Data modeling in R*

Learning outcomes

1. Create models from data
2. Store modeling information with data
3. Group and nest data for analysis

Introduction

This is the fourth in a series of five exercises that constitute *Training Module 1: Introduction to Scientific Programming*, taught through the IDEAS PhD program at the University of Georgia Odum School of Ecology in conjunction with the Center for the Ecology of Infectious Diseases.

This exercise explores methods for using modeling to shape our inquiry of data. It reaches back to previous modules that taught us about data manipulation, data visualization and functions.

The term "modeling" covers a vast array of ideas and techniques. A single module is always going to be rather modest in scope. Here we are going to focus more on statistical modeling than on mechanistic modeling (e.g. linear model fitting, not SIR modeling). However, even here we have to limit the scope. We're not really going to learn how to do statistical modeling (except very briefly). Rather, we're going to learn good practices for having our tidy data integrate with statistical modeling to help us perform exploratory analysis (hypothesis generation, not hypothesis confirmation).

```
library(tidyverse)
library(magrittr)
library(GGally)
```

Case study

We're going to work with the Lyme disease/Climate/Demography data set that you previously assembled.

Importing data

Task 1: Using either `read_csv` *(note: the underscore version, not* `read.csv`*) or* `load`*, import the data set you put together in the module on 'Data wrangling in R'.*

Using visualization to acquaint ourselves with our data

There's not a one-size-fits-all approach for how best to visually inspect and summarize your data. It depends what kind of data we're looking at - and you already learned some good ideas in the visualization module. For quantitative data, such as climate, demographic and disease case data, we might be interested in knowing the range of data, which kinds of values are rare and common, and whether data are correlated with each other.

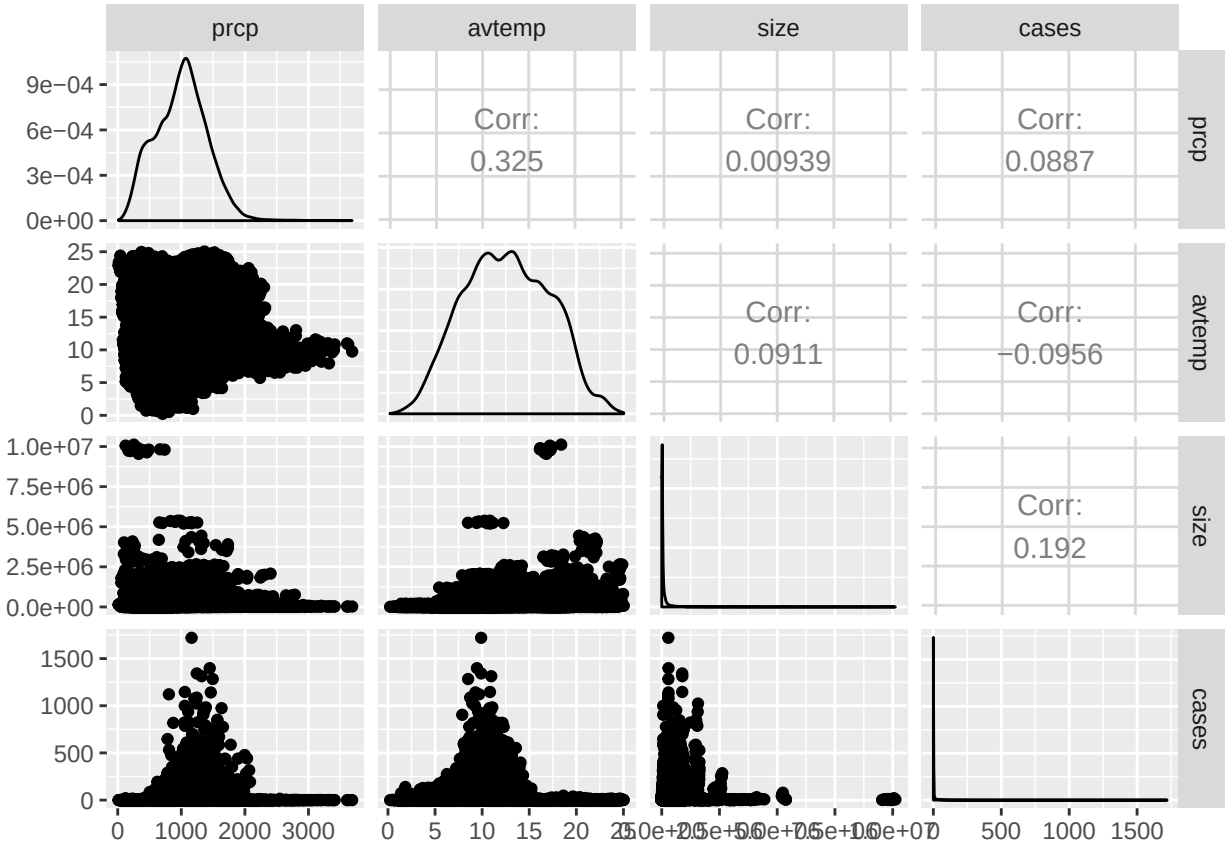
One way to do this is all in one go is with the function `ggpairs`, part of the `GGally` package, which has a language that is based on ggplot. For a data frame `df` where you're interested in numeric data columns `x`, `y`, and `z`, we load the GGally library and issue the command `ggpairs(df,columns=c("x","y","z"))`. This will make a 3x3 plot (because we have 3 columns: x, y and z). The main diagonals will display the density

*Contributions to lectures and practicals by Andrew W. Park, John M. Drake and Ana I. Bento

of the data (like a histogram, but continuous rather than binned). The lower triangle plots will show the correlation between each pair of data, and the upper triangle will report the correlation coefficient. The correlation coefficient is a number between -1 and +1, where numbers close to +1 (-1) indicate a strong positive (negative) correlation and numbers close to 0 indicate weak or no association. Note that `ggpairs` doesn't always display this way - it depends what kind of data you're visualizing.

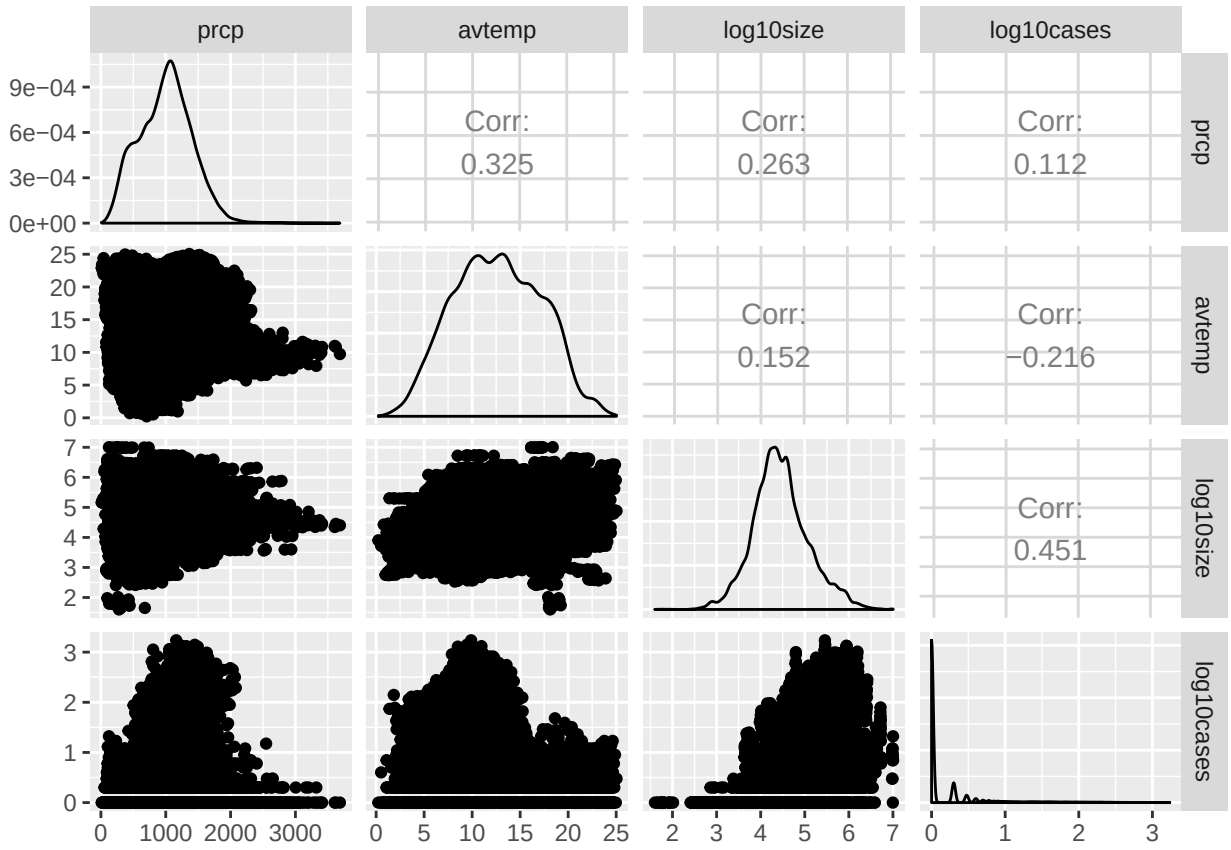
Task 2: Use the ***`ggpairs`*** *function to obtain a 4x4 summary plot of precipitation (prcp), average temperature (avtemp), population size (size), number of Lyme disease cases (cases). Note: it may take several seconds for this plot to appear as there are nearly 50,000 data points.*

```
ggpairs(ld.prism.pop,columns=c("prcp","avtemp","size","cases"))
```



You'll note from the density plots on the diagonals, that the data columns 'size' and 'cases' are very clumped, with many low values and a few large values. These may be easier to visualize by transforming to a logarithmic scale.

Task 3: Create two new columns for log10(size) and log10(cases+1) and substitute these for the original size and cases supplied when you recreate the ggpairs plot. Why do we add 1 to the number of cases?



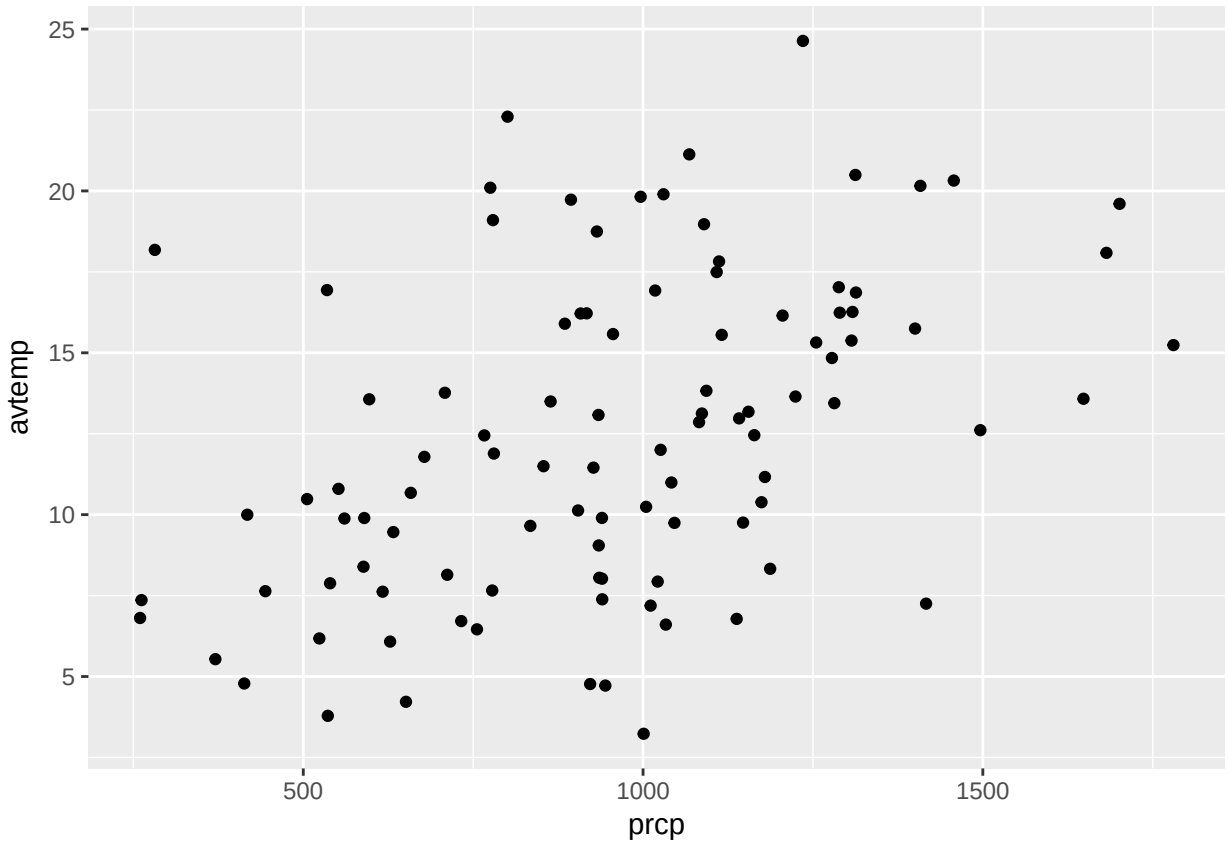
A simple linear model

Our `ggpairs` plot suggests that precipitation and average temperature are positively correlated with each other (perhaps not too surprising). Let's look at that for a random subset of the data (it's a bit easier to see that pattern when the data are thinned out).

Task 4: Using `set.seed(222)` for reproducibility, create a new data frame to be a random sample (n=100 rows) of the full data frame and plot precipitation (x-axis) vs average temperature (y-axis).

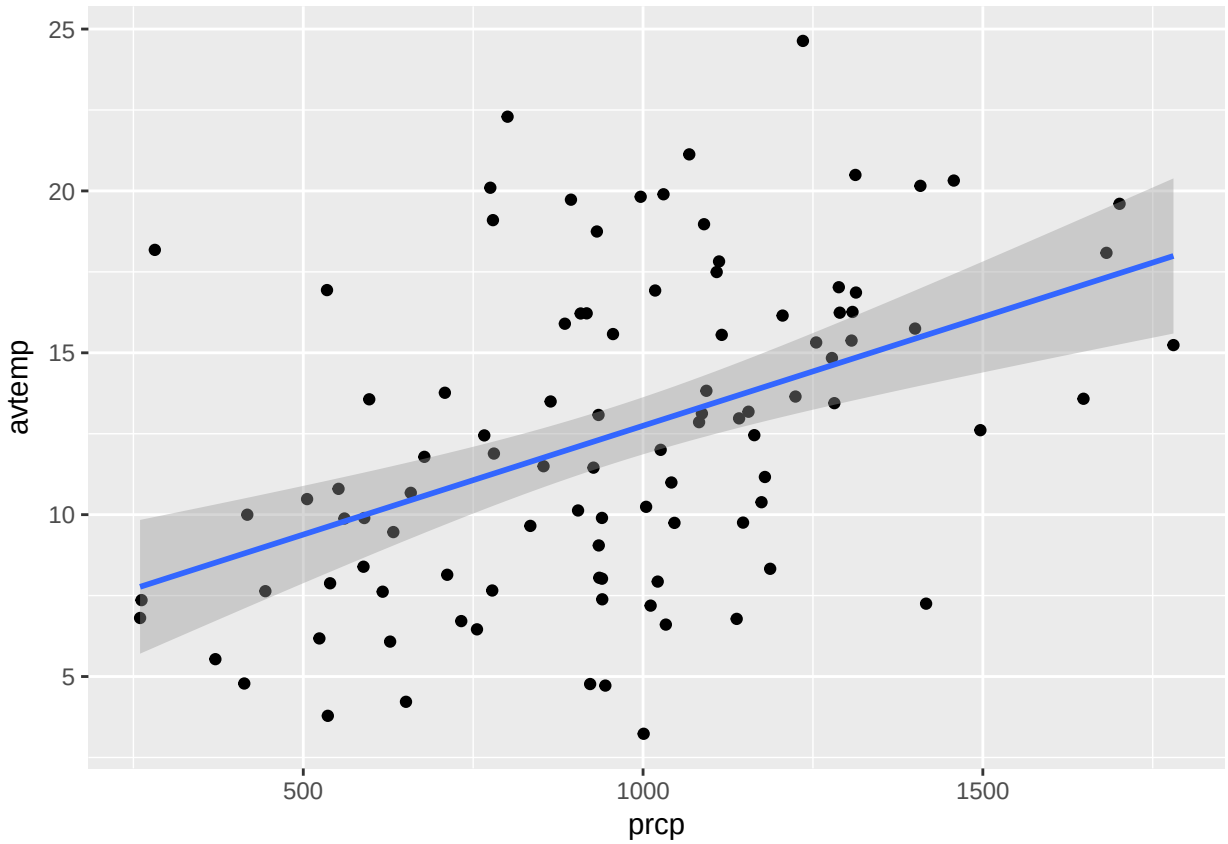
Hints:

- You can make use of the dplyr function `sample_n`.
- Name your ggplot (`myPlot <- ggplot...`) then call it (plot it) on a separate line (this will make it easy to add a subsequent layer to the plot)



Task 5: Add the best straight line to the plot using `geom_smooth`.

Hint: You'll need to use the appropriate `method` which you can find in the help file (`?geom_smooth`)



Further, we can store the linear model in memory and get a summary.

Task 6: Create a linear model (lm) object with a call like `myModel <- lm(y ~ x, data = myData)` *for the subsetted data, where y=avtemp and x=prcp. In addition, view the summary with a call along the lines of* `summary(myModel)`

```
##
## Call:
## lm(formula = avtemp ~ prcp, data = smallData)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9.5195 -3.2711 -0.3451  2.1937 10.8865
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  6.02541    1.36916   4.401 2.75e-05 ***
## prcp         0.00672    0.00136   4.942 3.19e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.397 on 98 degrees of freedom
## Multiple R-squared:  0.1995, Adjusted R-squared:  0.1913
## F-statistic: 24.42 on 1 and 98 DF,  p-value: 3.19e-06
```

We can extract information from this model object. For example, if your model object was actually called `myModel` (maybe not the best name; what happens if you have another model?) we can access the slope with `summary(myModel)$coefficients[2,1]` and the associated p-value with

```
summary(myModel)$coefficients[2,4]
```

Task 7: What is the slope of the line you plotted in Task 5, and is the slope significantly different from 0 ($p<0.05$)?

```
##     Estimate     Pr(>|t|)
## 6.720321e-03 3.190341e-06
```

The `modelr` package

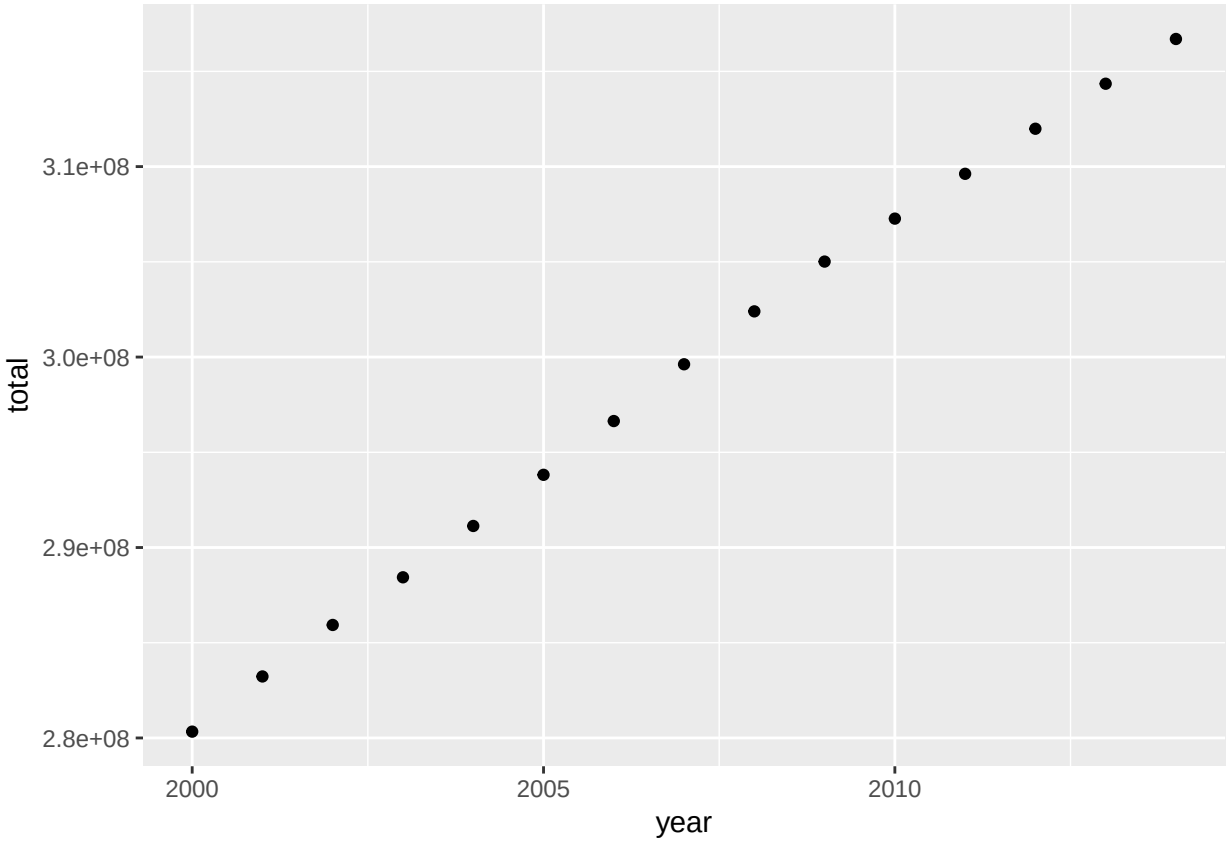
The `modelr` package is essentially a set of functions for modeling that are designed to help you seamlessly integrate modeling into a pipeline of data manipulation and visualization.

We'll start with an illustrative exercise. We know the size of the US population has been growing.

Task 8: Write a single line of code to generate a ggplot of total population size by year.

Hint: you should pass the main (large) data frame to a `group_by` call and then a `summarize` call, then you can pass this new, unnamed data frame to ggplot using `ggplot(.)` and specify the aesthetics in a `geom_point` call.

```
ld.prism.pop %>% group_by(year) %>% summarize(total=sum(size)) %>%
  ggplot(.)+geom_point(aes(x=year,y=total))
```



While there is no doubt that the population has been growing in recent years, it's not clear if all states are contributing equally to this growth. Manipulating data to explore this has the potential to get quite cumbersome, but the `modelr` tools are designed to make this kind of task fairly painless.

Grouped data frames versus nested data frames

We're going to create a nested data frame, which we do by first creating a grouped data frame (which you've already done in another context).

Task 9: Create a data frame called "by_state" from the main data frame, that groups by state, and inspect it.

```
## # A tibble: 46,630 x 11
## # Groups:   state [49]
##    state  county  cases  year  fips  prcp avtemp str_year   size log10size
##    <chr>  <chr>   <int> <dbl> <dbl> <dbl>  <dbl> <chr>     <int>     <dbl>
##  1 Alaba~ Autaug~     0 2000. 1001.  959.   18.1 pop2000   43872      4.64
##  2 Alaba~ Baldwi~     1 2000. 1003. 1019.   19.7 pop2000  141358      5.15
##  3 Alaba~ Barbou~     0 2000. 1005. 1006.   18.1 pop2000   29035      4.46
##  4 Alaba~ Bibb C~     0 2000. 1007.  994.   17.4 pop2000   19936      4.30
##  5 Alaba~ Blount~     0 2000. 1009. 1179.   16.3 pop2000   51181      4.71
##  6 Alaba~ Bulloc~     0 2000. 1011. 1068.   17.7 pop2000   11604      4.06
##  7 Alaba~ Butler~     0 2000. 1013. 1019.   18.7 pop2000   21313      4.33
##  8 Alaba~ Calhou~     0 2000. 1015. 1004.   16.1 pop2000  111342      5.05
##  9 Alaba~ Chambe~     0 2000. 1017. 1043.   16.4 pop2000   36593      4.56
## 10 Alaba~ Cherok~     0 2000. 1019. 1146.   15.6 pop2000   24053      4.38
## # ... with 46,620 more rows, and 1 more variable: log10cases <dbl>
```

Task 10: Next, update this new data frame so that it is nested (simply pass it to `nest`*). Again, inspect the data frame by typing its name in the console so see how things changed.*

```
by_state %<>% nest
by_state
```

```
## # A tibble: 49 x 2
##    state                data
##    <chr>                <list>
##  1 Alabama              <tibble [1,005 x 10]>
##  2 Arizona              <tibble [225 x 10]>
##  3 Arkansas             <tibble [1,125 x 10]>
##  4 California           <tibble [870 x 10]>
##  5 Colorado             <tibble [960 x 10]>
##  6 Connecticut          <tibble [120 x 10]>
##  7 Delaware             <tibble [45 x 10]>
##  8 District of Columbia <tibble [15 x 10]>
##  9 Florida              <tibble [1,005 x 10]>
## 10 Georgia              <tibble [2,385 x 10]>
## # ... with 39 more rows
```

You should see that by_state has a list-column called "data". List elements are accessed with [[]]. For example to see the data for Georgia, the 10th state in the alphabetized data frame, we would type `by_state$data[[10]]` in the console.

Task 11: Display the Georgia data in the console window.

```
## # A tibble: 2,385 x 10
##    county         cases  year   fips  prcp avtemp str_year  size log10size
##    <chr>          <int> <dbl>  <dbl> <dbl>  <dbl> <chr>    <int>     <dbl>
##  1 Appling County     0 2000. 13001.  965.   18.6 pop2000  17408      4.24
##  2 Atkinson Coun~     0 2000. 13003. 1055.   18.7 pop2000   7610      3.88
##  3 Bacon County       0 2000. 13005. 1054.   18.8 pop2000  10122      4.01
##  4 Baker County       0 2000. 13007.  963.   18.9 pop2000   4053      3.61
##  5 Baldwin County     0 2000. 13009.  866.   17.3 pop2000  44738      4.65
```

```
##  6 Banks County       0 2000. 13011.  993.   15.6 pop2000  14504      4.16
##  7 Barrow County      0 2000. 13013.  979.   15.9 pop2000  46561      4.67
##  8 Bartow County      0 2000. 13015. 1104.   15.6 pop2000  76703      4.88
##  9 Ben Hill Coun~     0 2000. 13017. 1102.   18.4 pop2000  17473      4.24
## 10 Berrien County     0 2000. 13019. 1083.   18.7 pop2000  16250      4.21
## # ... with 2,375 more rows, and 1 more variable: log10cases <dbl>
```

Hopefully you noticed that this particular element of the data frame is itself a dataframe! The containing column, which contains several such data frames differing in length due to the number of counties per state, is most usefully organized as a list - hence the column-list format. This method of organizing data comes in very useful for exploratory modeling, as we'll see. First we're going to create another function.

Task 12: Write a function that takes a data frame as its argument and returns a linear model object that predicts size by year.

```
linGrowth_model <- function(df){
  lm(size ~ year, data = df)
}
```

Next, we're introduced to one of the functions of `purrr`. This is a package that is installed as part of the `tidyverse`. When we use its functions - we need to make sure we have activated the library. The function we're interested in is called "map". To illustrate its potential, we can immediately apply a state-wise statistical modeling exercise (assuming you named the function of Task 12 `linGrowth_model`):

```
models <- purrr::map(by_state$data, linGrowth_model)
```

Function conflicts: why the `purrr::`?

Depending on what other libraries are loaded, we sometimes have be careful that we're calling the function we think we are. For example, if we have a geographical mapping library loaded, it's quite likely that it also has a function called `map`.

If you know this to be the case (e.g. the library called `maps` has a function called `map`), you can unload it, if it's currently loaded: `detach("package:maps", unload=TRUE)`. Alternatively, you can clarify a function call by preceding it with the package name, followed by two colons. For example `maps::map` or `purrr::map`.

Back to `purrr:map`

For data science good practice, it makes sense to put the model object fitted for each state with the appropriate state in the original data frame - not in a new data frame, like we just saw, as that requires extra coordination and possible mistakes.

Task 13: Add a column to the `by_state` dataframe, where each row (state) has its own model object.

```
by_state %<>% mutate(model = map(data, linGrowth_model))
```

Continuing in this format, we can, for example, store the residuals for each model (the discrepancy between the model prediction and the actual data). For this we will use the associated function to `purrr:map`, called `map2`, which takes 2 arguments and creates new data from them (in this case residuals).

```
library(modelr)
by_state %<>% mutate(resids = map2(data, model, add_residuals))
```

Task 14: Run these commands and inspect "resids". What is the structure of "resids"?

Task 15: Write a function that accepts an object of the type in the resids list, and returns a sum of the absolute values, i.e. ignoring sign: abs(3)+abs(-2)=5. Use the function to add a column called `totalResid` to `by_state` that provides the total size of residuals summed over counties and years.

```
sum_resids <- function(x){
  sum(abs(x$resid))
}
by_state %<>% mutate(totalResid = map(resids,sum_resids))
```

In addition, we can obtain and visualize the slope of population growth by state.

Task 16: Write a function that accepts a linear model and returns the slope (model `M` has slope `M$coefficients[2]`) and then use this function to create a new column called `slope` in the `by_state` data frame, that is the slope for each state.

```
get_slope <- function(model){
  model$coefficients[2]
}
by_state %<>% mutate(slope = purrr::map(model, get_slope))
```

While we're doing a great job of keeping our data organized, we've created another list-column (`slope` in data frame `by_state`). For visualization, we're going to want to un-nest these data structures...

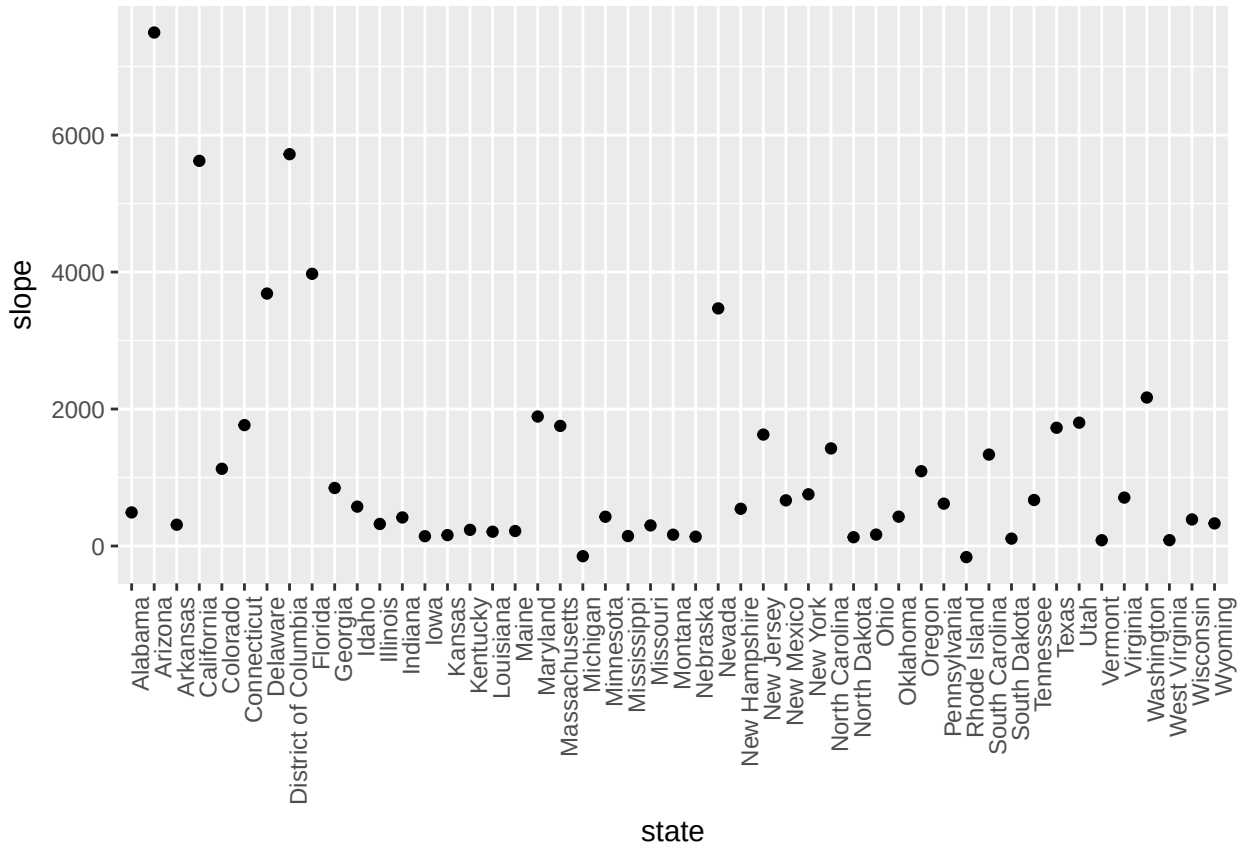
```
slopes <- unnest(by_state, slope)
totalResids <- unnest(by_state, totalResid)
```

Now we can pass these new data frames to ggplot to see how the growth rate manifested in different states.

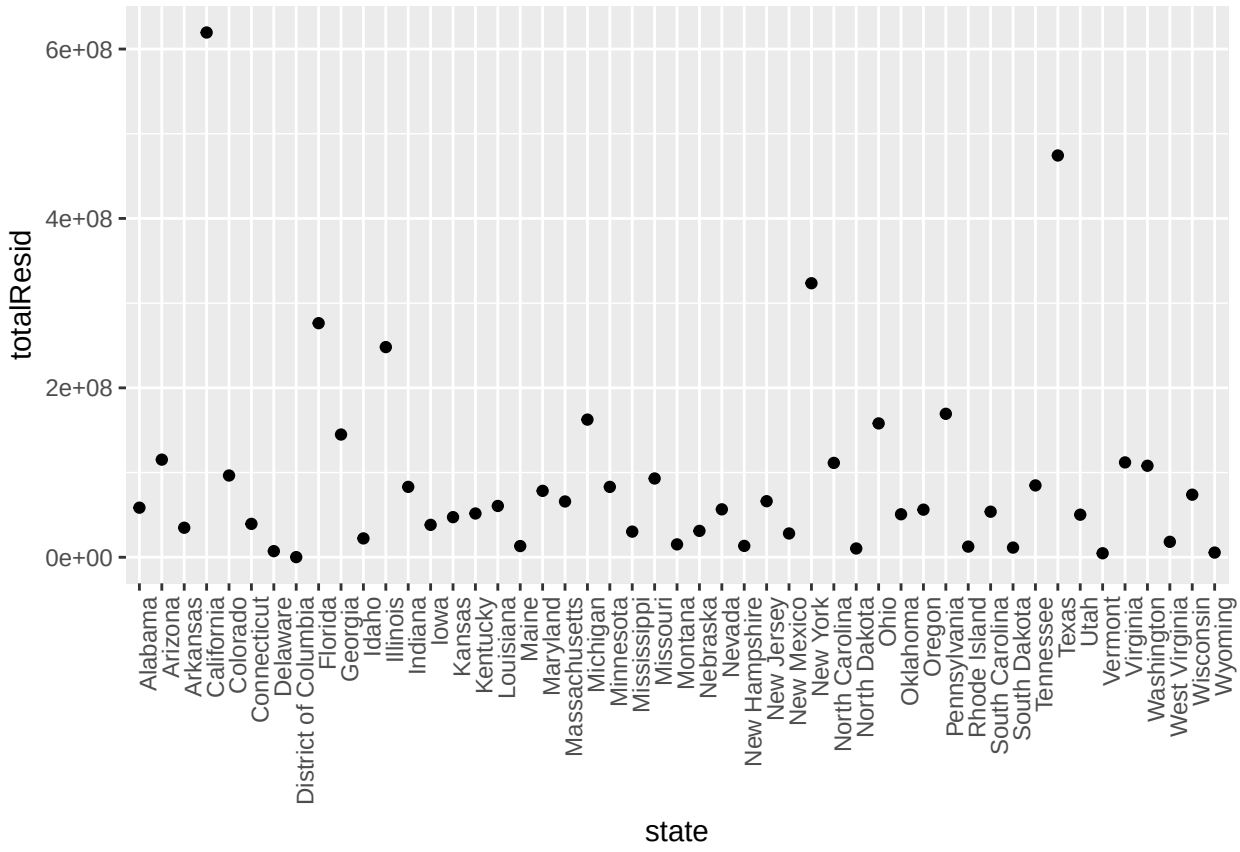
Task 17: Plot the growth rate (slope value) for all states.

Hint: If the states (x-axis) are hard to read, we can rotate them to be vertical by adding `+ theme(axis.text.x = element_text(angle = 90, hjust = 1))` to the ggplot layers.

```
slopes %>% ggplot(aes(state,slope))+geom_point()+
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



Task 18: Plot the total resisduals for all states.



This manipulation helps us to determine, which states are growing quickly, and which state's growth is relatively reasonably described by a linear model.

Task 19: Repeat Tasks 9 and 10 using a different data frame name, ***by_state2****.*

Task 20: Write a function that accepts an element of the ***by_state2$data*** *list-column and returns the spearman correlation coefficient between Lyme disease cases and precipitation*

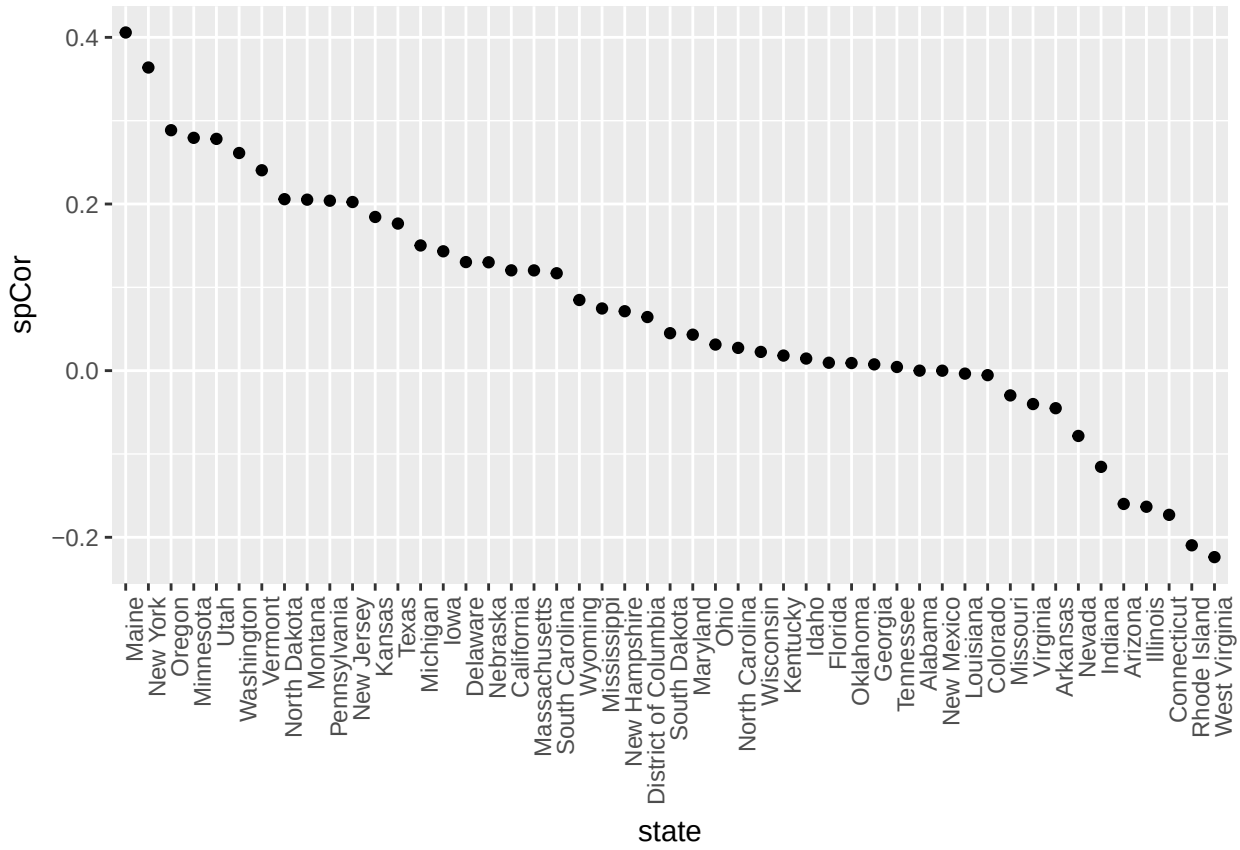
Hints:

- Use ?cor.test for general help with correlation tests
- Append "$estimate" to the end of the cor.test call to pull out the correlation coefficient
- Wrap the entire thing in "supressWarnings()" to prevent R repeatedly warning you about computing p-values with ties (which is not a big deal)

```
runCor <- function(df){
  suppressWarnings(cor.test(df$cases,df$prcp,method="spearman")$estimate)
}

by_state2 %<>% mutate(spCor = purrr::map(data, runCor))

spCors <- unnest(by_state2,spCor)
spCors %<>% arrange(desc(spCor))
spCors$state <- factor(spCors$state, levels=unique(spCors$state))
ggplot(spCors,aes(state,spCor))+geom_point()+
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



You've come a long way in importing and visualizing data and in keeping complex modeling information well organized to help you pick up on important features. While we only looked at a few examples, they introduce you to the style of combining data with modeling inquiries. Your own research can make use of these good practices, which will become routine allowing you to focus more on the research questions.