Introduction to R, RStudio, and R Markdown

Ralph Trane

Spring

Preface

- We will **NEVER** need to actually open `R` itself - always work through RStudio
- We won't deal with `R` scripts. Instead we will use what is known as R Markdown, which is a simple way of including `R` code, `R` output, and text in the same document.

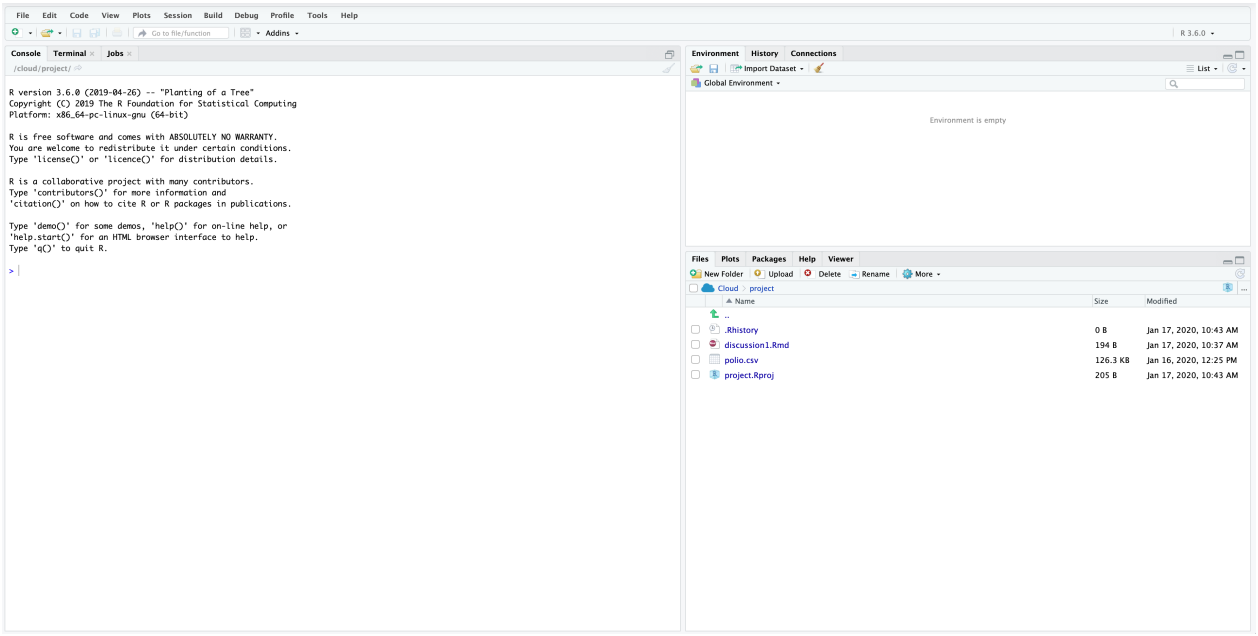
Getting Started

Installation

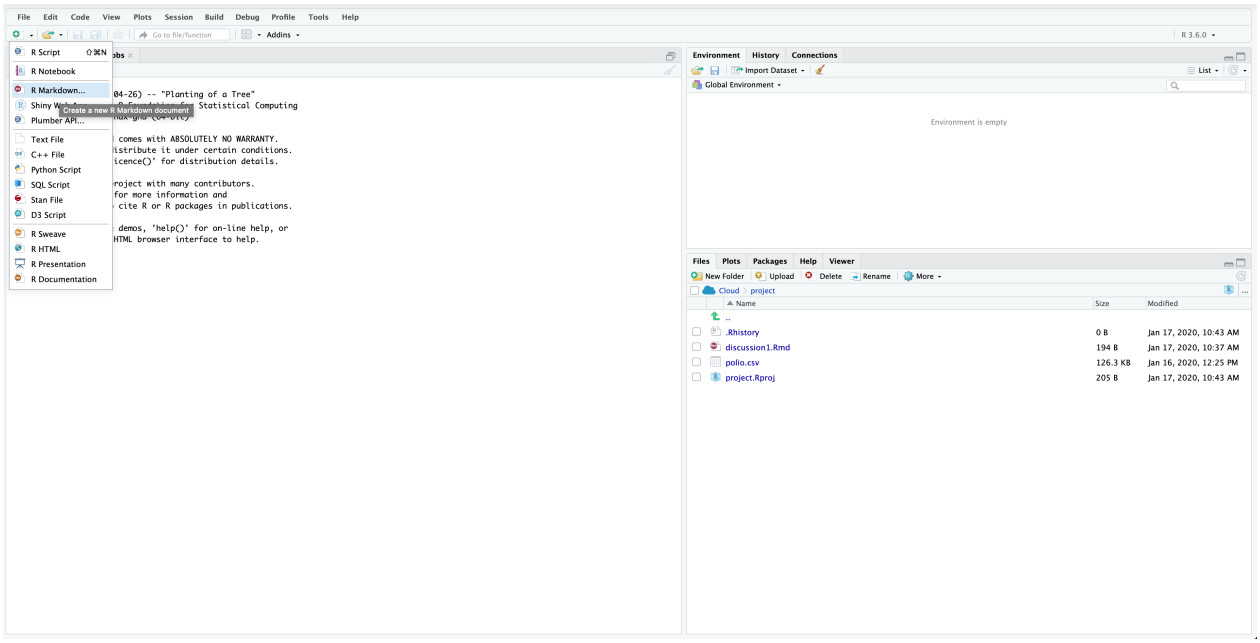
See canvas announcement about installation of `R` and RStudio.

RStudio

When you first open RStudio, you will be presented with something like this:

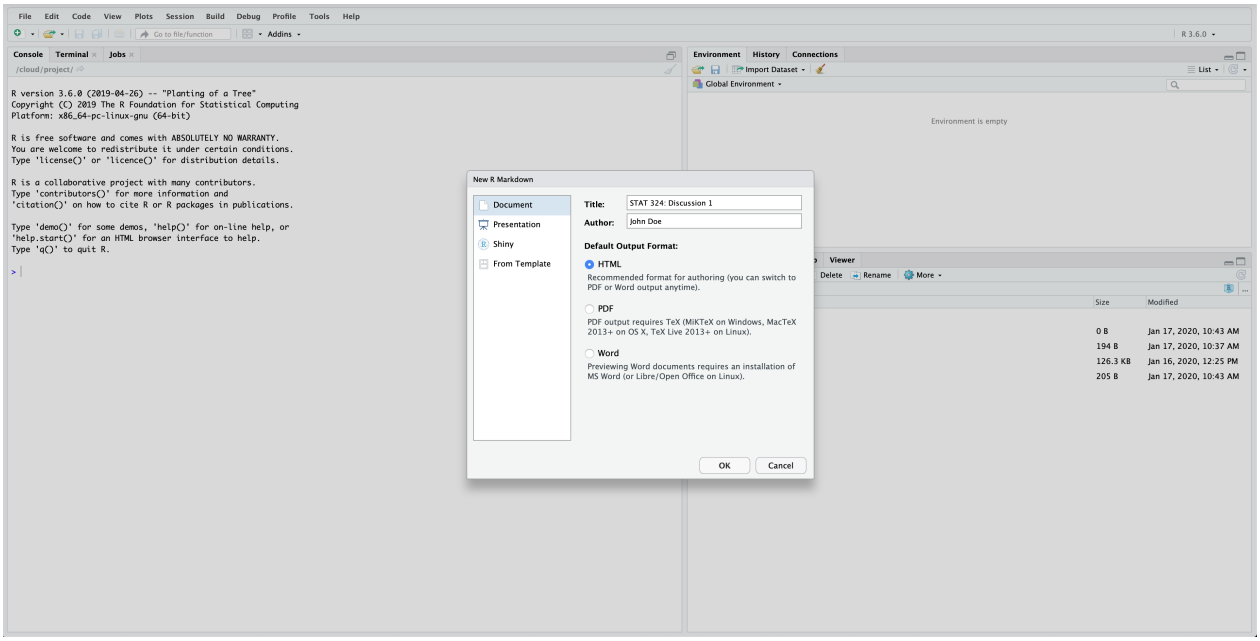


To get started with a new document, click the R Markdown document in the dropdown menu:



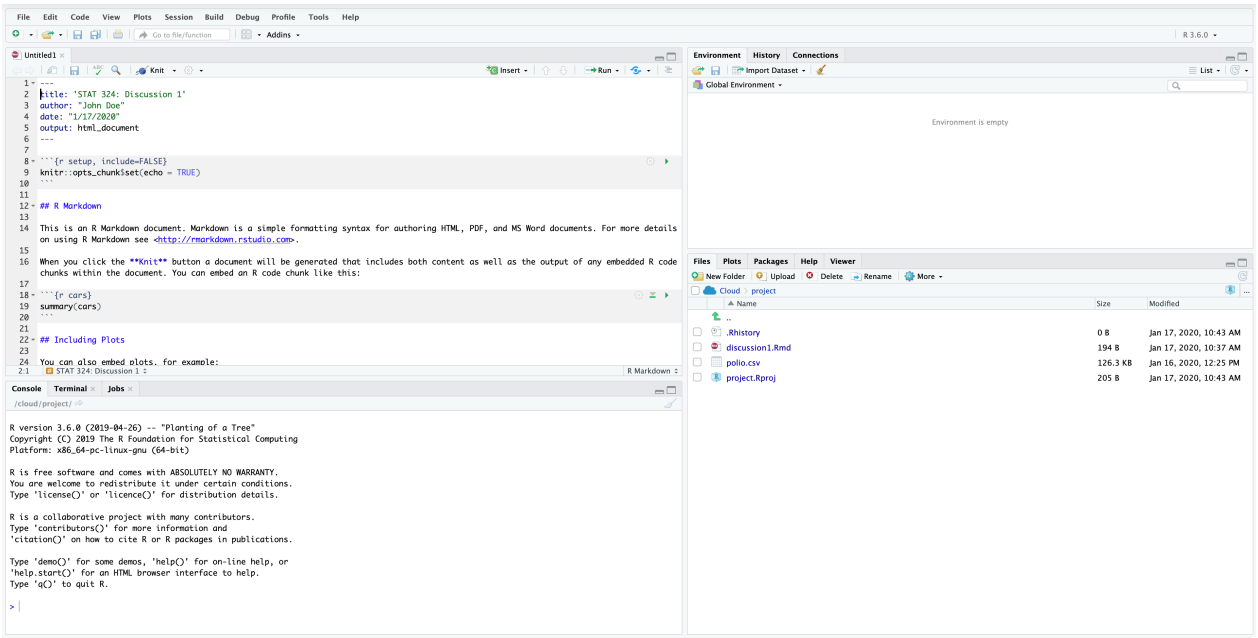
The first time you do this, a pop-up box will tell you you do not have the needed packages installed. Just click yes/ok/whatever it says. This will install everything you need.

Next, a pop-up menu will be displayed. Here you will give your document a title, and put your name down as the author of the document. I will let you decide if you want to final document to be an html or pdf file. If you choose pdf, there are a few more steps (see here). Pdf files look better, but are sometimes also a bit harder to work with. With html files you are more free to focus on the content – no worrying about page breaks and other annoying things.



Clicking OK will open a new document ("Untitled1") with an R Markdown template. The first part of it is called the yaml header, and is enclosed by `---`. Here we can specify different options. You will see that the

title you chose before is already there as is your name, the date, and the output format you chose. If you ever change you mind about any of these, you can change it right here.



Looking at the screenshot above, you will see four panels. These are:

- Top left: your document. This is where you will be working most of the time.
- Bottom left: the console. This is where R code is actually run. If you need to install packages, or quickly try something simple, this is where you'd do that.
- Top right: any data sets, variables, etc. that we define in our working session will pop up here. Very neat to keep track of what's going on.
- Bottom right: any plots or help pages will show up here.

The most important parts are on your left hand side (the document and console).

R Markdown

The R Markdown document consists of three main parts:

1. The header
2. Text (more specifically, markdown portion)
3. R code chunks

We've already mentioned the header – it is the bit between the `---` at the top of the document.

Text is simply everything that is NOT an R chunk.

R chunks are where we write our code. This will be run and the output shown below it once we knit our document (more on this in a second). A code chunk starts with ```` ```{r} ```` and end with ```` ``` ````. RStudio will help us keep track of code chunks by greying them out slightly. To get a new code chunk, you can either type out the beginning and end as mentioned, click `Insert -> R` at the top right of the document panel, click `Code -> Insert Chunk` at the top of the window, or use the keyboard shortcut `Cmd/ctrl + Alt + I`.

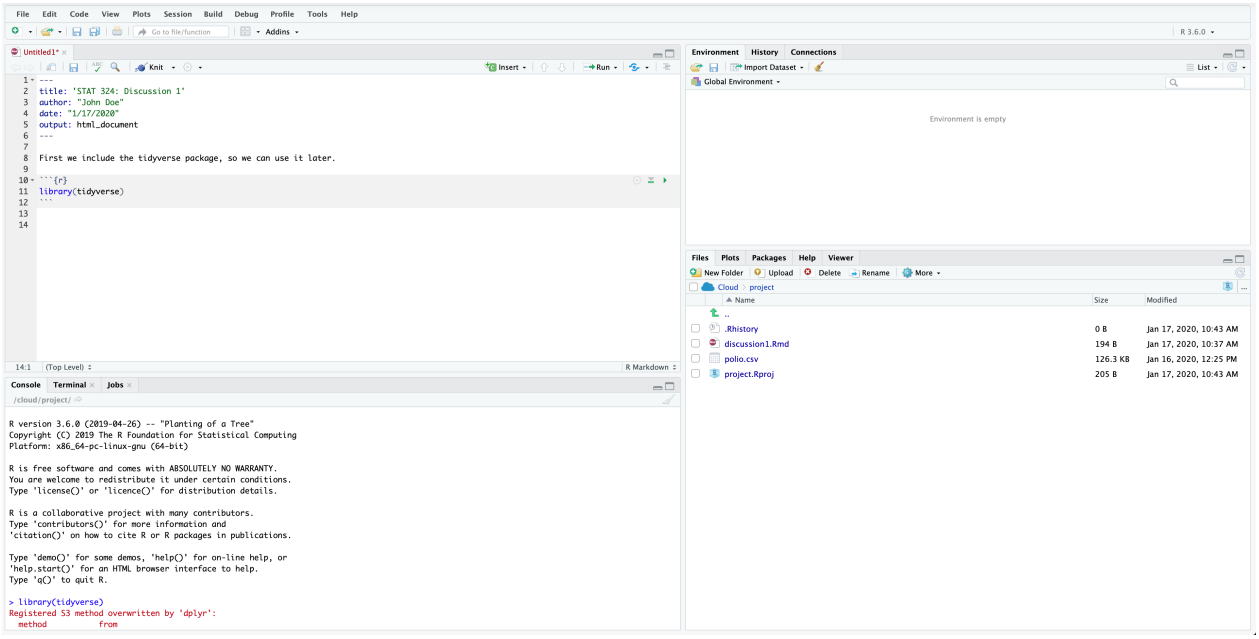
When not inside the code chunk, you are basically just writing text. There are a bunch of different things you can do in terms of formatting. We will introduce these as we go along.

R

R is an incredibly powerful statistical software. Its main strength is the incredible community that constantly develops packages, which we can take full advantage of. A few of the ones we will use are **ggplot2**, **dplyr**, and **tidyr**. These are all part of a bigger picture often refered to as the **tidyverse**. Quite conveniently, they are all bundled together into one big package called... **tidyverse**. So to install these, we can simply install this package. Do this by running the following bit of code in the console:

```
install.packages("tidyverse")
```

Once installed, we can use it. But before we can use it, we need to tell R that we want to use is. We usually do this as the first thing in our document. Delete everything after the header in your document, insert a new code chunk, and write `library(tidyverse)` in it. Include a line above this that explains what you are doing. The result should be something like this:



To run this code, place the cursor on the line, and hit `Cmd/ctrl + Enter`. Notice how the line is copied to the console! This indicates that the line has been run. There's also a bunch of output. Don't worry about this.

Now, R can obviously be used as a basic calculator (go ahead and write `2+5` in the console, hit enter, then try `sqrt(9)`). This is unbelievably uninteresting. R is meant to be used for data analysis, which is exactly what we will use it for! I strongly believe in "learning by doing", so instead of going through all the boring steps that most "R Tutorials" do (you can find literally hundreds of these if you are interested!), we will take a look at some actual data!

A big part of using R is using the appropriate functions. R comes with a bunch of built in functions, such as **mean**, **sd**, **length**, and **sum**, and all the packages we load add to the long list of available functions.

Take a look at the following, and see if you can guess what the output would be

```
mean(c(1, 2, 3))
length(c(3,2,1,8,5,2,9,5))
sum(c(1,1,1,2))
```

Our First Analysis

The objective here is for you to create a very preliminary data analysis. I will provide all the code you need, but you will need to copy it all into your own document, include text around it to describe what it does, and comment on the results. As we go along, I will ask you some questions. Write your answers in your document.

Load packages

We load the `tidyverse` package.

```
library(tidyverse)
```

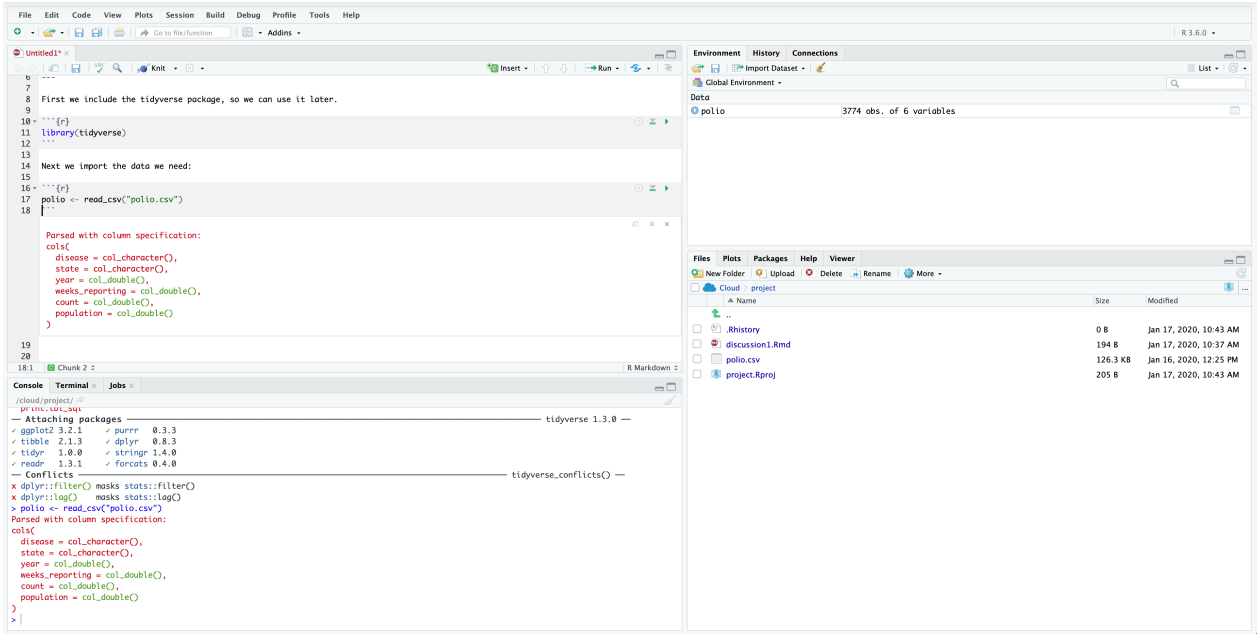
Import Data

We will use a data set containing data about polio in the US with data from 1938 to 2011. Before we can use the data, we need to read it into `R`. Since this data is provided as a .csv file, we do so using the `read_csv` function:

```
polio <- read_csv("polio.csv")
```

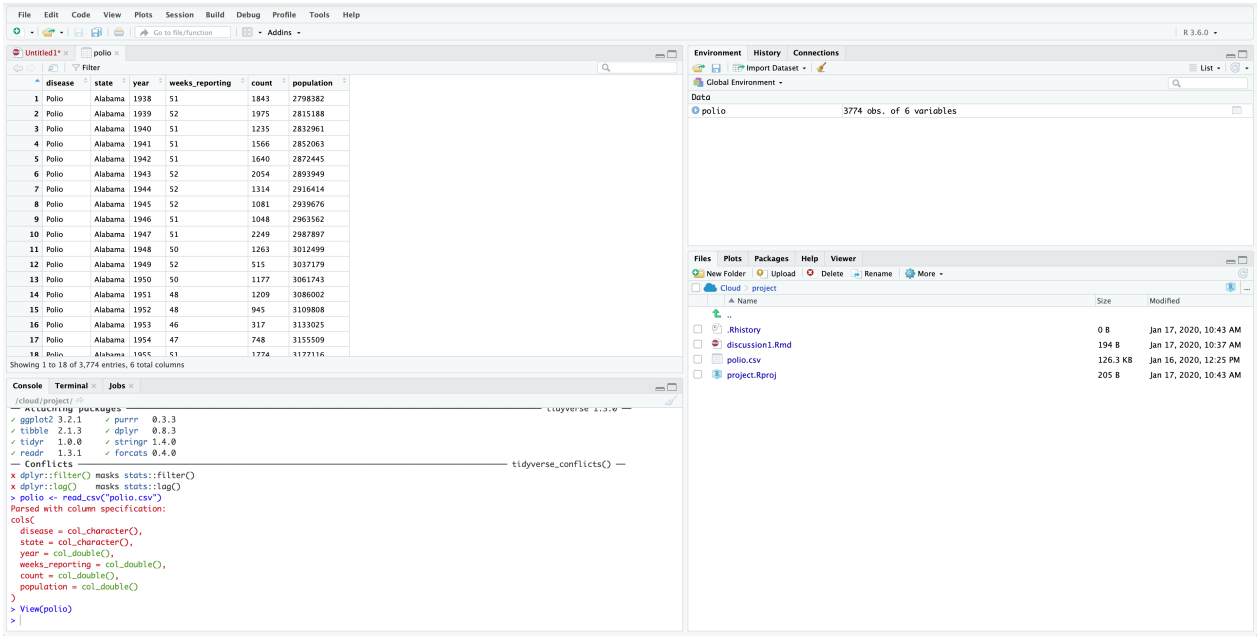
```
## Parsed with column specification:
## cols(
##   disease = col_character(),
##   state = col_character(),
##   year = col_double(),
##   weeks_reporting = col_double(),
##   count = col_double(),
##   population = col_double()
## )
```

This will save the data as an object called `polio`. Once you've run the code, you'll see that it shows up in the top right panel:



First look at data

If you click the object, you will be able to see it in the top left panel:



You can now answer the first two questions. Once you have answered these questions, save your document, and click the knit button. A much better looking document should appear. This is what you will have to hand in for your homework!

As you work on your document, you will want to knit every so often to make sure you are not making mistakes along the way.

Some summaries

The first thing one could be interested in could be the mean number of cases a year. In other words, we want to calculate the mean of the `count` column. To do so, we simply use the `mean` function:

```
mean(polio$count)
```

```
## [1] 617.1224
```

As you can see above, you can use `$` to extract a single column of your data.

Now, the truth is that this is not very informative. What would be much more informative would be to find the mean for each year. There are many ways of doing this. One way would be to *group* the data by `year`, then *summarize* it:

```
measles_grouped <- group_by(polio, year)
summarize(measles_grouped, mean_count = mean(count))
```

```
## # A tibble: 74 x 2
##     year mean_count
##    <dbl>      <dbl>
##  1  1938      4100.
##  2  1939      3640.
##  3  1940      3383.
##  4  1941      3969.
##  5  1942      3484.
##  6  1943      3500.
##  7  1944      2070.
##  8  1945      2347.
##  9  1946      1971.
## 10  1947      2904.
## # ... with 64 more rows
```

This quickly becomes hard to keep track of. You have to constantly come up with new names for the intermediate steps, and when the analysis is a bit more convoluted than this example, it becomes both hard to come up with names, and hard to keep track of what object has what information.

Luckily, there's a rather simple solution to this. It's called the pipe, and the symbol is `%>%`. (Keyboard shortcut: `Cmd/ctrl + shift + M`.) What this does is it takes the output of what is before it, and sends it to the first argument of the function after it.

An example: the following three lines of code are the same:

```
sqrt(7)
```

```
## [1] 2.645751
```

```
7 %>% sqrt
```

```
## [1] 2.645751
```

```
3 %>%
  sum(4) %>%
  sqrt
```

```
## [1] 2.645751
```

So are the following three.

```
sum(2,1,7,-6,2)
```

```
## [1] 6
```

```
c(2,1,7,-6,2) %>% sum
```

```
## [1] 6
```

```
c(2,1) %>%
  sum(7,-6, 2)
```

```
## [1] 6
```

We can use this to simplify the code above:

```
polio %>%
  group_by(year) %>%
  summarize(mean_count = mean(count))
```

```
## # A tibble: 74 x 2
##     year mean_count
##    <dbl>      <dbl>
##  1  1938      4100.
##  2  1939      3640.
##  3  1940      3383.
##  4  1941      3969.
##  5  1942      3484.
##  6  1943      3500.
##  7  1944      2070.
##  8  1945      2347.
##  9  1946      1971.
## 10  1947      2904.
## # ... with 64 more rows
```

What this does is it takes our data (`polio`), groups it by year, then summarizes (within each group) by calculating the mean of the count column. The way I usually think about it is that it chops up our data into small data sets (one for each year), and then basically runs `mean(count)` for each data set.

We can also use this to calculate the overall mean – we simply don't group the data. Notice how this is the same as what we calculated above.

```
polio %>%
  summarize(mean_count = mean(count))
```

```
## # A tibble: 1 x 1
##   mean_count
##        <dbl>
## 1       617.
```

It's really cool, because we can easily modify this so it works on a per state basis instead of per year:

```
polio %>%
  group_by(state) %>%
  summarize(mean_count = mean(count))
```

```
## # A tibble: 51 x 2
##    state                mean_count
##    <chr>                     <dbl>
##  1 Alabama                   345.
##  2 Alaska                     13.4
##  3 Arizona                   357.
##  4 Arkansas                  300.
##  5 California               2096.
##  6 Colorado                  446.
##  7 Connecticut               551.
##  8 Delaware                   59.8
##  9 District Of Columbia      117.
## 10 Florida                   250.
## # ... with 41 more rows
```

Comparing these doesn't make much sense – obviously states with larger populations will have a higher count. So, instead of using the counts directly, we turn them into rates. We do this using the `mutate` function. This mutates the data by creating new variables. For example, to calculate the rate we can do this:

```
polio %>%
  mutate(rate = count / population)
```

```
## # A tibble: 3,774 x 7
##    disease state    year weeks_reporting count population     rate
##    <chr>   <chr>   <dbl>           <dbl> <dbl>      <dbl>    <dbl>
##  1 Polio   Alabama  1938              51  1843    2798382 0.000659
##  2 Polio   Alabama  1939              52  1975    2815188 0.000702
##  3 Polio   Alabama  1940              51  1235    2832961 0.000436
##  4 Polio   Alabama  1941              51  1566    2852063 0.000549
##  5 Polio   Alabama  1942              51  1640    2872445 0.000571
##  6 Polio   Alabama  1943              52  2054    2893949 0.000710
##  7 Polio   Alabama  1944              52  1314    2916414 0.000451
##  8 Polio   Alabama  1945              52  1081    2939676 0.000368
##  9 Polio   Alabama  1946              51  1048    2963562 0.000354
## 10 Polio   Alabama  1947              51  2249    2987897 0.000753
## # ... with 3,764 more rows
```

Notice how the data now has 7 columns (i.e. variables) instead of 6. You can check a few, if you'd like, to make sure that rate is indead count per population (`count / population`).

This can be directly piped into the `group_by` and `summarize` functions:

```
polio %>%
  mutate(rate = count / population) %>%
  group_by(state) %>%
  summarize(mean_rate = mean(rate))
```

```
## # A tibble: 51 x 2
##    state                mean_rate
##    <chr>                    <dbl>
##  1 Alabama              0.000115
##  2 Alaska              NA
##  3 Arizona              0.000446
##  4 Arkansas             0.000152
##  5 California           0.000243
##  6 Colorado             0.000309
##  7 Connecticut          0.000298
##  8 Delaware             0.000204
##  9 District Of Columbia 0.000167
## 10 Florida              0.0000864
## # ... with 41 more rows
```

Notice what happened to Alaska. We didn't get a number, but rather it simply writes `NA`. This is `R`'s way of telling us something is missing, or 'Not Available'/'Not Applicable'/'No Answer'. Wonder why? Let's take a look at just Alaska:

```
polio %>%
  filter(state == "Alaska")
```

```
## # A tibble: 74 x 6
##    disease state   year weeks_reporting count population
##    <chr>   <chr>  <dbl>           <dbl> <dbl>      <dbl>
##  1 Polio   Alaska  1938               0     0         NA
##  2 Polio   Alaska  1939               0     0         NA
##  3 Polio   Alaska  1940               0     0         NA
##  4 Polio   Alaska  1941               0     0         NA
##  5 Polio   Alaska  1942               0     0         NA
##  6 Polio   Alaska  1943               0     0         NA
##  7 Polio   Alaska  1944               0     0         NA
##  8 Polio   Alaska  1945               0     0         NA
##  9 Polio   Alaska  1946               0     0         NA
## 10 Polio   Alaska  1947               0     0         NA
## # ... with 64 more rows
```

We simply don't have data for the population in Alaska until 1960. Something similar is the case with Hawaii. We will simply exclude these two states from our futher analysis. To do so, we use the function `filter` together with `!=` (which means "not equal"). We also create the new variable `rate`

```
new_polio <- polio %>%
  filter(state != "Alaska", state != "Hawaii") %>%
  mutate(rate = count / population)

new_polio
```

```
## # A tibble: 3,626 x 7
##    disease state    year weeks_reporting count population     rate
##    <chr>   <chr>   <dbl>           <dbl> <dbl>      <dbl>    <dbl>
##  1 Polio   Alabama  1938              51  1843    2798382 0.000659
##  2 Polio   Alabama  1939              52  1975    2815188 0.000702
##  3 Polio   Alabama  1940              51  1235    2832961 0.000436
##  4 Polio   Alabama  1941              51  1566    2852063 0.000549
##  5 Polio   Alabama  1942              51  1640    2872445 0.000571
##  6 Polio   Alabama  1943              52  2054    2893949 0.000710
##  7 Polio   Alabama  1944              52  1314    2916414 0.000451
##  8 Polio   Alabama  1945              52  1081    2939676 0.000368
##  9 Polio   Alabama  1946              51  1048    2963562 0.000354
## 10 Polio   Alabama  1947              51  2249    2987897 0.000753
## # ... with 3,616 more rows
```

```
new_polio %>%
  filter(state == "Wisconsin", year == 1993)
```

```
## # A tibble: 1 x 7
##   disease state      year weeks_reporting count population      rate
##   <chr>   <chr>     <dbl>           <dbl> <dbl>      <dbl>     <dbl>
## 1 Polio   Wisconsin  1993              43   323    5013015 0.0000644
```

Now we have the rate for each state for each year. Which state has had the highest average rate of polio over the years? We can use `arrange` to basically sort the data by `mean_rate`:

```
new_polio %>%
  group_by(state) %>%
  summarize(mean_rate = mean(rate)) %>%
  arrange(mean_rate)
```

```
## # A tibble: 49 x 2
##    state        mean_rate
##    <chr>            <dbl>
##  1 Mississippi  0.0000192
##  2 Louisiana    0.0000400
##  3 Nebraska     0.0000695
##  4 Missouri     0.0000699
##  5 Georgia      0.0000761
##  6 Oklahoma     0.0000856
##  7 Florida      0.0000864
##  8 South Dakota 0.0000913
##  9 Nevada       0.0000924
## 10 Iowa         0.0000973
## # ... with 39 more rows
```

By default, it is sorted in increasing order. Luckily it is easy to change it to decreasing:

```
new_polio %>%
  group_by(state) %>%
  summarize(mean_rate = mean(rate)) %>%
  arrange(desc(mean_rate))
```

```
## # A tibble: 49 x 2
##    state          mean_rate
##    <chr>              <dbl>
##  1 Vermont         0.000829
##  2 Utah            0.000592
##  3 Wisconsin       0.000470
##  4 Arizona         0.000446
##  5 Maine           0.000362
##  6 Texas           0.000347
##  7 New Jersey      0.000334
##  8 New Mexico      0.000329
##  9 North Carolina  0.000328
## 10 Rhode Island    0.000324
## # ... with 39 more rows
```

One thing that could be interesting to look at is how the rate of polio has changed over the years. To do so, we calculate the mean rate per year:

```
new_polio %>%
  group_by(year) %>%
  summarize(mean_rate = mean(rate))
```

```
## # A tibble: 74 x 2
##     year mean_rate
##    <dbl>     <dbl>
##  1  1938  0.00176
##  2  1939  0.00158
##  3  1940  0.00130
##  4  1941  0.00154
##  5  1942  0.00133
##  6  1943  0.00130
##  7  1944  0.000864
##  8  1945  0.000850
##  9  1946  0.000710
## 10  1947  0.000987
## # ... with 64 more rows
```

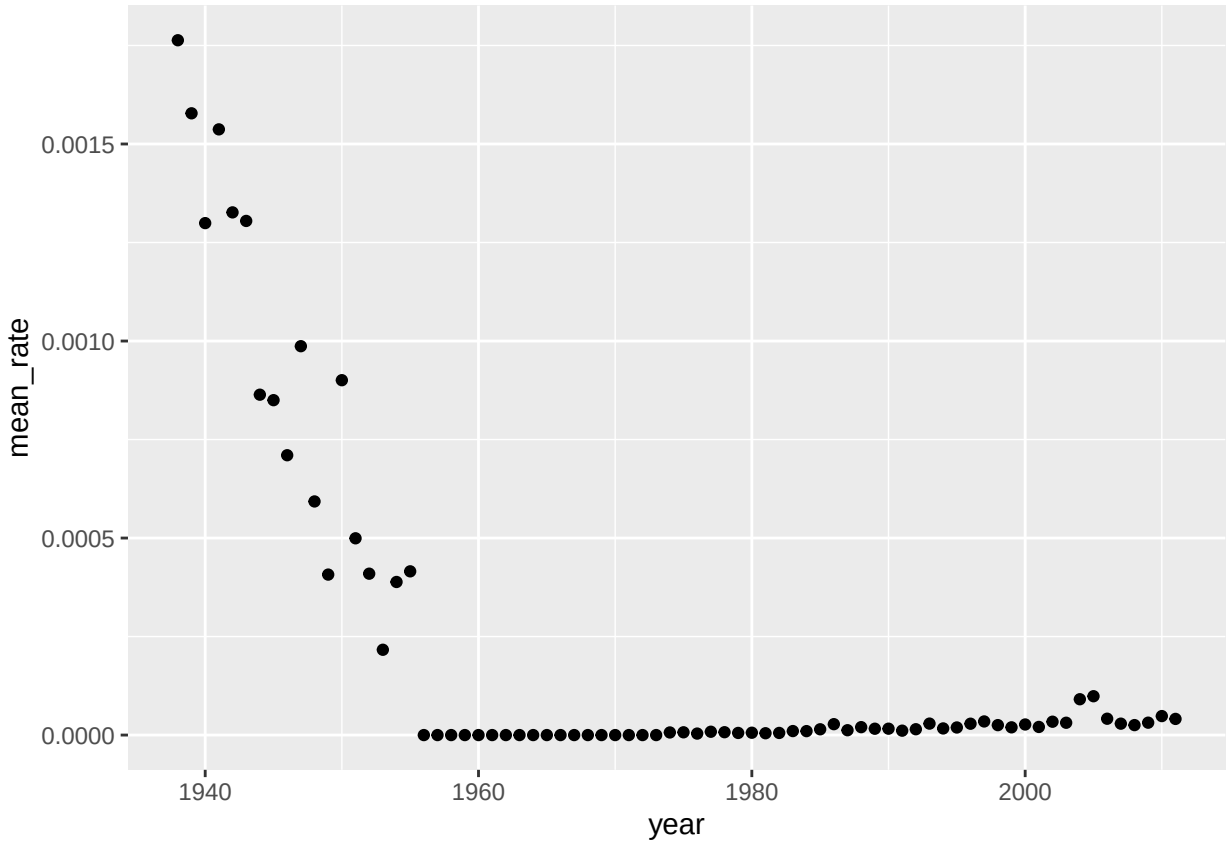
This long table of numbers is kind of boring... Let's create a plot! First, we will need to save the mean rates in a new object:

```
mean_rates_per_year <- new_polio %>%
  group_by(year) %>%
  summarize(mean_rate = mean(rate))
```

We then use the function `ggplot` to create a plot. We need to tell this function a few things:

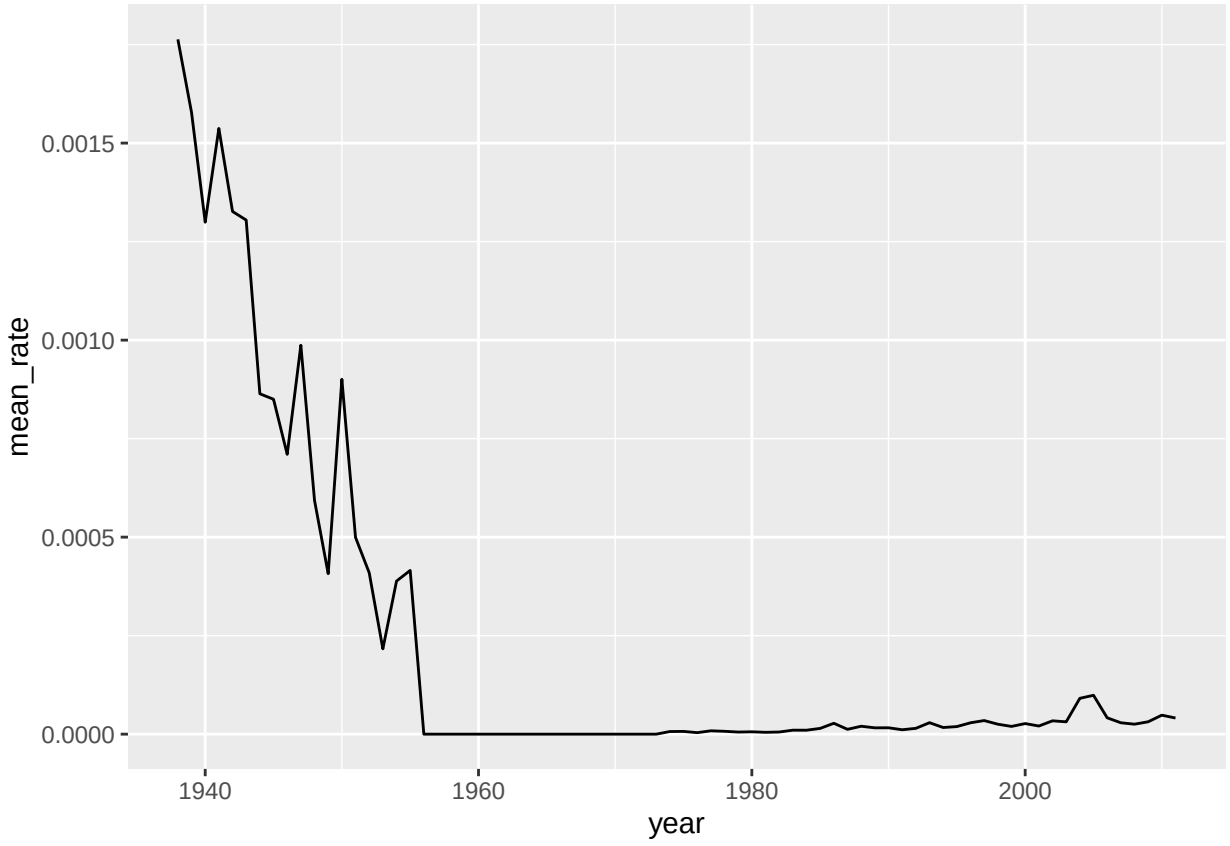
1. what data set to use
2. what aesthetics to use (`aes`)
3. what kind of plot we want

```
ggplot(mean_rates_per_year,
       aes(x = year, y = mean_rate)) +
  geom_point()
```



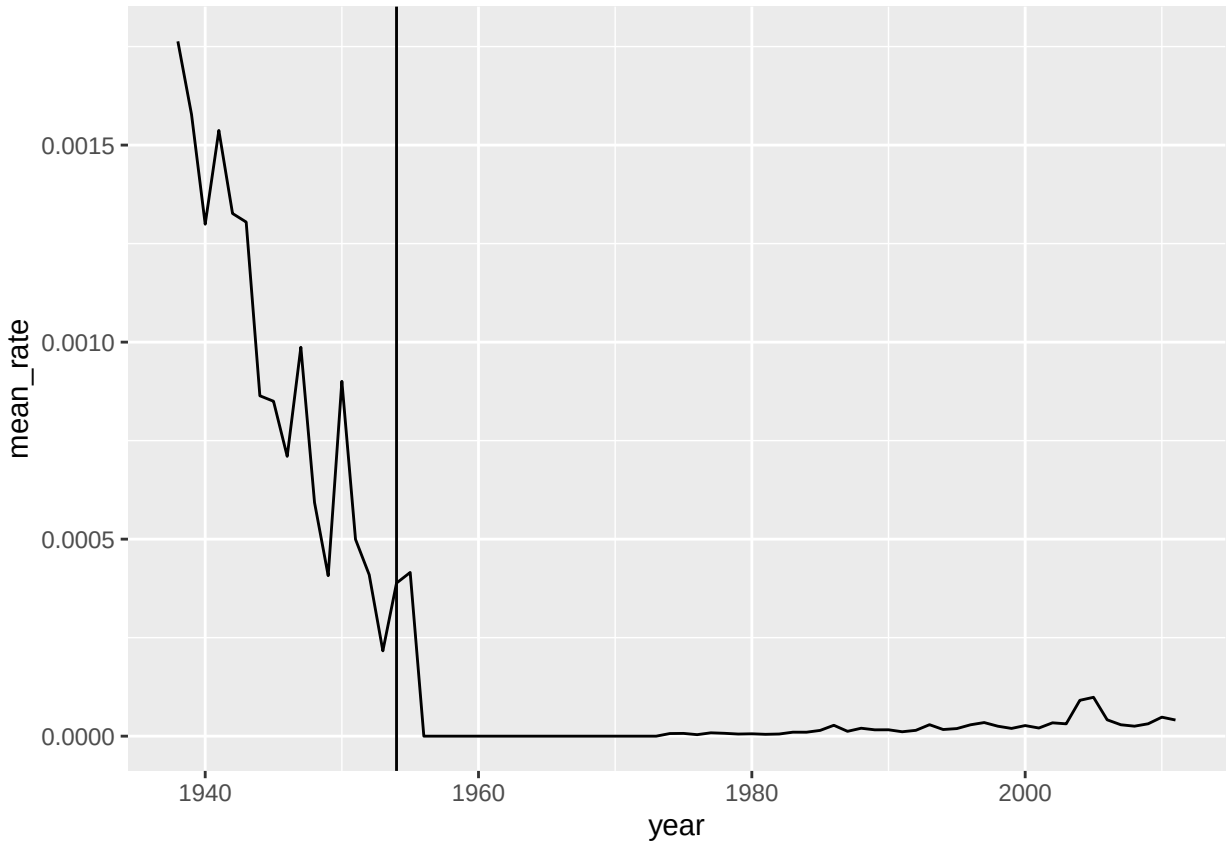
We could also have done this using a line instead of points:

```
ggplot(data = mean_rates_per_year,
       aes(x = year, y = mean_rate)) +
  geom_line()
```



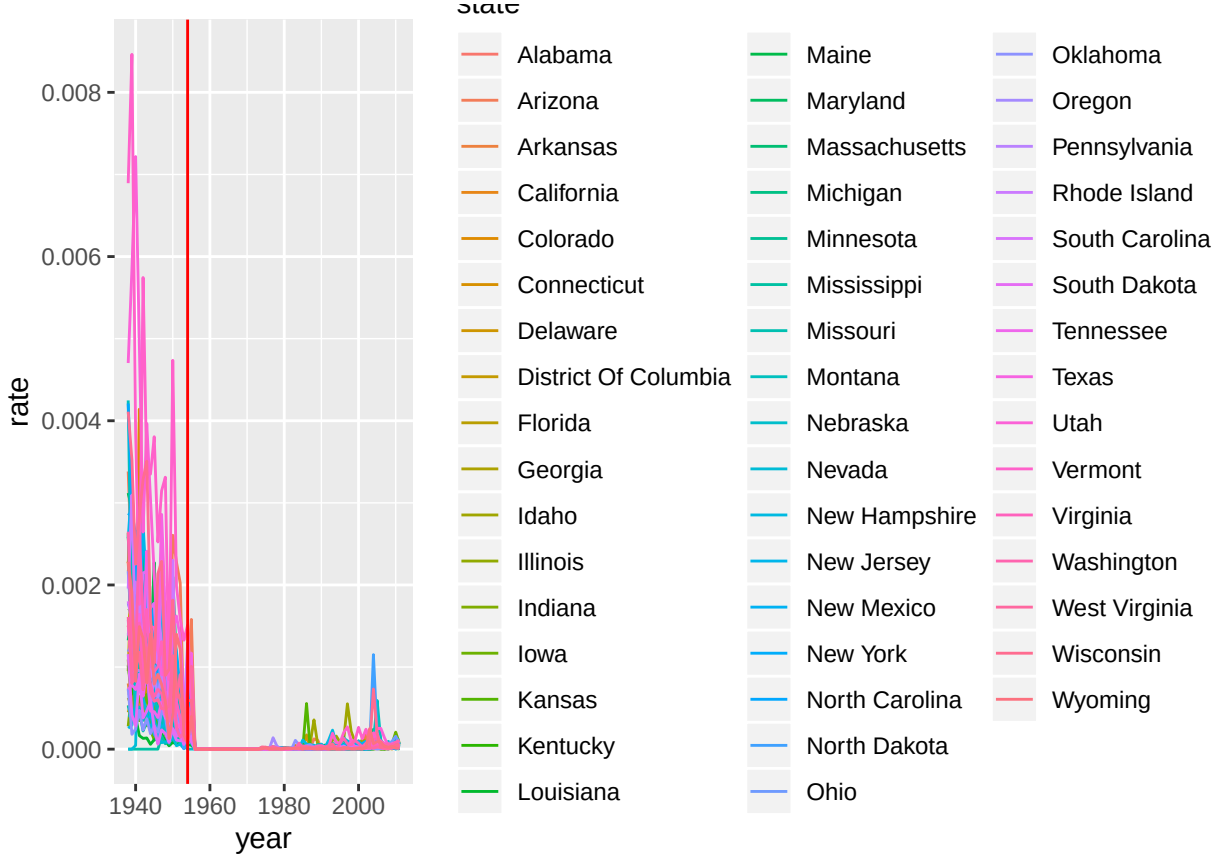
Something definitely happens around 1955. Turns out the polio vaccine became widely used in 1954:

```
ggplot(data = mean_rates_per_year,
       aes(x = year, y = mean_rate)) +
  geom_line() +
  geom_vline(xintercept = 1954)
```



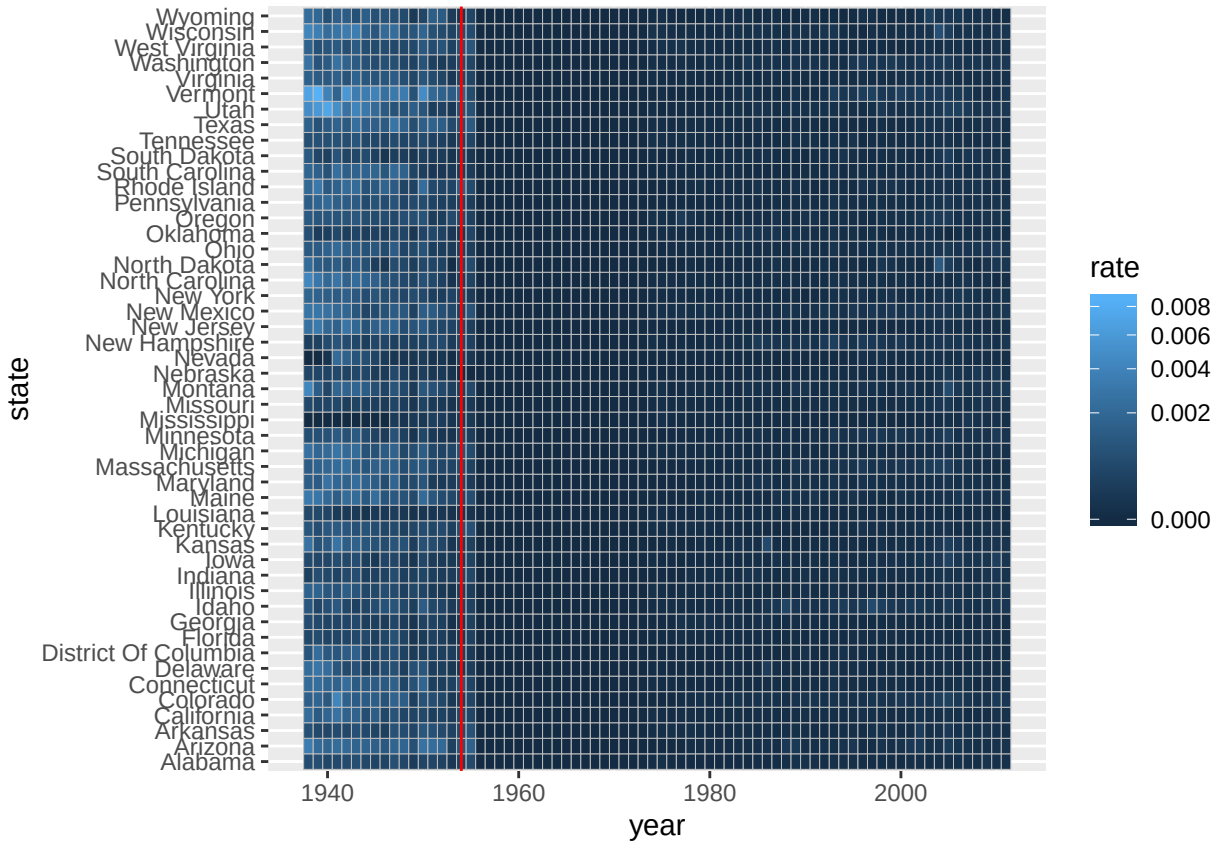
This was all combining all states. If we use the original data set, we can create the same plot, but with one line for each state:

```
ggplot(data = new_polio,
       aes(x = year, y = rate, color = state)) +
  geom_line() +
  geom_vline(xintercept = 1954, color = "red")
```



This is obviously not very useful. A much better way to present this data:

```
ggplot(data = new_polio,
       aes(x = year, y = state, fill = rate)) +
  geom_tile(color = "grey") +
  scale_fill_continuous(trans = "sqrt")  +
  geom_vline(xintercept = 1954, color = "red")
```



Take-aways

- import data using 'read_csv"
- extract single column using `$`
- create new variable using `mutate` and simple (or complicated, if you'd like) math
- calculate values per state/year using `group_by` and `summarize`
- filter data using `filter`
- create plots using `ggplot`
  - need to specify data, aesthetics, and type of plot (`geom_*`)

What's next? Discussion 1 and homework 1 will be exactly the same exercise, just with a different disease. Hopefully repetition will help you understand better what's going on.

PDF

To get a pdf out in the end, you will need to install a few more pieces. To do so, run the following lines of code in the console:

```
install.packages('tinytex')
tinytex::install_tinytex()
```

DISCLAIMER: this does not always work right away, and might take a few extra steps. If you're having issues, you can simply revert to using html as the output. If you really want to get it to work, ask Ralph.