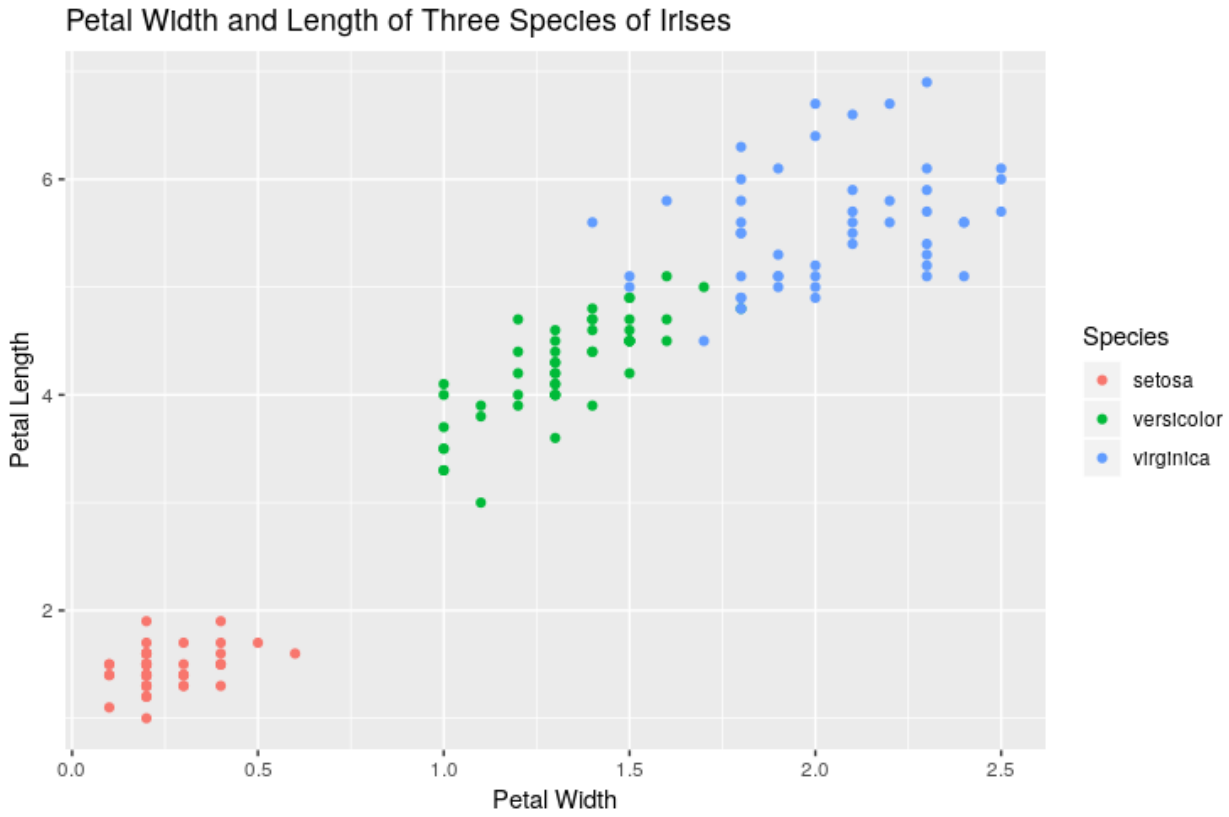
- Instead of coloring third variables, you can also make grids of plots for each categorical group of the data
- Needs a formula! Specify the column with a ~ before it
- `facet_wrap(~ colname)` or

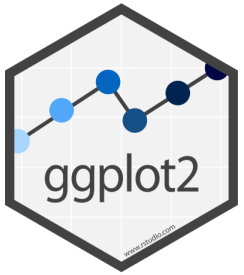
  `facet_grid(~ colname)`



LABELING

- `labs(title="", x="", y="")`
- Add axis titles:
  - `x="title"` and `y="title"`
- Main title:
  - `title = "title"`
  - `subtitle = "title"`





PUTTING IT TOGETHER

- `ggplot` uses `+` to link several layers
- Just like the pipe, you can combine tons of these together!

ALL TOGETHER NOW

Integrating all the components

A FULL TIDYVERSE WORKFLOW

- You can combine %>% and + to get long, but very readable chains of commands!
- However, don't just rely on these alone.
- If you have too many commands at once, it becomes easy to lose track.
  - Add intermediate variables with *meaningful* names!

ACKNOWLEDGEMENTS

- Much of this tutorial is based on Hadley Wickham's Amazing *R For Data Science*, available for free at https://r4ds.had.co.nz
- The tidyverse hex stickers are used under the terms of the Creative Commons 1.0 Universal license.