Data Visualization and Management

UW Tacoma

Charles Lanfear

Apr 18, 2020

Updated: Apr 12, 2020

Overview

1. Visualizing Data
2. Summarizing Data
3. Tidying Data
4. Joining Data
5. Resources for Further Learning

ggplot2


WELCOME TO GG PLOT2 Unit
PROPERTY
ggplot2
R
tidyverse
FEAT. MARKOV CHAIN MC
Wicked Wickham

Setup

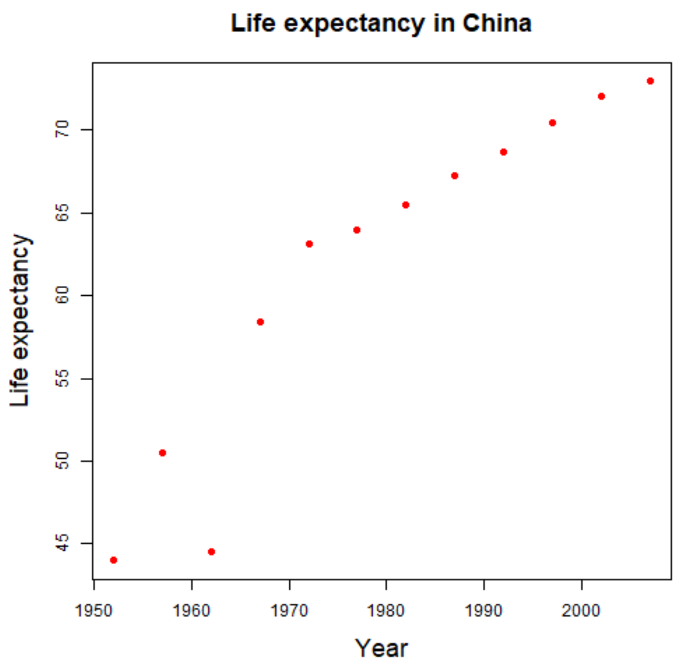
To give us something to visualize, we'll load the `gapminder` data from the last unit. We'll also load `dplyr` to give us tools to manipulate it for visualization--and later, to summarize, tidy, and join data.

```
library(gapminder)
library(dplyr)
China <- gapminder %>%
  filter(country == "China")
head(China, 4)
```

```
## # A tibble: 4 x 6
##   country continent  year lifeExp       pop gdpPercap
##   <fct>   <fct>     <int>   <dbl>     <int>     <dbl>
## 1 China   Asia       1952    44   556263527      400.
## 2 China   Asia       1957    50.5 637408000      576.
## 3 China   Asia       1962    44.5 665770000      488.
## 4 China   Asia       1967    58.4 754550000      613.
```

Base R Plots

```
plot(lifeExp ~ year,
     data = China,
     xlab = "Year",
     ylab = "Life expectancy",
     main = "Life expectancy in China",
     col = "red",
     cex.lab = 1.5,
     cex.main= 1.5,
     pch = 16)
```



`ggplot2`

An alternative way of plotting many prefer (myself included)[1] uses the `ggplot2` package in R, which is part of the `tidyverse`.

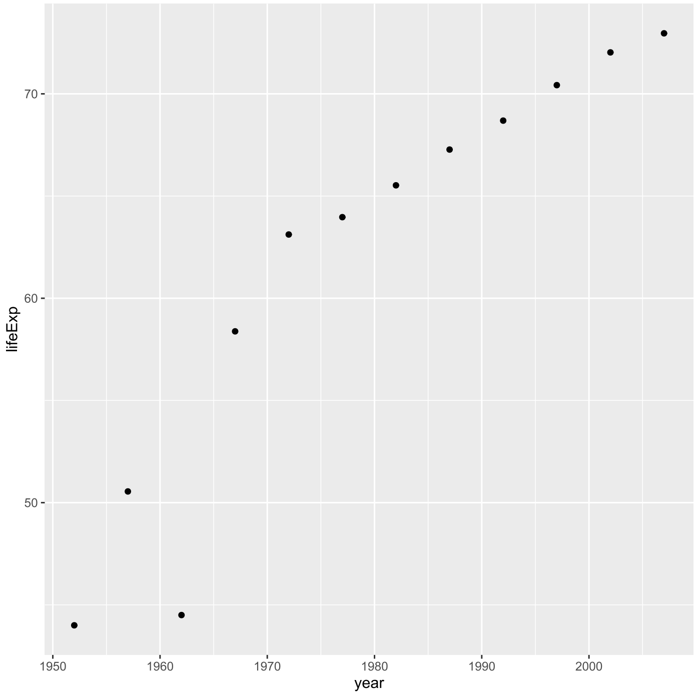
```
library(ggplot2)
```

The core idea underlying this package is the **layered grammar of graphics**: we can break up elements of a plot into pieces and combine them.

[1] Though this is not without debate

Chinese Life Expectancy in `ggplot`

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
    geom_point()
```



Structure of a ggplot

`ggplot2` graphics objects consist of two primary components:

1. **Layers**, the components of a graph.
    - We *add* layers to a `ggplot2` object using `+`.
    - This includes lines, shapes, and text.
2. **Aesthetics**, which determine how the layers appear.
    - We *set* aesthetics using *arguments* (e.g. `color="red"`) inside layer functions.
    - This includes locations, colors, and sizes.
    - Aesthetics also determine how data *map* to appearances.

Layers

Layers are the components of the graph, such as:

- `ggplot()`: initializes `ggplot2` object, specifies input data
- `geom_point()`: layer of scatterplot points
- `geom_line()`: layer of lines
- `ggtitle()`, `xlab()`, `ylab()`: layers of labels
- `facet_wrap()`: layer creating separate panels stratified by some factor wrapping around
- `facet_grid()`: same idea, but can split by two variables along rows and columns (e.g. `facet_grid(gender ~ age_group)`)
- `theme_bw()`: replace default gray background with black-and-white

Layers are separated by a `+` sign. For clarity, I usually put each layer on a new line, unless it takes few or no arguments (e.g. `xlab()`, `ylab()`, `theme_bw()`).

Aesthetics

Aesthetics control the appearance of the layers:

- `x`, `y`: x and y coordinate values to use
- `color`: set color of elements based on some data value
- `group`: describe which points are conceptually grouped together for the plot (often used with lines)
- `size`: set size of points/lines based on some data value
- `alpha`: set transparency based on some data value

Aesthetics: Setting vs. mapping

Layers take arguments to control their appearance, such as point/line colors or transparency (`alpha` between 0 and 1).

- Arguments like `color`, `size`, `linetype`, `shape`, `fill`, and `alpha` can be used directly on the layers (**setting aesthetics**), e.g. `geom_point(color = "red")`. See the `ggplot2` documentation for options. These *don't depend on the data.*

- Arguments inside `aes()` (**mapping aesthetics**) will *depend on the data*, e.g. `geom_point(aes(color = continent))`.

- `aes()` in the `ggplot()` layer gives overall aesthetics to use in other layers, but can be changed on individual layers (including switching `x` or `y` to different variables)

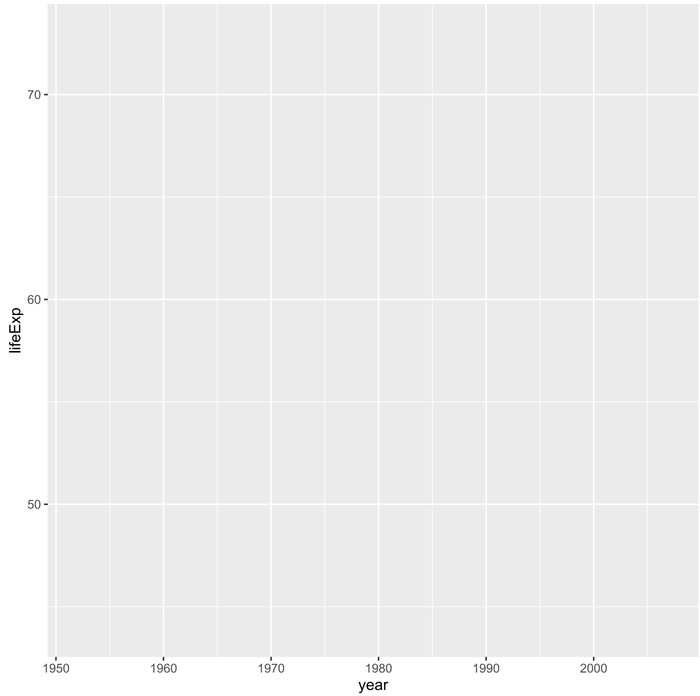
This may seem pedantic, but precise language makes searching for help easier.

Now let's see all this jargon in action.

Axis Labels, Points, No Background

1: Base Plot

```
ggplot(data = China,
       aes(x = year, y = lifeExp))
```

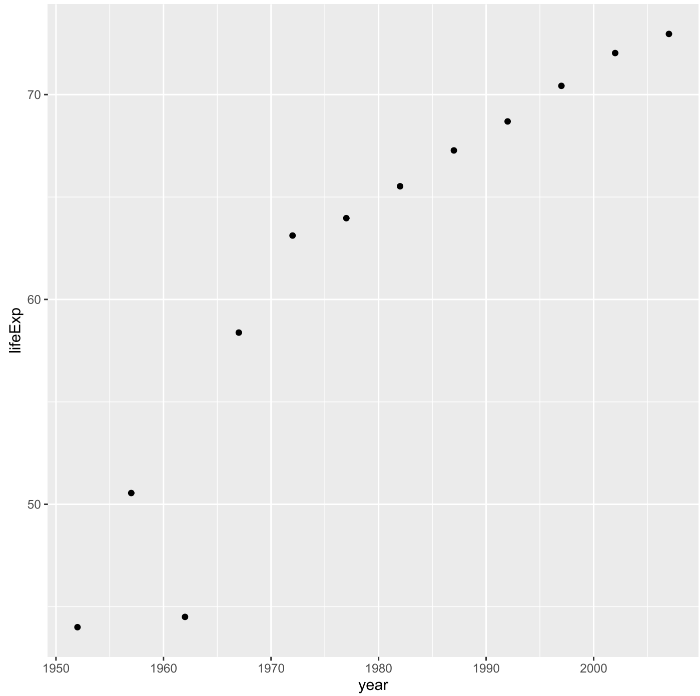


Initialize the plot with `ggplot()` and `x` and `y` aesthetics **mapped** to variables.

Axis Labels, Points, No Background

2: Scatterplot

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
  geom_point()
```

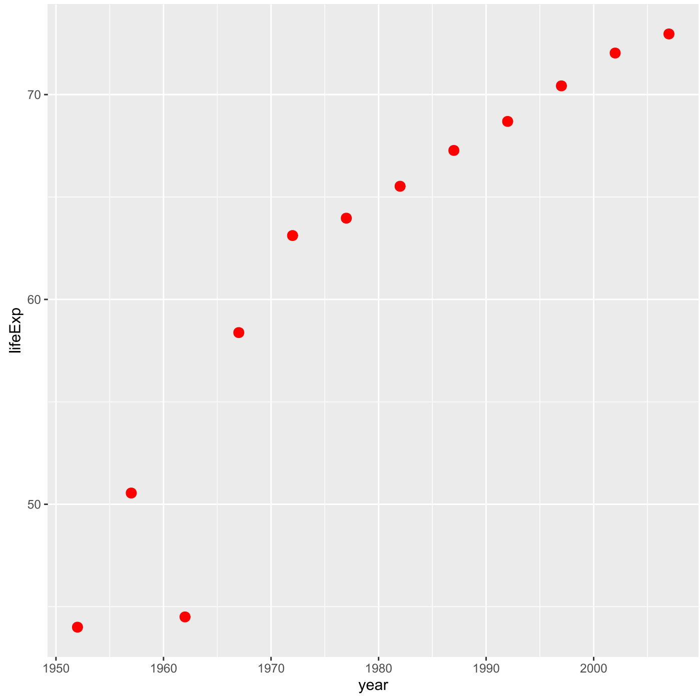


Add a scatterplot **layer**.

Axis Labels, Points, No Background

3: Point Color and Size

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
  geom_point(color = "red", size = 3)
```

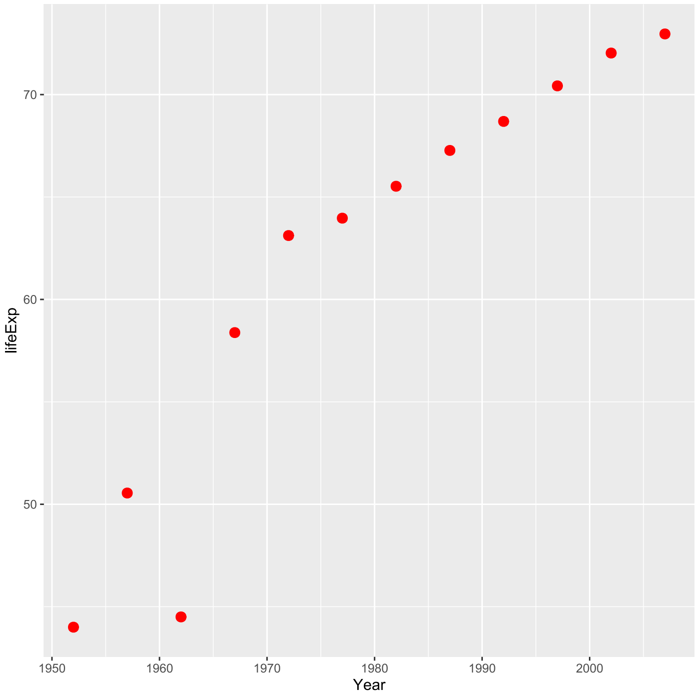


Set aesthetics to make the points large and red.

Axis Labels, Points, No Background

4: X-Axis Label

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
  geom_point(color = "red", size = 3) +
  xlab("Year")
```

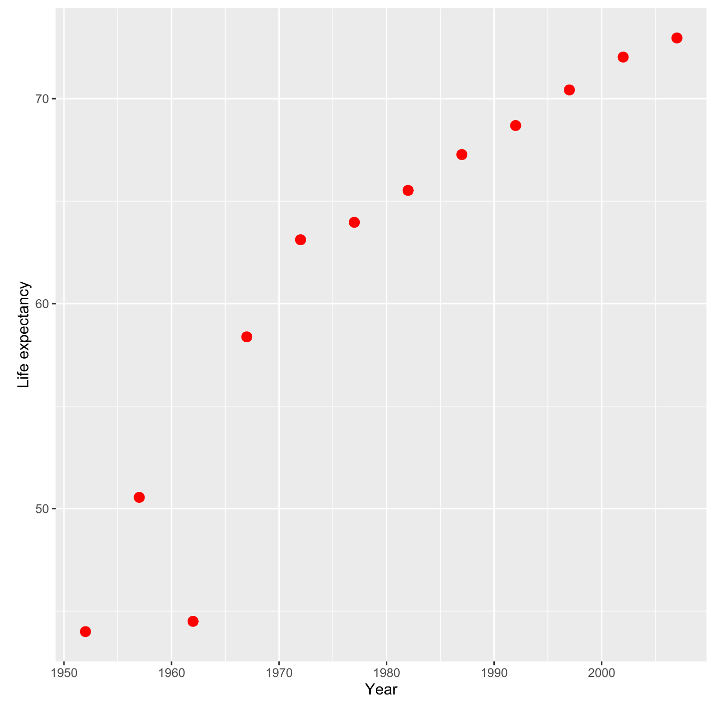


Add a layer to capitalize the x-axis label.

Axis Labels, Points, No Background

5: Y-Axis Label

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
  geom_point(color = "red", size = 3) +
  xlab("Year") +
  ylab("Life expectancy")
```

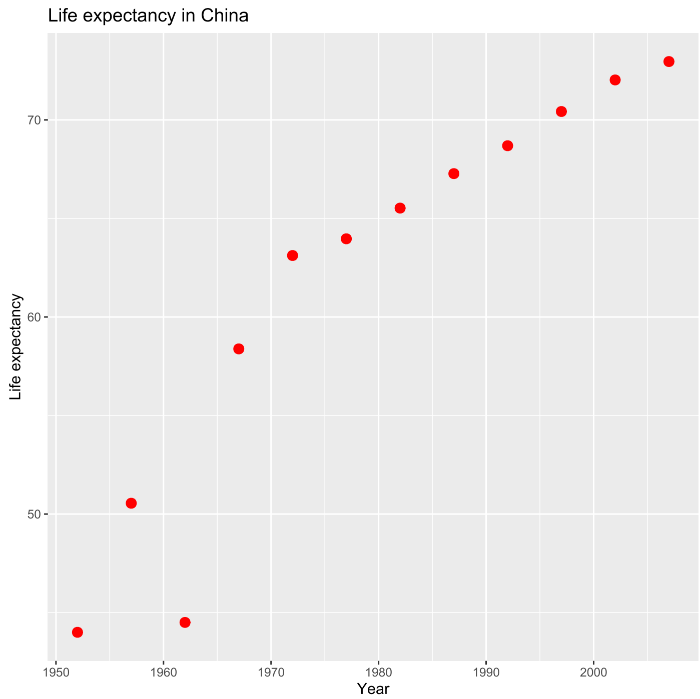


Add a layer to clean up the y-axis label.

Axis Labels, Points, No Background

6: Title

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
  geom_point(color = "red", size = 3) +
  xlab("Year") +
  ylab("Life expectancy") +
  ggtitle("Life expectancy in China")
```

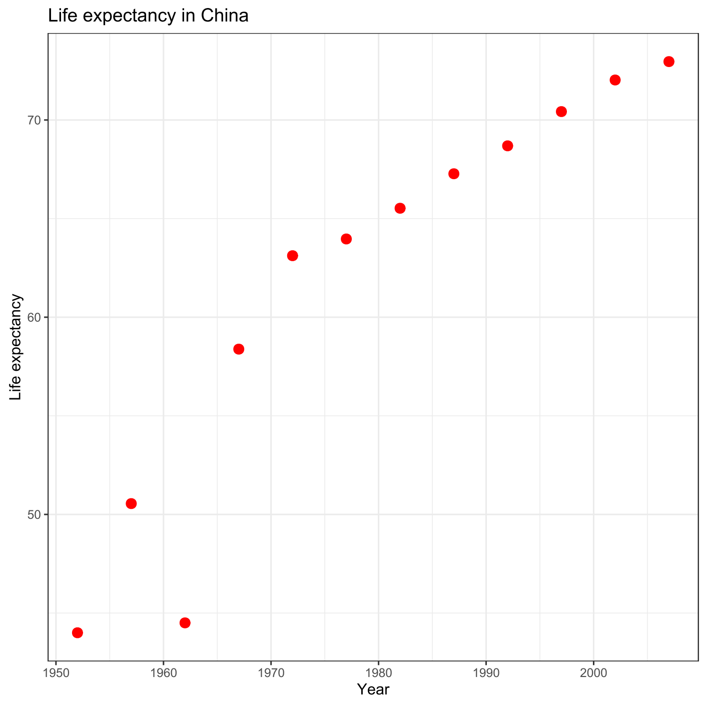


Add a title layer.

Axis Labels, Points, No Background

7: Theme

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
  geom_point(color = "red", size = 3) +
  xlab("Year") +
  ylab("Life expectancy") +
  ggtitle("Life expectancy in China") +
  theme_bw()
```

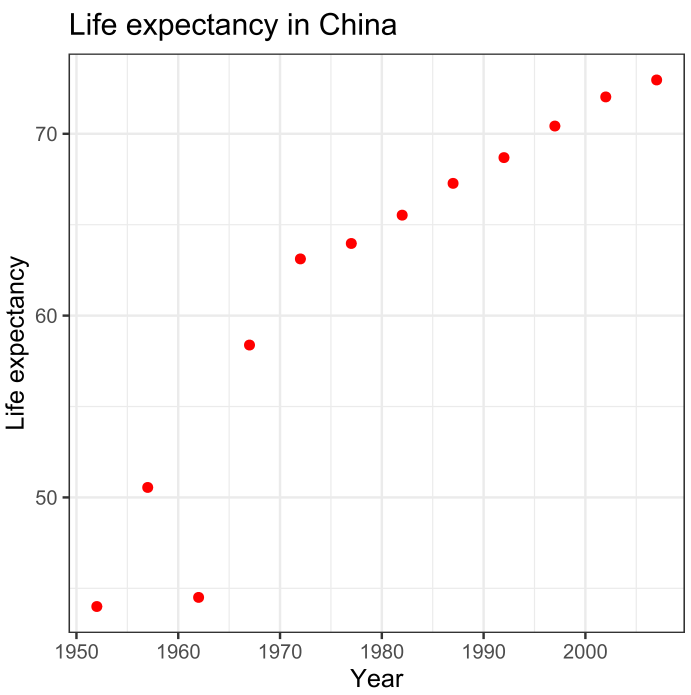


Pick a nicer theme with a new layer.

Axis Labels, Points, No Background

8: Text Size

```
ggplot(data = China,
       aes(x = year, y = lifeExp)) +
  geom_point(color = "red", size = 3) +
  xlab("Year") +
  ylab("Life expectancy") +
  ggtitle("Life expectancy in China") +
  theme_bw(base_size=18)
```

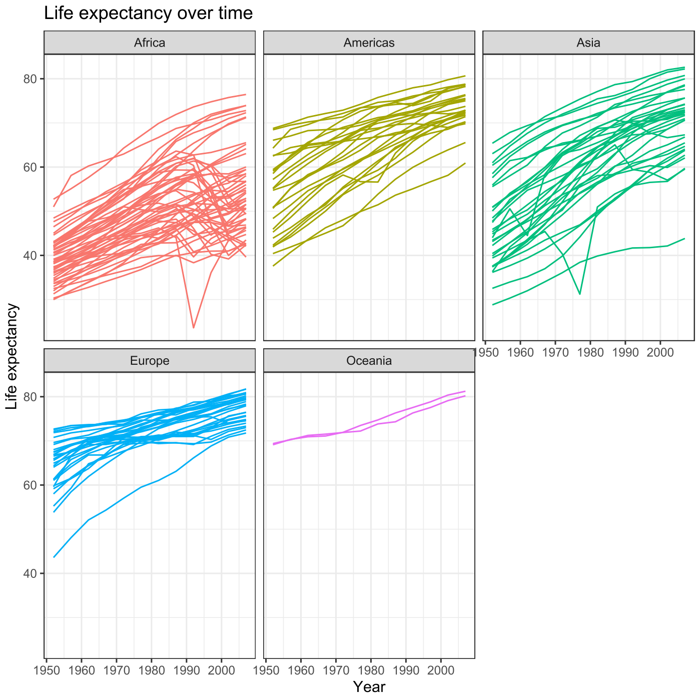


Increase the base text size.

Plotting All Countries

9: No Legend

```
ggplot(data = gapminder,
       aes(x = year, y = lifeExp,
           group = country,
           color = continent)) +
  geom_line() +
  xlab("Year") +
  ylab("Life expectancy") +
  ggtitle("Life expectancy over time") +
  theme_bw() +
  facet_wrap(~ continent) +
  theme(legend.position = "none")
```



Looking good!

Storing Plots

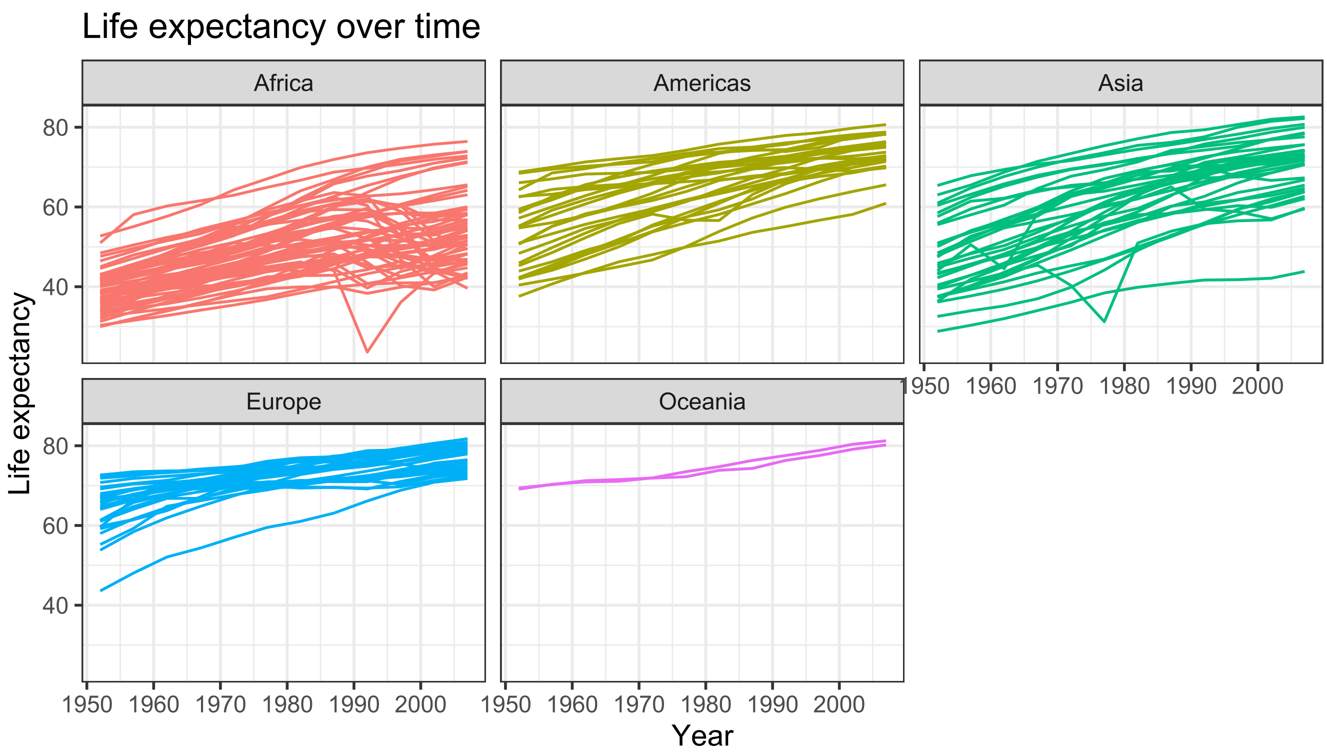
We can assign a `ggplot` object to a name:

```
lifeExp_by_year <-
  ggplot(data = gapminder,
       aes(x = year, y = lifeExp,
           group = country,
           color = continent)) +
  geom_line() +
  xlab("Year") +
  ylab("Life expectancy") +
  ggtitle("Life expectancy over time") +
  theme_bw() +
  facet_wrap(~ continent) +
  theme(legend.position = "none")
```

The graph won't be displayed when you do this. You can show the graph using a single line of code with just the object name, *or take the object and add more layers*.

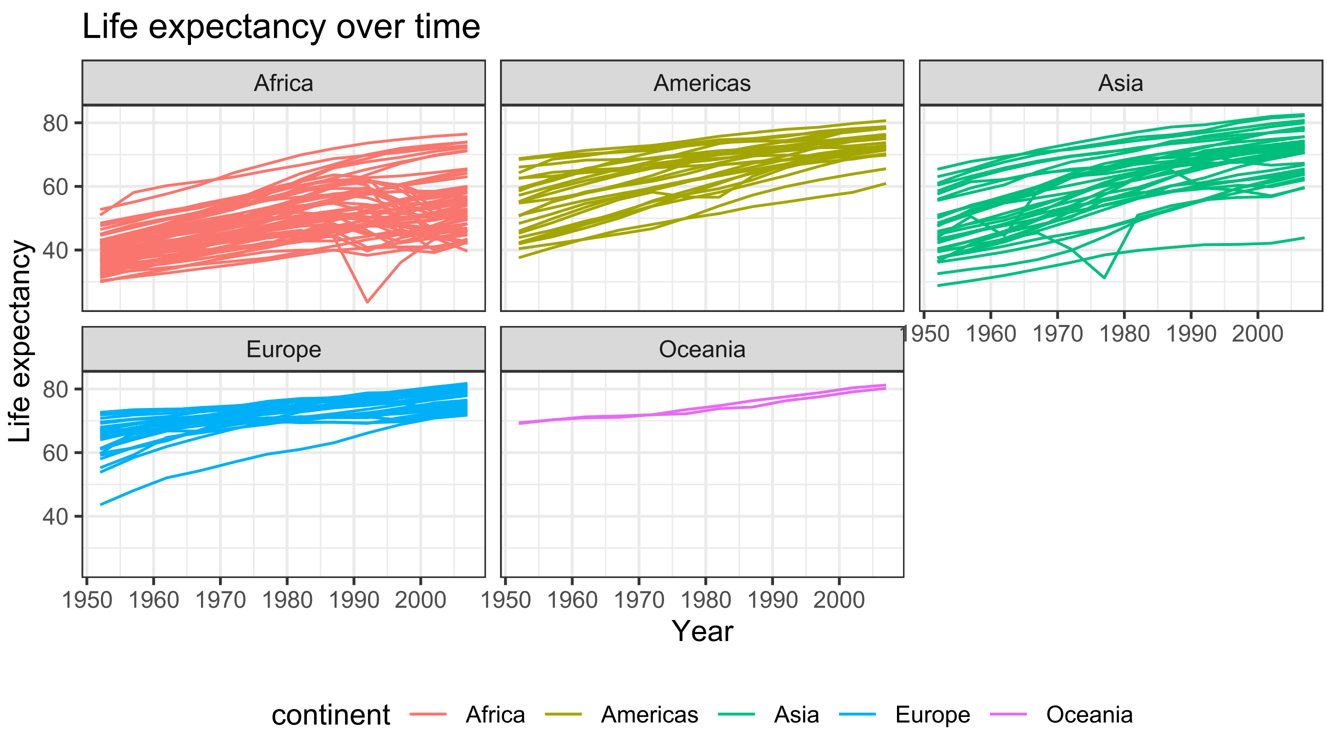
Showing a Stored Graph

```
lifeExp_by_year
```



Adding a Layer

```
lifeExp_by_year +
    theme(legend.position = "bottom")
```



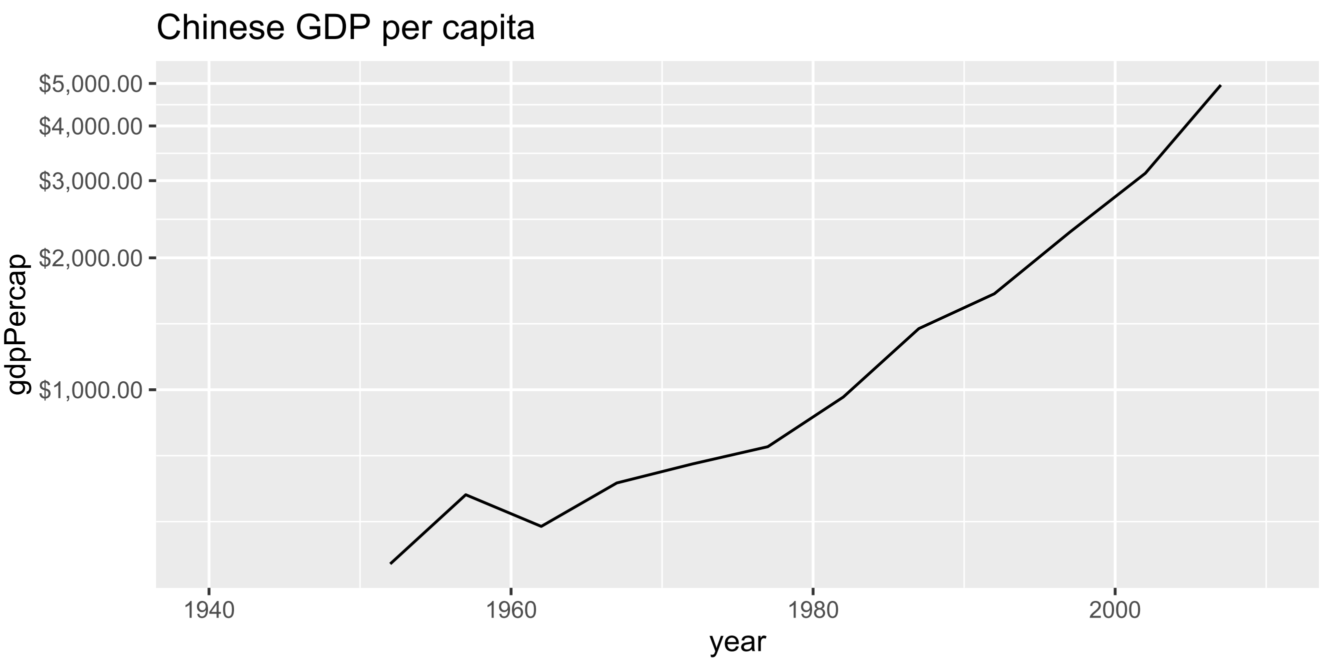
Changing the Axes

We can modify the axes in a variety of ways, such as:

- Change the x or y range using `xlim()` or `ylim()` layers
- Change to a logarithmic or square-root scale on either axis: `scale_x_log10()`, `scale_y_sqrt()`
- Change where the major/minor breaks are: `scale_x_continuous(breaks =, minor_breaks = )`

Axis Changes

```
ggplot(data = China, aes(x = year, y = gdpPercap)) +
    geom_line() +
    scale_y_log10(breaks = c(1000, 2000, 3000, 4000, 5000),
                  labels = scales::dollar) +
    xlim(1940, 2010) + ggtitle("Chinese GDP per capita")
```



Saving `ggplot` Plots

When you knit an R Markdown file, any plots you make are automatically saved in the "figure" folder in `.png` format. If you want to save another copy (perhaps of a different file type for use in a manuscript), use `ggsave()`:

```
ggsave("I_saved_a_file.pdf", plot = lifeExp_by_year,
       height = 3, width = 5, units = "in")
```

If you didn't manually set font sizes, these will usually come out at a reasonable size given the dimensions of your output file.

Bad/non-reproducible way[1]: choose *Export* on the plot preview or take a screenshot / snip.

[1] I still do this for quick emails of simple plots. Bad me!

Bonus Plot

`ggplot2` is well suited to making complex, publication ready plots.

This is the complete syntax for one plot from a recent article of mine.[1]

```
ggplot(estimated_pes, aes(x = Target, y = PE, group = Reporter)) +
  facet_grid(`Crime Type` ~ Neighborhood) +
  geom_errorbar(aes(ymin = LB, ymax = UB),
                position = position_dodge(width = .4), size = 0.75, width = 0.15) +
  geom_point(shape = 21, position = position_dodge(width = .4),
             size = 2, aes(fill = Reporter)) +
  scale_fill_manual("Reporter",
                    values = c("Any White" = "white", "All Black" = "black")) +
  ggtitle("Figure 3. Probability of Arrest",
          subtitle = "by Reporter and Target Race, Neighborhood and Crime Type") +
  xlab("Race of Target") + ylab("Estimated Probability") +
  theme_bw() + theme(legend.position = c(0.86, 0.15),
                     legend.background = element_rect(color = 1))
```

[1] Lanfear, Charles C., Lindsey R. Beach, Timothy A. Thomas. 2018. "Formal Social Control in Changing Neighborhoods: Racial Implications of Neighborhood Context on Reactive Policing." *City & Community* 17(4):1075-1099

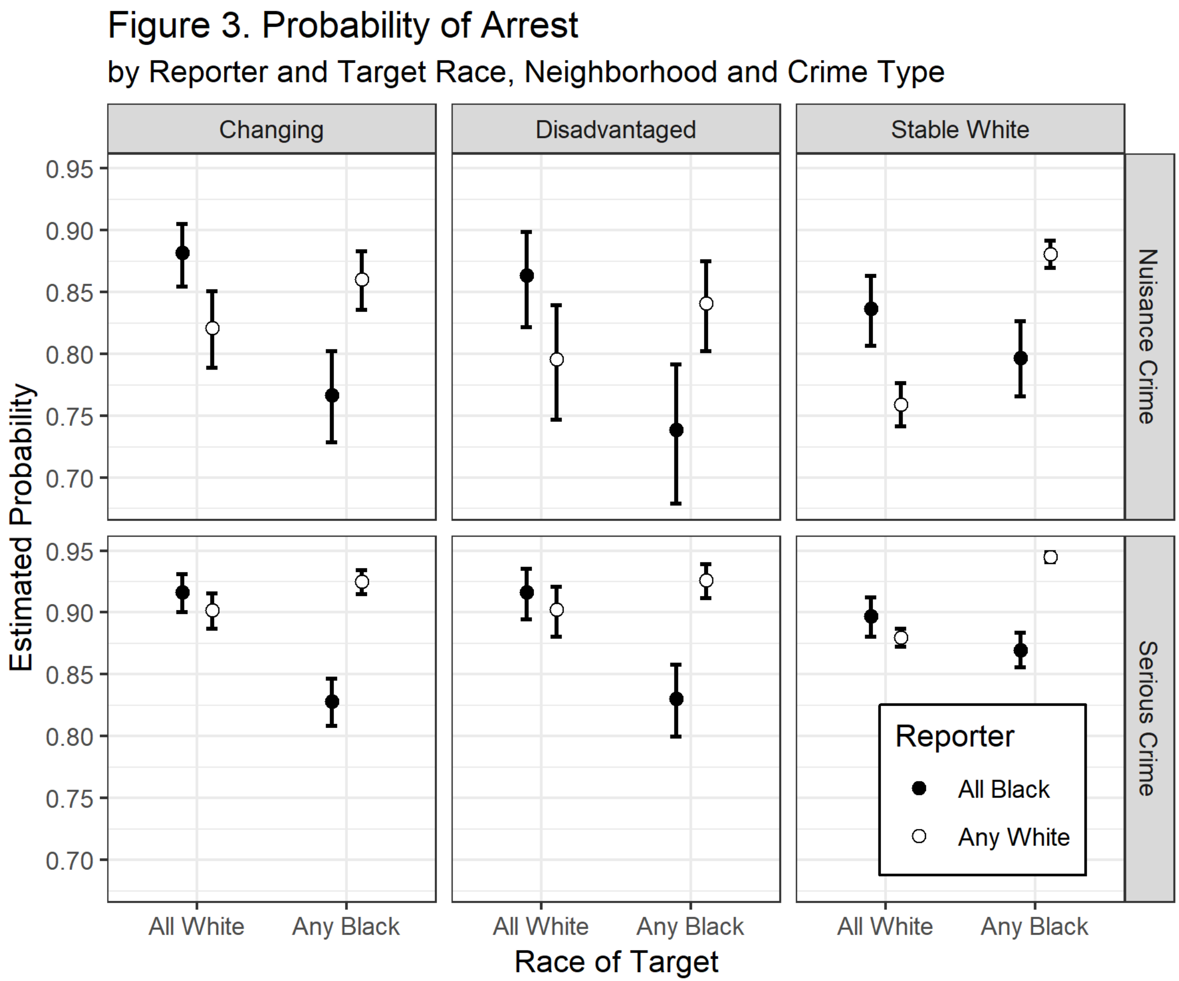
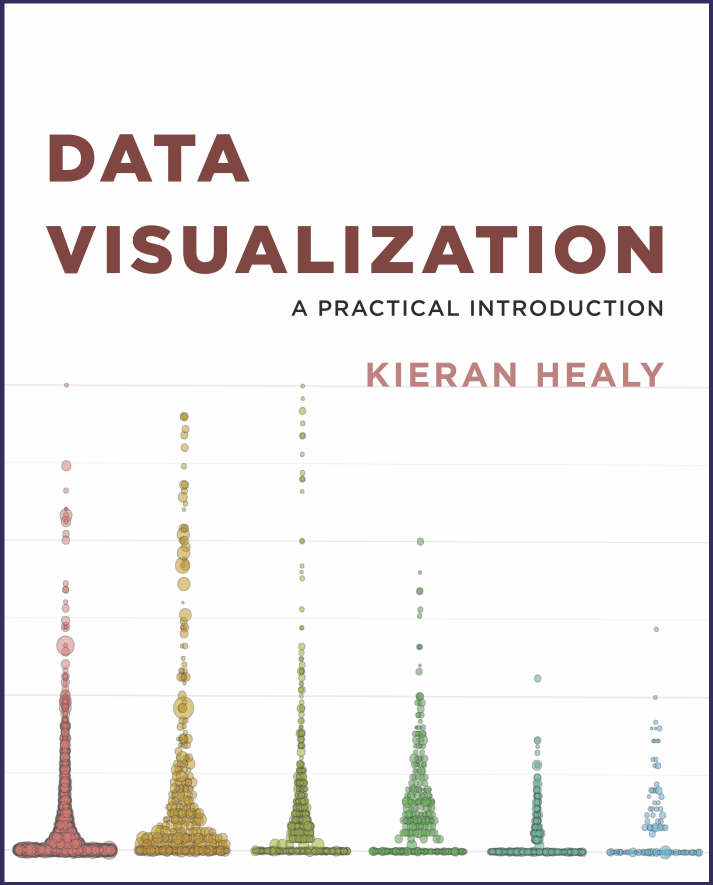


Figure 3. Probability of Arrest
by Reporter and Target Race, Neighborhood and Crime Type
Changing
Disadvantaged
Stable White
Nuisance Crime
Serious Crime
Estimated Probability
0.95
0.90
0.85
0.80
0.75
0.70
All White
Any Black
Race of Target
Reporter
All Black
Any White

Book Recommendation



- Targeted at Social Scientists without technical backgrounds
- Teaches good visualization principles
- Uses R, `ggplot2`, and `tidyverse`
- [Free online version!]
- Affordable in print

Summarizing Data

General Aggregation: `summarize()`

`summarize()` takes your column(s) of data and computes something using every row:

- Count how many rows there are
- Calculate the mean
- Compute the sum
- Obtain a minimum or maximum value

You can use any function in `summarize()` that aggregates *multiple values* into a *single value* (like `sd()`, `mean()`, or `max()`).

`summarize()` Example

Remember our Yugoslavia example from last unit?

```
yugoslavia <- gapminder %>% filter(country %in% c("Bosnia and Herzegovina",
               "Croatia", "Macedonia", "Montenegro", "Serbia", "Slovenia"))
```

For the year 1982, let's get the *number of observations*, *total population*, *mean life expectancy*, and *range of life expectancy* for former Yugoslavian countries.

```
yugoslavia %>% filter(year == 1982) %>%
    summarize(n_obs = n(),
              total_pop = sum(pop),
              mean_life_exp = mean(lifeExp),
              range_life_exp = max(lifeExp) - min(lifeExp))
```

```
## # A tibble: 1 x 4
##   n_obs total_pop mean_life_exp range_life_exp
##   <int>     <int>         <dbl>          <dbl>
## 1     5  20042685          71.3           3.94
```

These new variables are calculated using *all of the rows* in `yugoslavia`

Avoiding Repetition:

`summarize_at()`

Maybe you need to calculate the mean and standard deviation of a bunch of columns. With **`summarize_at()`**, put the variables to compute over first `vars()` (using `select()` syntax) and put the functions to use in `funs()` after.

```
yugoslavia %>%
    filter(year == 1982) %>%
    summarize_at(vars(lifeExp, pop), list(~ mean(.), ~ sd(.)))
```

```
## # A tibble: 1 x 4
##   lifeExp_mean pop_mean lifeExp_sd   pop_sd
##          <dbl>    <dbl>      <dbl>    <dbl>
## 1         71.3  4008537       1.60 3237282.
```

Note it automatically names the summarized variables based on the functions used to summarize.

Avoiding Repetition

Other functions:

There are additional `dplyr` functions similar to `summarize_at()`:

- `summarize_all()` and `mutate_all()` summarize / mutate *all* variables sent to them in the same way. For instance, getting the mean and standard deviation of an entire dataframe:

```
dataframe %>% summarize_all(list(~ mean(.), ~ sd(.)))
```

- `summarize_if()` and `mutate_if()` summarize / mutate all variables that satisfy some logical condition. For instance, summarizing every numeric column in a dataframe at once:

```
dataframe %>% summarize_if(is.numeric, list(~ mean(.), ~ sd(.)))
```

You can use all of these to avoid typing out the same code repeatedly!

`group_by()`

The special function **`group_by()`** changes how subsequent functions operate on the data, most importantly `summarize()`.

Functions after `group_by()` are computed *within each group* as defined by unique valus of the variables given, rather than over all rows at once.

Typically the variables you group by will be integers, factors, or characters, and *not continuous real values*.

`group_by()` example

```
yugoslavia %>%
  group_by(year) %>%
    summarize(num_countries = n_distinct(country),
              total_pop = sum(pop),
              total_gdp_per_cap = sum(pop*gdpPercap)/total_pop) %>%
    head(5)
```

```
## # A tibble: 5 x 4
##    year num_countries total_pop total_gdp_per_cap
##   <int>         <int>     <int>             <dbl>
## 1  1952             5  15436728             3030.
## 2  1957             5  16314276             4187.
## 3  1962             5  17099107             5257.
## 4  1967             5  17878535             6656.
## 5  1972             5  18579786             8730.
```

Because we did `group_by()` with `year` then used `summarize()`, we get *one row per value of `year`*!

Each value of year is its own **group**!

Window Functions

Grouping can also be used with `mutate()` or `filter()` to give rank orders within a group, lagged values, and cumulative sums. You can read more about window functions in this vignette.

```
yugoslavia %>%
  select(country, year, pop) %>%
  filter(year >= 2002) %>%
  group_by(country) %>%
  mutate(lag_pop = lag(pop, order_by = year),
         pop_chg = pop - lag_pop) %>%
  head(4)
```

```
## # A tibble: 4 x 5
## # Groups:   country [2]
##   country                 year     pop lag_pop pop_chg
##   <fct>                  <int>   <int>   <int>   <int>
## 1 Bosnia and Herzegovina  2002 4165416      NA      NA
## 2 Bosnia and Herzegovina  2007 4552198 4165416  386782
## 3 Croatia                 2002 4481020      NA      NA
## 4 Croatia                 2007 4493312 4481020   12292
```

Tidying Data

Initial Spot Checks

First things to check after loading new data:

- Did the last rows/columns from the original file make it in?
    - May need to use different package or manually specify range
- Are the column names in good shape?
    - Modify a `col_names=` argument or fix with `rename()`
- Are there "decorative" blank rows or columns to remove?
    - `filter()` or `select()` out those rows/columns
- How are missing values represented: `NA`, `" "` (blank), `.` (period), `999`?
    - Use `mutate()` with `ifelse()` to fix these (perhaps *en masse* with looping)
- Are there character data (e.g. ZIP codes with leading zeroes) being incorrectly represented as numeric or vice versa?
    - Modify `col_types=` argument, or use `mutate()` and `as.numeric()`

Slightly Messy Data

Program	Female	Male
Evans School	10	6
Arts & Sciences	5	6
Public Health	2	3
Other	5	1

- What is an observation?
    - A group of students from a program of a given gender
- What are the variables?
    - Program, Gender, Count
- What are the values?
    - Program: Evans School, Arts & Sciences, Public Health, Other
    - Gender: Female, Male -- **in the column headings, not its own column!**
    - Count: **spread over two columns!**

Tidy Version

Program	Gender	Count
Evans School	Female	10
Evans School	Male	6
Arts & Sciences	Female	5
Arts & Sciences	Male	6
Public Health	Female	2
Public Health	Male	3
Other	Female	5
Other	Male	1

Each variable is a column.

Each observation is a row.

Ready to throw into `ggplot()` or a model!

Billboard Data

We're going to work with some *ugly* data: *The Billboard Hot 100 for the year 2000*.

We can load it like so:

```
library(readr) # Contains read_csv()
billboard_2000_raw <-
  read_csv(file = "https://github.com/clanfear/Intermediate_R_Worksho
           col_types = paste(c("icccD", rep("i", 76)), collapse=""))
```

`col_types=` is used to specify column types. [See here for details.]

Billboard is Just Ugly-Messy

year	artist	track	time	date.entered	wk1	wk2	wk3	wk4	wk5
2000	2 Pac	Baby Don't Cry (Keep...	4:22	2000-02-26	87	82	72	77	87
2000	2Ge+her	The Hardest Part Of ...	3:15	2000-09-02	91	87	92	NA	NA
2000	3 Doors Down	Kryptonite	3:53	2000-04-08	81	70	68	67	66
2000	3 Doors Down	Loser	4:24	2000-10-21	76	76	72	69	67
2000	504 Boyz	Wobble Wobble	3:35	2000-04-15	57	34	25	17	17
2000	98^0	Give Me Just One Nig...	3:24	2000-08-19	51	39	34	26	26
2000	A*Teens	Dancing Queen	3:44	2000-07-08	97	97	96	95	100
2000	Aaliyah	I Don't Wanna	4:15	2000-01-29	84	62	51	41	38
2000	Aaliyah	Try Again	4:03	2000-03-18	59	53	38	28	21
2000	Adams, Yolanda	Open My Heart	5:30	2000-08-26	76	76	74	69	68
2000	Adkins, Trace	More	3:05	2000-04-29	84	84	75	73	73
2000	Aguilera, Christina	Come On Over Baby (A...	3:38	2000-08-05	57	47	45	29	23

Week columns continue up to `wk76`!

Billboard

- What are the **observations** in the data?
  - Week since entering the Billboard Hot 100 per song
- What are the **variables** in the data?
  - Year, artist, track, song length, date entered Hot 100, week since first entered Hot 100 (**spread over many columns**), rank during week (**spread over many columns**)
- What are the **values** in the data?
  - e.g. 2000; 3 Doors Down; Kryptonite; 3 minutes 53 seconds; April 8, 2000; Week 3 (**stuck in column headings**); rank 68 (**spread over many columns**)

Tidy Data

Tidy data (aka "long data") are such that:

1. The values for a single observation are in their own row.
2. The values for a single variable are in their own column.
3. The observations are all of the same nature.

Why do we want tidy data?

- Easier to understand many rows than many columns
- Required for plotting in `ggplot2`
- Required for many types of statistical procedures (e.g. hierarchical or mixed effects models)
- Fewer confusing variable names
- Fewer issues with missing values and "imbalanced" repeated measures data

`tidyr`

The `tidyr` package provides functions to tidy up data, similar to `reshape` in Stata or `varstocases` in SPSS. Key functions:

- **`pivot_longer()`**: takes a set of columns and pivots them down to make two new columns (which you can name yourself):
  - A `name` column that stores the original column names
  - A `value` with the values in those original columns
- **`pivot_wider()`**: inverts `pivot_longer()` by taking two columns and pivoting them up into multiple columns
- **`separate()`**: pulls apart one column into multiple columns (common after `pivot_longer()` where values are embedded in column names)
  - `extract_numeric()` does a simple version of this for the common case when you just want grab the number part
- **`extract()`** for pivoting a column into multiple *sets* of columns.
  - See Hadley's response to this question for an example.

`pivot_longer()`

Let's use `pivot_longer()` to get the week and rank variables out of their current layout into two columns (big increase in rows, big drop in columns):

```
library(tidyr)
billboard_2000 <- billboard_2000_raw %>%
  pivot_longer(starts_with("wk"),
               names_to ="week", values_to = "rank")
dim(billboard_2000)
```

```
## [1] 24092     7
```

`starts_with()` and other helper functions from `dplyr::select()` work here too.

We could instead use: `pivot_longer(wk1:wk76, names_to = "week", values_to = "rank")` to pull out these contiguous columns.

`pivot`ed Weeks

```
head(billboard_2000)
```

```
## # A tibble: 6 x 7
##    year artist track                   time  date.entered week   rank
##   <int> <chr>  <chr>                   <chr> <date>       <chr> <int>
## 1  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk1      87
## 2  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk2      82
## 3  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk3      72
## 4  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk4      77
## 5  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk5      87
## 6  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26   wk6      94
```

Now we have a single week column!

Pivoting Better?

```
summary(billboard_2000$rank)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##       1      26      51      51      76     100   18785
```

This is an improvement, but we don't want to keep the 18785 rows with missing ranks (i.e. observations for weeks since entering the Hot 100 that the song was no longer on the Hot 100).

Pivoting Better: `values_drop_na`

The argument `values_drop_na = TRUE` to `pivot_longer()` will remove rows with missing ranks.

```
billboard_2000 <- billboard_2000_raw %>%
  pivot_longer(starts_with("wk"),
               names_to ="week", values_to = "rank",
               values_drop_na = TRUE)
summary(billboard_2000$rank)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0    26.0    51.0    51.1    76.0   100.0
```

`parse_number()`

`tidyr` provides a convenience function to grab just the numeric information from a column that mixes text and numbers:

```
billboard_2000 <- billboard_2000 %>%
    mutate(week = parse_number(week))
summary(billboard_2000$week)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0     5.0    10.0    11.5    16.0    65.0
```

For more sophisticated conversion or pattern checking, you'll need to use string parsing (to be covered in week 8).

Or use `names_prefix`

```
billboard_2000 <- billboard_2000_raw %>%
  pivot_longer(starts_with("wk"),
               names_to ="week", values_to = "rank",
               values_drop_na = TRUE,
               names_prefix = "wk",
               names_ptypes = list("week" =numeric(0)))
head(billboard_2000)
```

```
## # A tibble: 6 x 7
##    year artist track                   time  date.entered  week  rank
##   <int> <chr>  <chr>                   <chr> <date>       <dbl> <int>
## 1  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26       1    87
## 2  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26       2    82
## 3  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26       3    72
## 4  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26       4    77
## 5  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26       5    87
## 6  2000 2 Pac  Baby Don't Cry (Keep... 4:22  2000-02-26       6    94
```

`separate()`

The track length column isn't analytically friendly. Let's convert it to a number rather than the character (minutes:seconds) format:

```
billboard_2000 <- billboard_2000 %>%
    separate(time, into = c("minutes", "seconds"),
             sep = ":", convert = TRUE) %>%
    mutate(length = minutes + seconds / 60) %>%
    select(-minutes, -seconds)
summary(billboard_2000$length)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.60    3.67    3.93    4.03    4.28    7.83
```

`sep = :` tells `separate()` to split the column into two where it finds a colon (`:`).

Then we add `seconds / 60` to `minutes` to produce a numeric `length` in minutes.

`pivot_wider()` Motivation

`pivot_wider()` is the opposite of `pivot_longer()`, which you use if you have data for the same observation taking up multiple rows.

Example of data that we probably want to pivot wider (unless we want to plot each statistic in its own facet):

Group	Statistic	Value
A	Mean	1.28
A	Median	1.0
A	SD	0.72
B	Mean	2.81
B	Median	2
B	SD	1.33

A common cue to use `pivot_wider()` is having measurements of different quantities in the same column.

Before `pivot_wider()`

```
(too_long_data <- data.frame(Group = c(rep("A", 3), rep("B", 3)),
                             Statistic = rep(c("Mean", "Median", "SD"), 2),
                             Value = c(1.28, 1.0, 0.72, 2.81, 2, 1.33)))
```

```
##   Group Statistic Value
## 1     A      Mean  1.28
## 2     A    Median  1.00
## 3     A        SD  0.72
## 4     B      Mean  2.81
## 5     B    Median  2.00
## 6     B        SD  1.33
```

After `pivot_wider()`

```
(just_right_data <- too_long_data %>%
    pivot_wider(names_from = Statistic, values_from = Value))
```

```
## # A tibble: 2 x 4
##   Group  Mean Median    SD
##   <fct> <dbl>  <dbl> <dbl>
## 1 A      1.28      1  0.72
## 2 B      2.81      2  1.33
```

Charts of 2000: Data Prep

Let's look at songs that hit #1 at some point and look how they got there versus songs that did not:

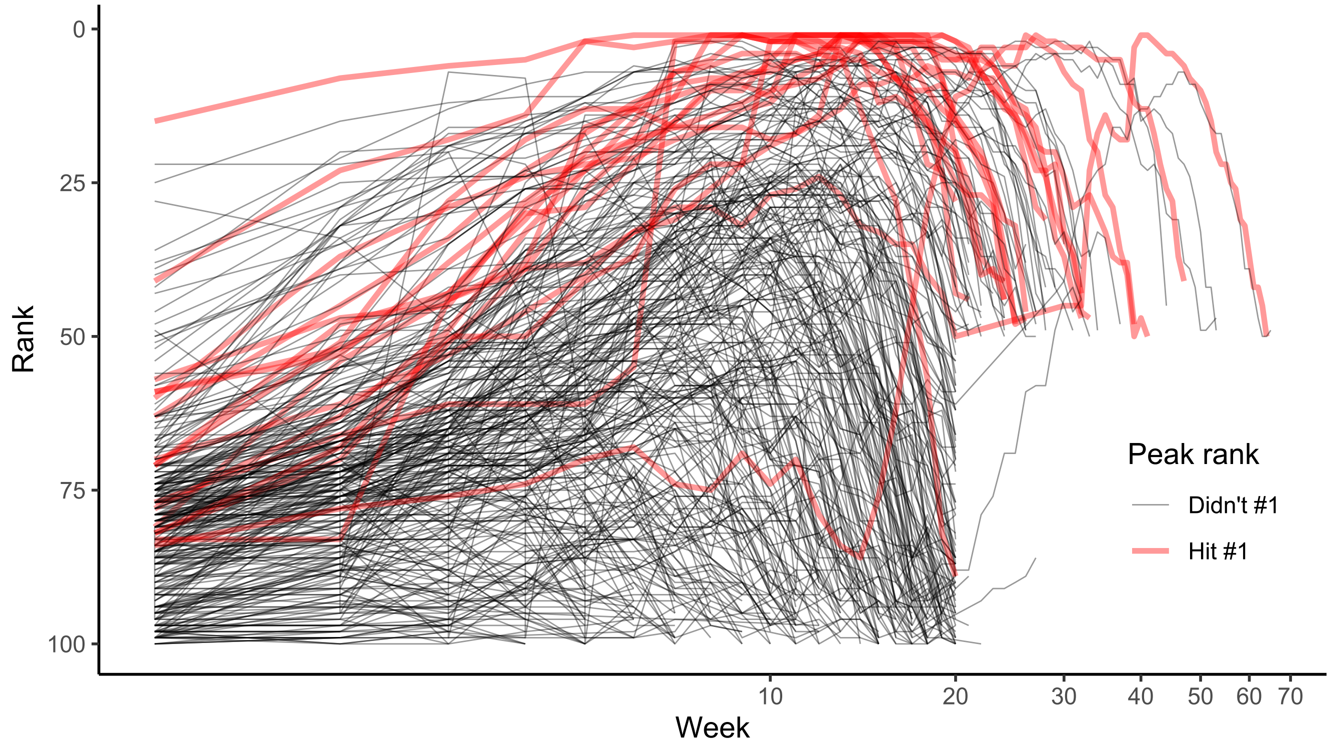
```r
# find best rank for each song
best_rank <- billboard_2000 %>%
    group_by(artist, track) %>%
    summarize(min_rank = min(rank),
              weeks_at_1 = sum(rank == 1)) %>%
    mutate(`Peak rank` = ifelse(min_rank == 1,
                                "Hit #1",
                                "Didn't #1"))

# merge onto original data
billboard_2000 <- billboard_2000 %>%
    left_join(best_rank, by = c("artist", "track"))
```

Note that because the "highest" rank is *numerically lowest* (1), we are summarizing with `min()`.

Charts of 2000: `ggplot2`

```r
library(ggplot2)
billboard_trajectories <-
  ggplot(data = billboard_2000,
         aes(x = week, y = rank, group = track,
             color = `Peak rank`)
         ) +
  geom_line(aes(size = `Peak rank`), alpha = 0.4) +
    # rescale time: early weeks more important
  scale_x_log10(breaks = seq(0, 70, 10)) +
  scale_y_reverse() + # want rank 1 on top, not bottom
  theme_classic() +
  xlab("Week") + ylab("Rank") +
  scale_color_manual(values = c("black", "red")) +
  scale_size_manual(values = c(0.25, 1)) +
  theme(legend.position = c(0.90, 0.25),
        legend.background = element_rect(fill="transparent"))
```

Charts of 2000: Beauty!



Observation: There appears to be censoring around week 20 for songs falling out of the top 50 that I'd want to follow up on.

Which Were #1 the Most Weeks?

```
billboard_2000 %>%
    select(artist, track, weeks_at_1) %>%
    distinct(artist, track, weeks_at_1) %>%
    arrange(desc(weeks_at_1)) %>%
    head(7)
```

```
## # A tibble: 7 x 3
##   artist              track                   weeks_at_1
##   <chr>               <chr>                        <int>
## 1 Destiny's Child     Independent Women Pa...         11
## 2 Santana             Maria, Maria                    10
## 3 Aguilera, Christina Come On Over Baby (A...          4
## 4 Madonna             Music                            4
## 5 Savage Garden       I Knew I Loved You               4
## 6 Destiny's Child     Say My Name                      3
## 7 Iglesias, Enrique   Be With You                      3
```

Getting Usable Dates

We have the date the songs first charted, but not the dates for later weeks. We can calculate these now that the data are tidy:

```
billboard_2000 <- billboard_2000 %>%
    mutate(date = date.entered + (week - 1) * 7)
billboard_2000 %>% arrange(artist, track, week) %>%
    select(artist, date.entered, week, date, rank) %>% head(4)
```

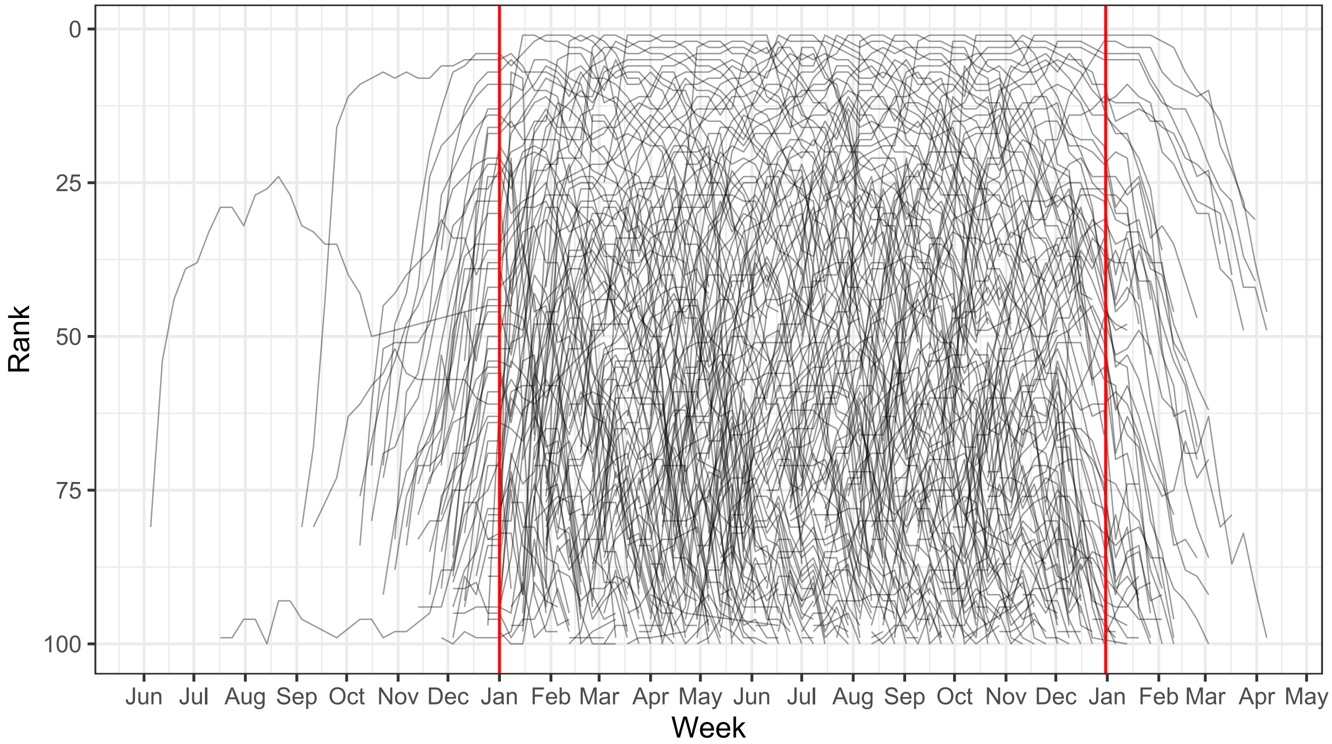
```
## # A tibble: 4 x 5
##   artist date.entered  week date         rank
##   <chr>  <date>       <dbl> <date>      <int>
## 1 2 Pac  2000-02-26       1 2000-02-26     87
## 2 2 Pac  2000-02-26       2 2000-03-04     82
## 3 2 Pac  2000-02-26       3 2000-03-11     72
## 4 2 Pac  2000-02-26       4 2000-03-18     77
```

This works because `date` objects are in units of days—we just add 7 days per week to the start date.

Preparing to Plot Over Calendar Time

```r
plot_by_day <-
  ggplot(billboard_2000, aes(x = date, y = rank, group = track)) +
  geom_line(size = 0.25, alpha = 0.4) +
  # just show the month abbreviation label (%b)
  scale_x_date(date_breaks = "1 month", date_labels = "%b") +
  scale_y_reverse() + theme_bw() +
  # add lines for start and end of year:
  # input as dates, then make numeric for plotting
  geom_vline(xintercept = as.numeric(as.Date("2000-01-01", "%Y-%m-%d")),
             col = "red") +
  geom_vline(xintercept = as.numeric(as.Date("2000-12-31", "%Y-%m-%d")),
             col = "red") +
  xlab("Week") + ylab("Rank")
```

Calendar Time Plot!



We see some of the entry dates are before 2000---presumably songs still charting during 2000 that came out earlier.

Joining Data

When Do We Need to Join Data?

- Want to make columns using criteria too complicated for `ifelse()` or `case_when()`
    - We can work with small sets of variables then combine them back together.
- Combine data stored in separate data sets: e.g. UW registrar information with police stop records.
    - Often large surveys are broken into different data sets for each level (e.g. household, individual, neighborhood)

Joining in Concept

We need to think about the following when we want to merge data frames `A` and `B`:

- Which *rows* are we keeping from each data frame?
- Which *columns* are we keeping from each data frame?
- Which variables determine whether rows *match*?

Join Types: Rows and columns kept

There are many types of joins[1]...

- `A %>% left_join(B)`: keep all rows from `A`, matched with `B` wherever possible (`NA` when not), keep columns from both `A` and `B`
- `A %>% right_join(B)`: keep all rows from `B`, matched with `A` wherever possible (`NA` when not), keep columns from both `A` and `B`
- `A %>% inner_join(B)`: keep only rows from `A` and `B` that match, keep columns from both `A` and `B`
- `A %>% full_join(B)`: keep all rows from both `A` and `B`, matched wherever possible (`NA` when not), keep columns from both `A` and `B`
- `A %>% semi_join(B)`: keep rows from `A` that match rows in `B`, keep columns from only `A`
- `A %>% anti_join(B)`: keep rows from `A` that *don't* match a row in `B`, keep columns from only `A`

[1] Usually `left_join()` does the job.

Matching Criteria

We say rows should *match* because they have some columns containing the same value. We list these in a `by =` argument to the join.

Matching Behavior:

- No `by`: Match using all variables in `A` and `B` that have identical names
- `by = c("var1", "var2", "var3")`: Match on identical values of `var1`, `var2`, and `var3` in both `A` and `B`
- `by = c("Avar1" = "Bvar1", "Avar2" = "Bvar2")`: Match identical values of `Avar1` variable in `A` to `Bvar1` variable in `B`, and `Avar2` variable in `A` to `Bvar2` variable in `B`

Note: If there are multiple matches, you'll get *one row for each possible combination* (except with `semi_join()` and `anti_join()`).

Need to get more complicated? Break it into multiple operations.

`nycflights13` Data

We'll use data in the [`nycflights13` package].

```
library(nycflights13)
```

It includes five dataframes, some of which contain missing data (`NA`):

- `flights`: flights leaving JFK, LGA, or EWR in 2013
- `airlines`: airline abbreviations
- `airports`: airport metadata
- `planes`: airplane metadata
- `weather`: hourly weather data for JFK, LGA, and EWR

Note these are *separate data frames*, each needing to be *loaded separately*:

```
data(flights)
data(airlines)
data(airports)
# and so on...
```

Join Example 1

Which airlines had the most flights to Seattle from NYC?

```
flights %>% filter(dest == "SEA") %>%
    select(carrier) %>%
    left_join(airlines, by = "carrier") %>%
    count(name) %>%
    arrange(desc(n))
```

```
## # A tibble: 5 x 2
##   name                        n
##   <chr>                   <int>
## 1 Delta Air Lines Inc.     1213
## 2 United Air Lines Inc.    1117
## 3 Alaska Airlines Inc.      714
## 4 JetBlue Airways           514
## 5 American Airlines Inc.    365
```

`count(name)` is a shortcut for `group_by(name) %>% summarize(n(.))`: It creates a variable `n` equal to the number of rows in each group.

Join Example 2

Who manufactures the planes that flew to SeaTac?

```
flights %>% filter(dest == "SEA") %>% select(tailnum) %>%
    left_join(planes %>% select(tailnum, manufacturer),
              by = "tailnum") %>%
    count(manufacturer) %>% # Count observations by manufacturer
    arrange(desc(n)) # Arrange data descending by count
```

```
## # A tibble: 6 x 2
##   manufacturer           n
##   <chr>              <int>
## 1 BOEING              2659
## 2 AIRBUS               475
## 3 AIRBUS INDUSTRIE     394
## 4 <NA>                 391
## 5 BARKER JACK L          2
## 6 CIRRUS DESIGN CORP     2
```

Note you can perform operations on the data inside functions such as `left_join()` and the *output* will be used by the function.

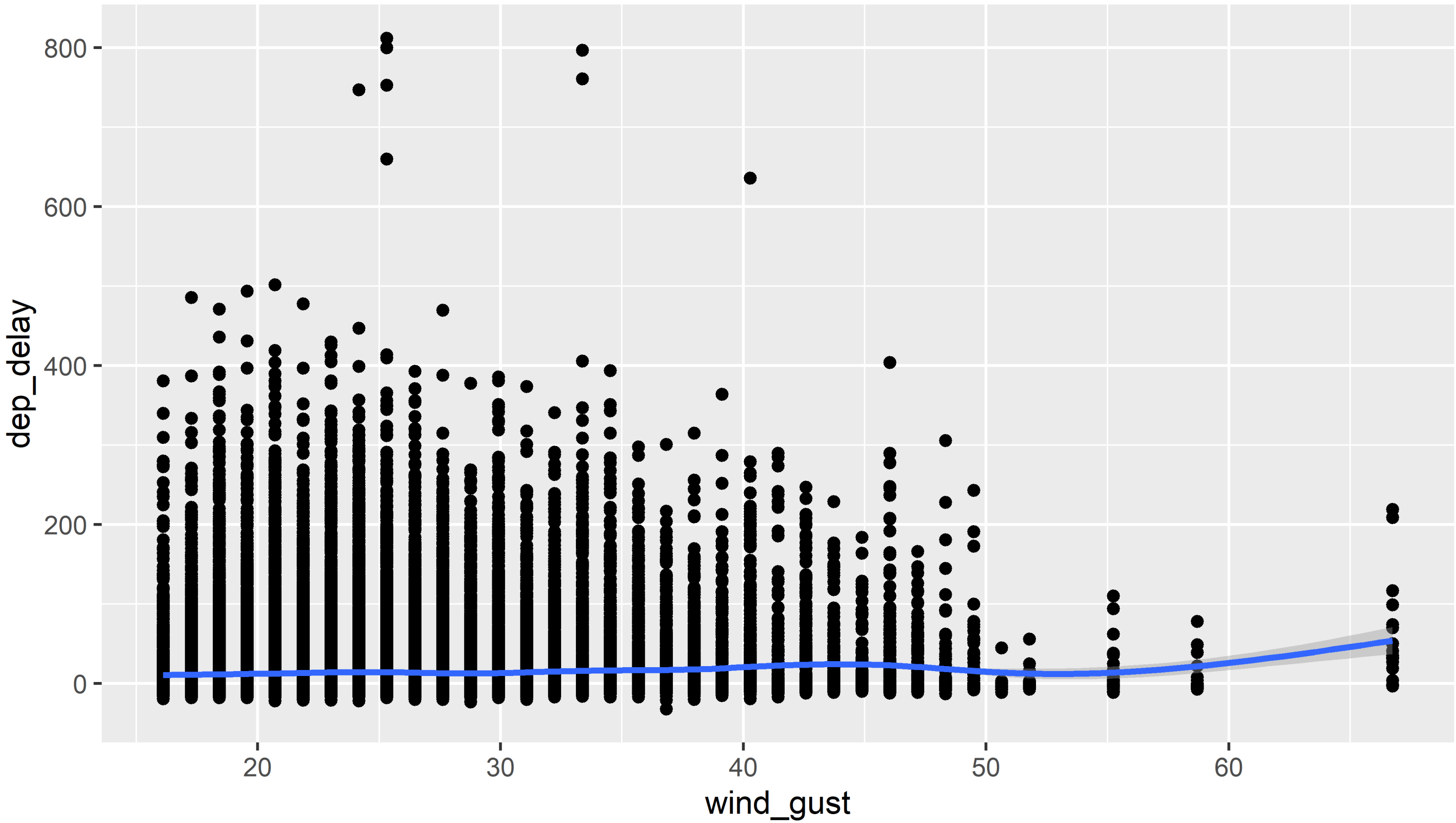
Join Example 3

Is there a relationship between departure delays and wind gusts?

```
flights %>%
    select(origin, year, month, day, hour, dep_delay) %>%
    inner_join(weather,
           by = c("origin", "year", "month", "day", "hour")) %>%
    select(dep_delay, wind_gust) %>%
    # removing rows with missing values
    filter(!is.na(dep_delay) & !is.na(wind_gust)) %>%
    ggplot(aes(x = wind_gust, y = dep_delay)) +
      geom_point() +
      geom_smooth()
```

Because the data are the first argument for `ggplot()`, we can pipe them straight into a plot.

Wind Gusts and Delays



Resources

- UW CSSS508: My University of Washington Introduction to R course which forms the basis for this workshop. All content including lecture videos is freely available.
- R for Data Science online textbook by Garrett Grolemund and Hadley Wickham. One of many good R texts available, but importantly it is free and focuses on the `tidyverse` collection of R packages which are the modern standard for data manipulation and visualization in R.
- Advanced R online textbook by Hadley Wickham. A great source for more in-depth and advanced R programming.
- `swirl`: Interactive tutorials inside R.
- Useful RStudio cheatsheets on R Markdown, RStudio shortcuts, etc.