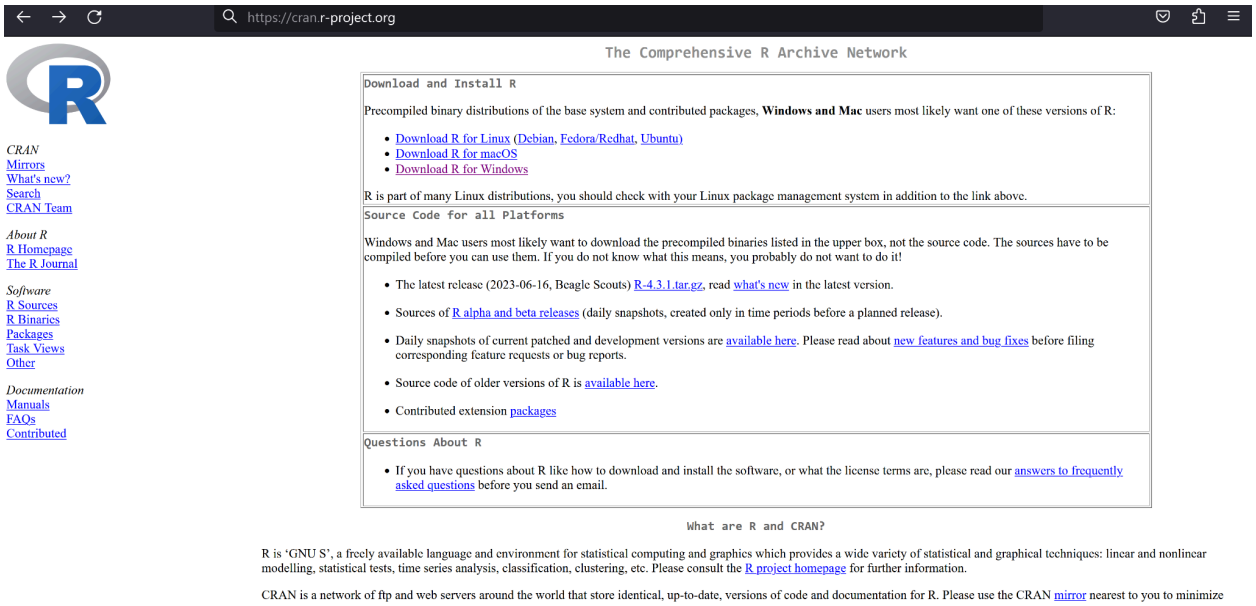
The goal of this very short workshop is NOT to become a statistician! I just want to give you some exposure to using R. Most importantly, you should gain some confidence in exploring what it has to offer on your own. While you definitely need teachers and mentors to walk you through coding, you also need to be comfortable with finding things out by yourself. The toughest roadblocks are often how to get started and not knowing how to look things up and hopefully this will help you overcome some of that!

Pre Class

Downloading the Software

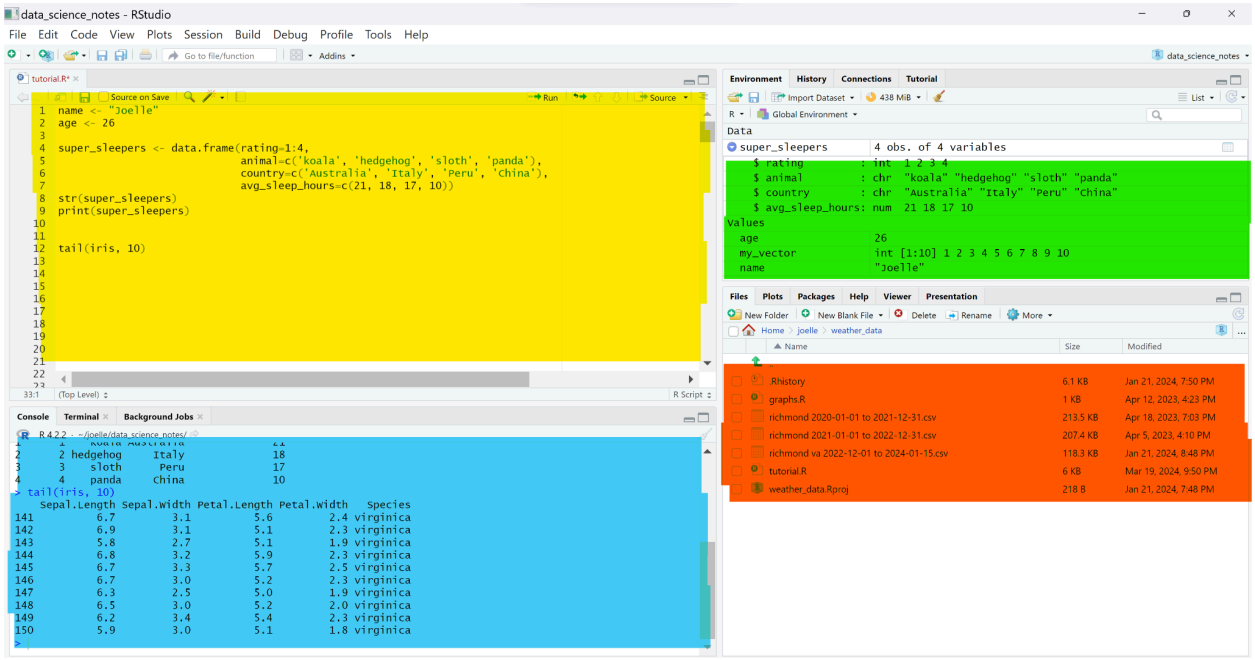
R vs R Studio: What's the difference? R is a programming language designed for statistical analysis. RStudio is an Integrated Development Environment (IDE) that helps you program in R. An IDE is a software application that provides tools for software development, and normally consists of a code editor, build automation tools, and a debugger. You can use R without RStudio but RStudio makes working with R easier (most people who use R prefer RStudio).

Information about installing R and RStudio: https://rstudio-education.github.io/hopr/starting.html Link to download: https://cran.r-project.org/. Make sure you are downloading the correct version for your operating system (Mac, Windows, or Linux but probably not Linux). Use recommended settings and remember where your software is located on your computer (try to choose a meaningful location but you can always just search it in your files each time you want to use it).



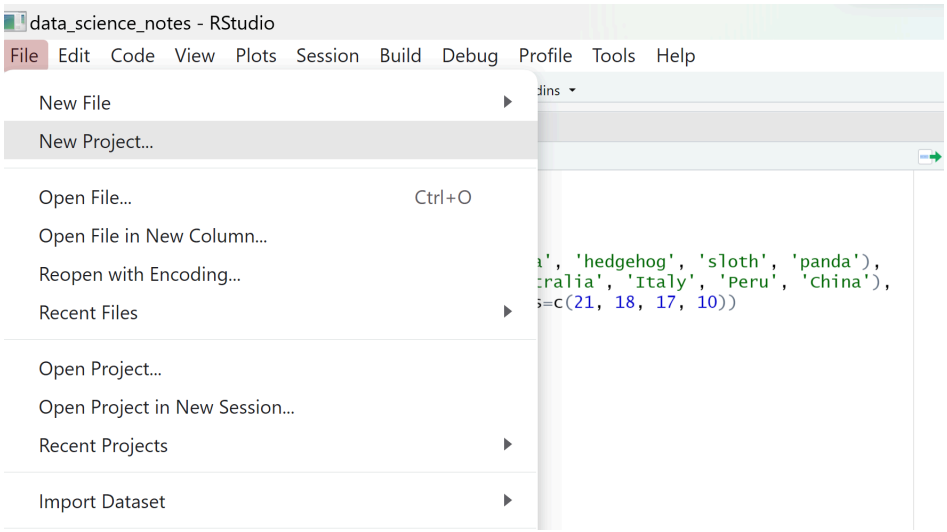
Parts of the RStudio IDE:

In yellow, we have the **source** pane. The source pane is where you create and edit R Scripts" - your collections of code. To have R actually evaluate your code, you need to first ‘send’ the code to the console. To execute your code and “send” it to the console you can highlight the code you wish to evaluate and click on the “Run” button on the top right of the source pane. Alternatively, you can use the hot-key “Command + Return” on Mac, or “Control + Enter” on PC to send all highlighted code to the console.The **console** pane is blue. This is where R actually evaluates code. The **environment** tab is green and it shows you the names of all the data objects (like vectors, matrices, and dataframes) that you’ve defined in your current R session. You can also see information like the number of observations and rows in data objects. And finally, the **output** console is in red. This is where you can see files, packages, and any plots you create.

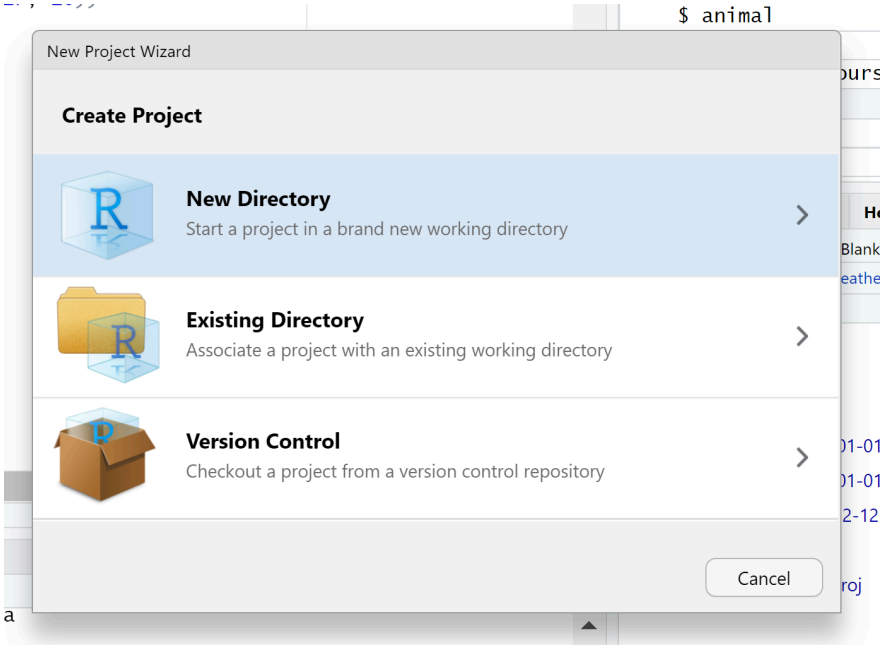


Creating your first project

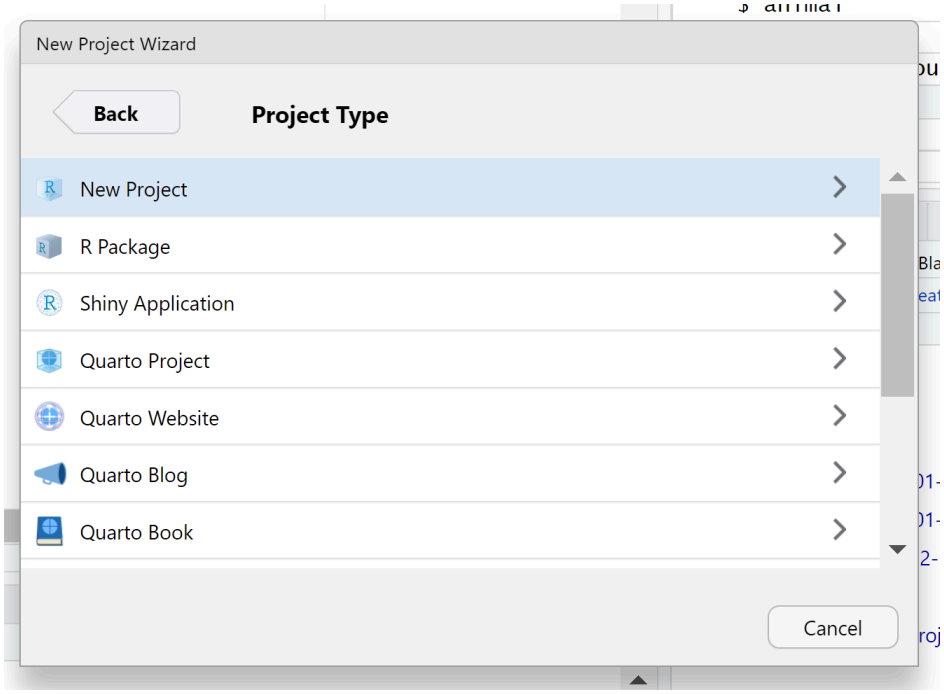
Let’s create a new R project from scratch. First open RStudio. In File click “New Project”.



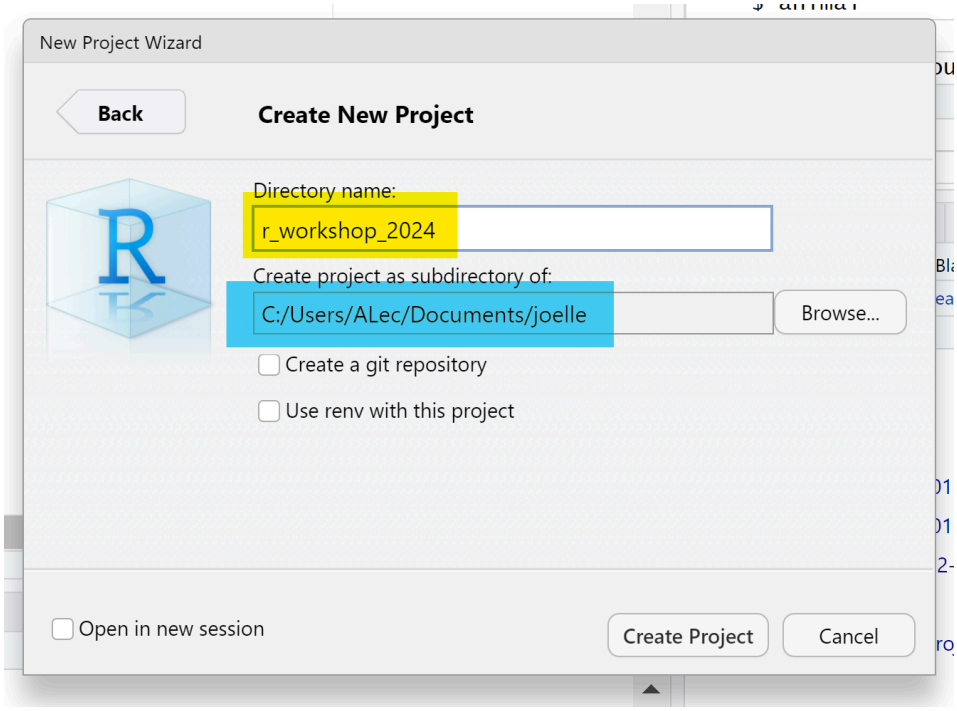
You will be prompted with different options. We will create a new directory. A directory is simply a folder on your computer.



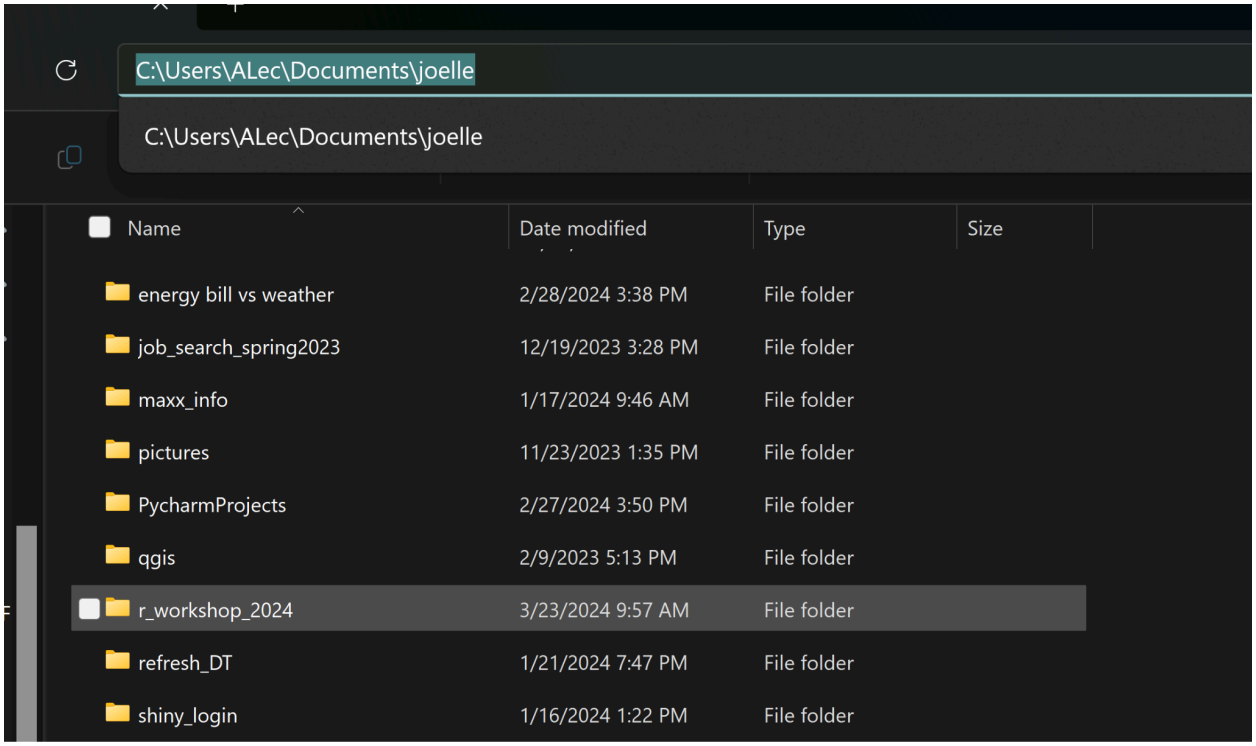
Next you will be asked what type of project. We are creating a new project.



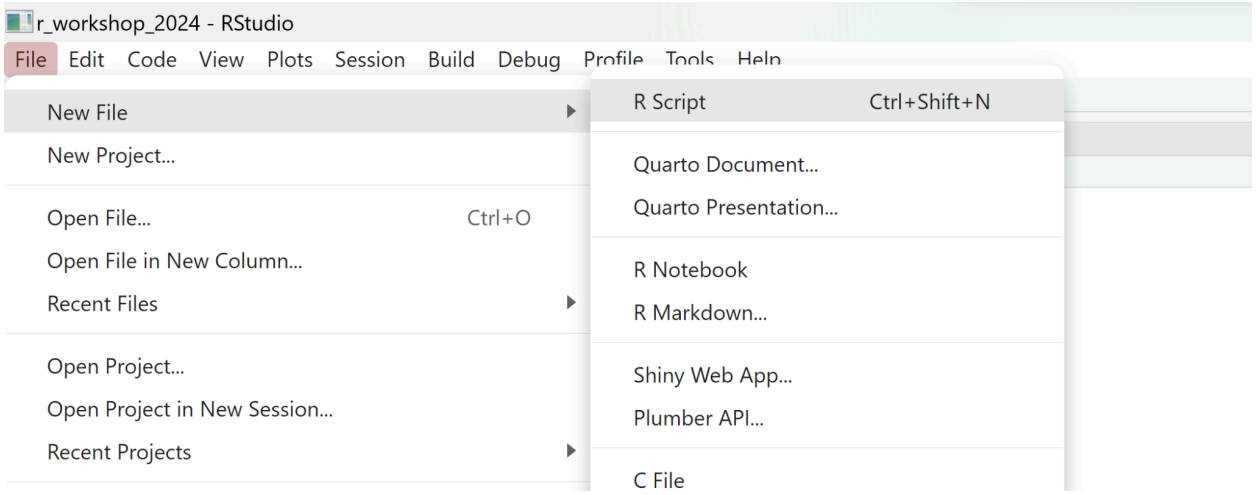
This next step is very important. Set the directory name. Again the directory is just a folder. So when you look through your files on your computer this is the folder name you will be looking for each time you try to open this workshop. Make sure you know where this folder is being saved! I have placed mine inside documents inside a folder called joelle. Once I click “Create Project” R has now created a new folder called “r_workshop_2024” within the “joelle” folder.



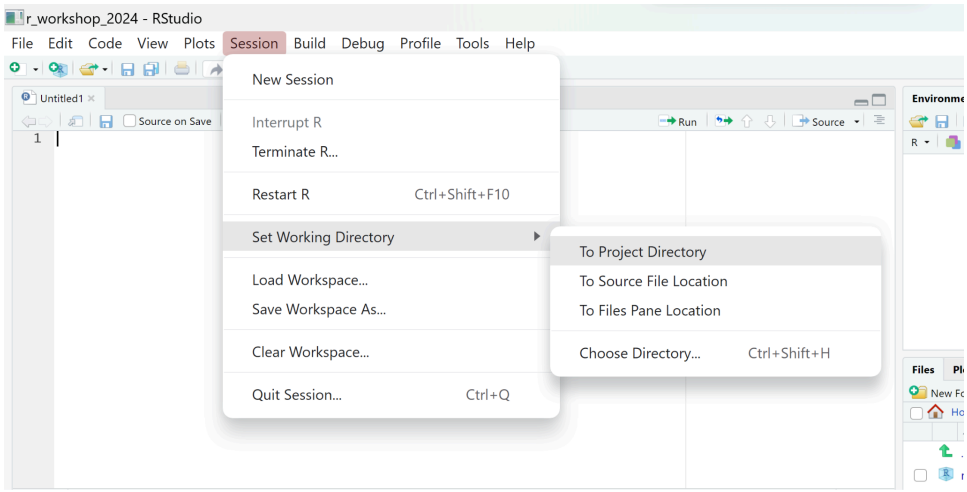
Check that you understand where your folder is on your computer. I went into my file explorer and navigated to my “joelle” folder. Inside there is now a new folder called “r_workshop_2024”.



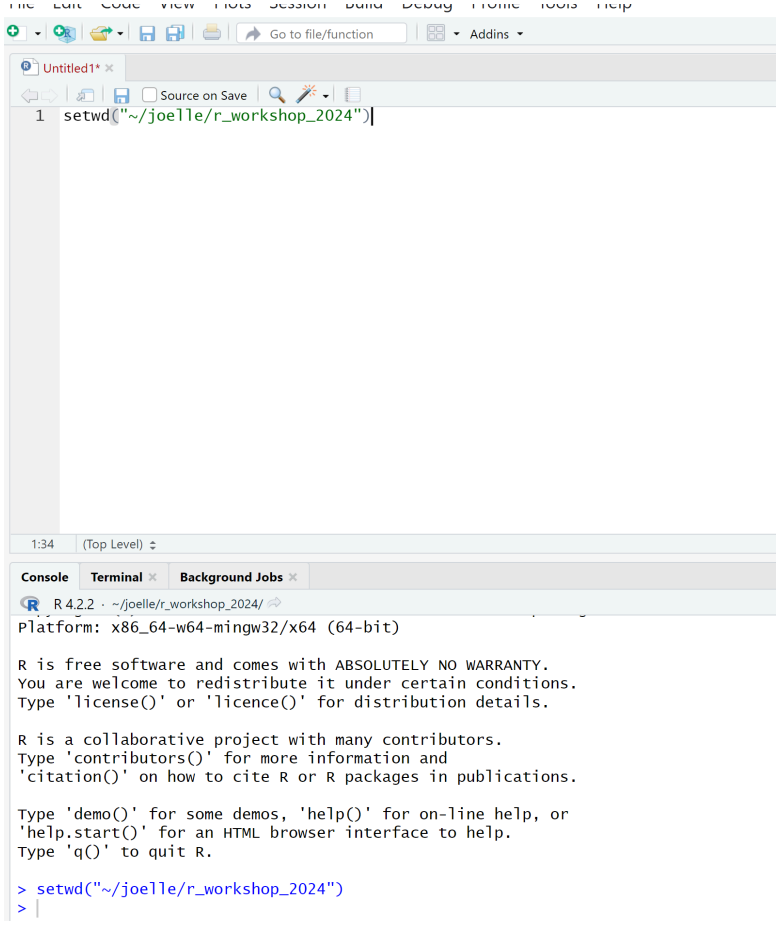
Now you can go back into RStudio and create an R Script. Remember this is where you will put your actual code. It is another file that will also go inside your “r_workshop_2024” folder.



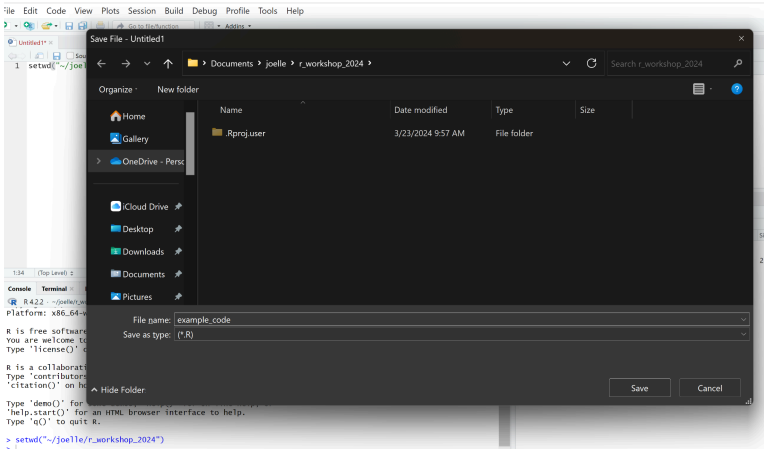
It is generally good practice to set a working directory. This is a folder you give to R to set as the default location to look for files you want to load and use. It is also the default location for any files you create in R and save to your computer. Go into session to set the working directory. Since we just created a project, set the working directory to project directory.



After you set the working directory, the code that performs this action appears in the console. You should copy and paste it into your R file so you can rerun the code and reset the working directory in case anything is disrupted.



Your R file is still called “Untitled1”. Save it and call it something meaningful. Generally the more exact your names are the better. This is important when you look back at old code so you can easily identify what you were doing in the past. I called this “example_code” but you should call it something different.

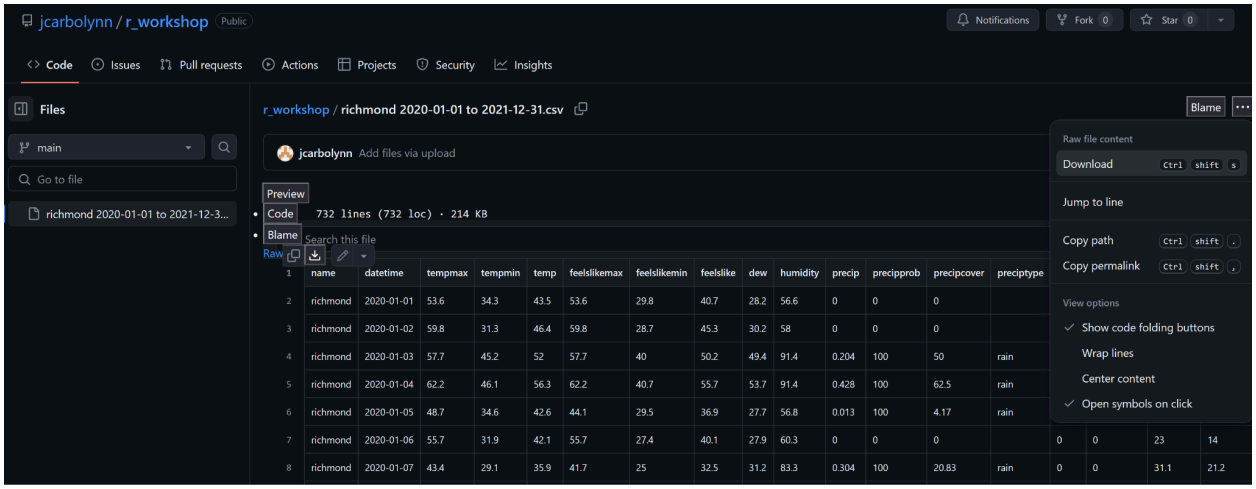


When there are new changes to your R file, the file name will turn red and this indicates you have unsaved changes.

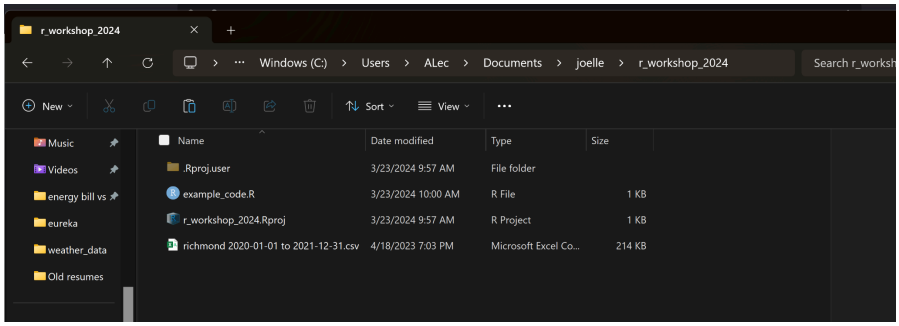
Downloading Data

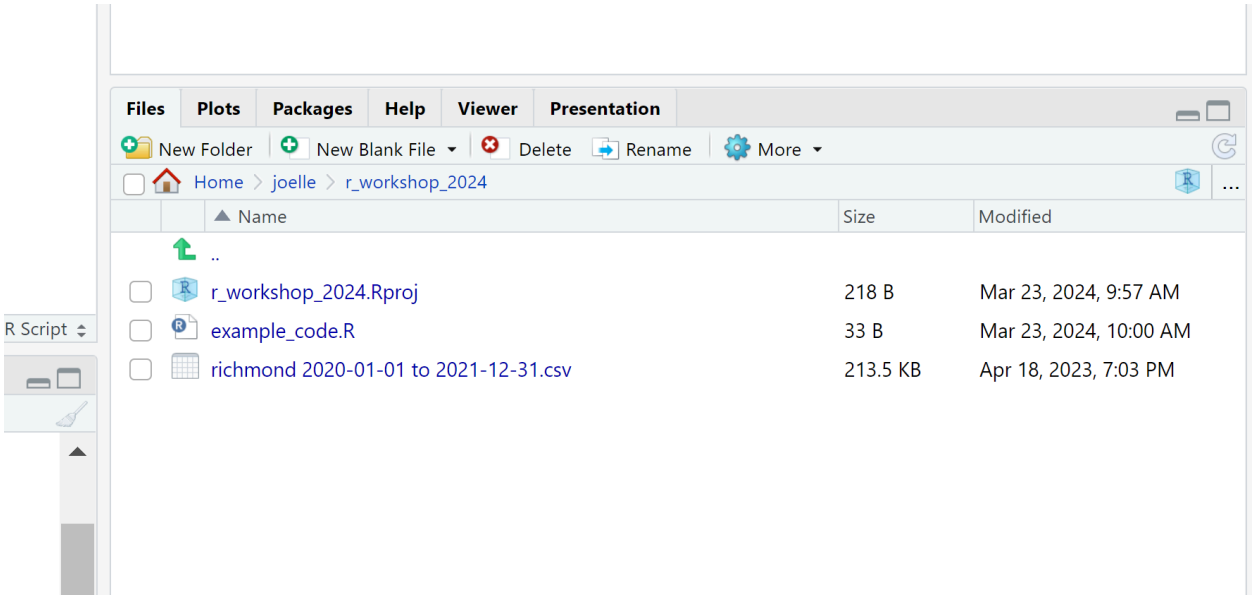
- Download example data and example code from github
- https://github.com/jcarbolynn/r_workshop

Once you go to the link, click on the csv file and download.



After downloading the example data (richmond 2020-01 to 2021-12-31.csv) I moved it to my project folder. After doing that, it shows up in my output console (bottom right window of RStudio).





Assignment 0:

******You MUST have all of these done before the first session******

- Download R and RStudio
- Create an R project called “r_workshop_2024”
- Create an R file in your R project and name it something
- Download example weather data from github
- Download example code from github

Session 1

Packages/Libraries

Packages or **Libraries** are collections of R functions, data, and compiled code in a well-defined format, created to add specific functionality. There are 10,000+ user contributed packages and growing.

There are a set of **standard (or base) packages** which are considered part of the R source code and automatically available as part of your R installation. Base packages contain the **basic functions** that allow R to work, and enable standard statistical and graphical functions on datasets; for example, all of the functions that we have been using so far in our examples.

You don't need to know a ton about these. Just understand that they exist and we will use them a few times.

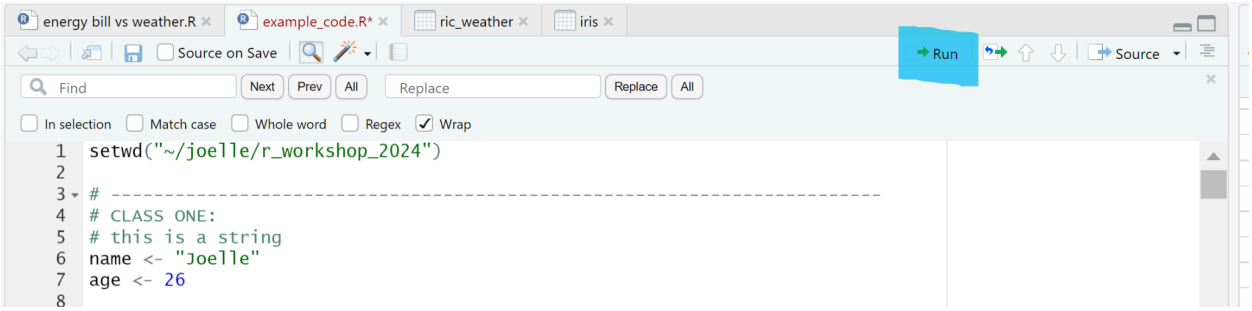
Comments

A very important part of coding is writing comments. In R a comment is written with a "#" in front of the comment. This lets the computer know that everything after is not code and should not be "run" or executed. Distinguishing between comments and code is important because you have to use a programming language when interacting with a computer as it will not understand plain English.

Because programming requires you to use a different language, it can be confusing to look over someone else's code. And especially if you are unfamiliar with what they are doing. Sometimes even something you wrote in the past is tricky to translate if you have forgotten what certain parts do. It is best practice to include comments to explain your code and make it more readable. You can also use comments to prevent execution when testing alternative code.

Executing Code

You can go to a line of code and click the run button in the top right of the source panel. You can also highlight multiple lines of code and run those all together. Or you can use a keyboard shortcut 'ctrl' + 'enter' for Windows or 'command' + 'enter' for Mac after highlighting the code you want to run.

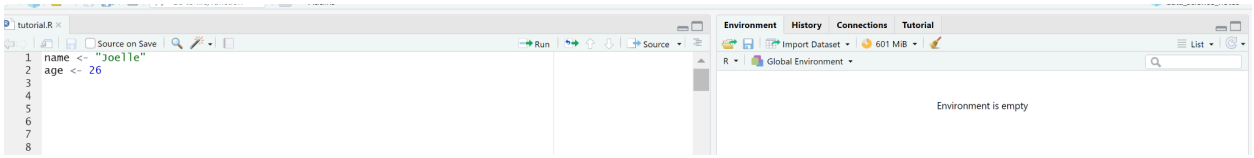


Creating Variables

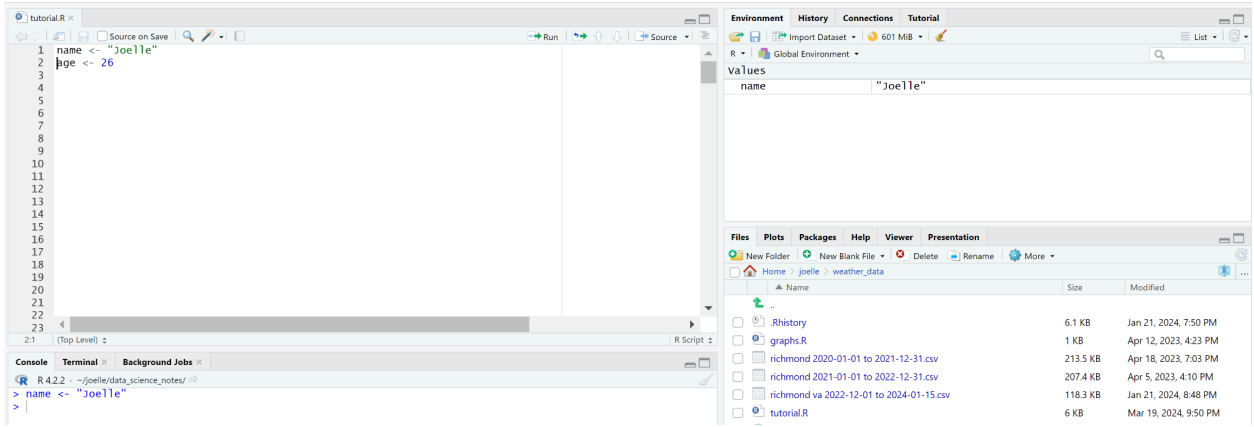
Variables are containers for storing data values. Just like in math class when you had to find what ‘x’ was. X was just a placeholder with an actual value. In R you assign a variable its value with “<-” or “carrot”

- my_integer <- 5
- This variable is called “my_integer” and it stores the value 5

Once you “run” a line of code, it will be executed and a couple of things will happen.



In the console in the bottom left part of RStudio, you will see that it was executed. When you run into errors this is where they will be displayed! In the top right, in the environment you will see a list of all your variables.



Data Types and Data Structures

Data Types:

- Logical/Boolean: true or false statement

```
bool1 <- TRUE
print(bool1)
print(class(bool1))

bool2 <- FALSE
print(bool2)
print(class(bool2))
```

Output

```
[1] TRUE
[1] "logical"
[1] FALSE
[1] "logical"
```

- Numeric: it's what you think it is, just a number (including all real numbers with or without a decimal); you can do math with these

```
# floating point values
weight <- 63.5
print(weight)
print(class(weight))

# real numbers
height <- 182
print(height)
print(class(height))
```

Output

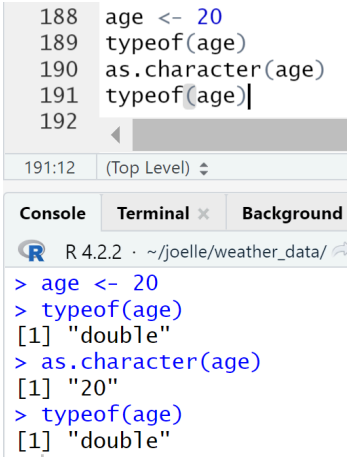
```
[1] 63.5
[1] "numeric"
[1] 182
[1] "numeric"
```

  - Float: refers to numbers with a decimal point
  - Integer: written with an L after the number; does not contain a decimal

```
integer_variable <- 186L
print(class(integer_variable))
```

Output

```
[1] "integer"
```

- String: you can think of it as a word or phrase; anything between the “” (quotes)
    - fruit <- “Apple”
    - my_sentance <- “This is a string”
- Character: a single symbol; can be any letter/number/punctuation but just one of them; generally denoted by single quotes ‘’
    - a_char <- ‘a’
    - number_as_char <- ‘9’
        - if it is between single quotes, R will understand it as a character and not a number, you cannot do math with it
- Factor: used to store categorical data with a limited number of different values; storing data as factors ensures that the modeling functions will treat such data correctly
    - Ex: You have groups that are identified by a number. If you want to look at differences between groups, R must understand the group number as a category and not as an integer.
- Casting: you can convert one type of data to another.
    - 

    - FINISH THIS

Data Structures:

- Vector: a list of variables (of any type), the simplest data structure in R. When creating a vector you must use a ‘c’ before the ().
    - vector1 <- c(5,9,3)
    - vector2 <- c(10,11,12,13,14,15)
    - anyname <- c(1,2,3,5,8,11)

```
13
14  my_vector <- (1, 1, 2, 3, 5, 8)
15
16
17
33:1   (Top Level)
```

```
Console   Terminal   Background Jobs
R 4.2.2 · ~/joelle/data_science_notes/
> my_vector <- (1, 1, 2, 3, 5, 8)
Error: unexpected ',' in "my_vector <- (1,"
>
```

- You can see that R will catch the error in your script (the red x indicates a syntax error) and when you attempt to execute broken code an error will appear in the console.

- List: a list is very similar to a vector except the datatypes of each element can be different.

```
21
22  my_vector <- c(1,4,"hello", TRUE)
23  str(my_vector)
24  my_list <- list(1,4,"hello", TRUE)
25  str(my_list)
26
27
25:13   (Top Level)
```

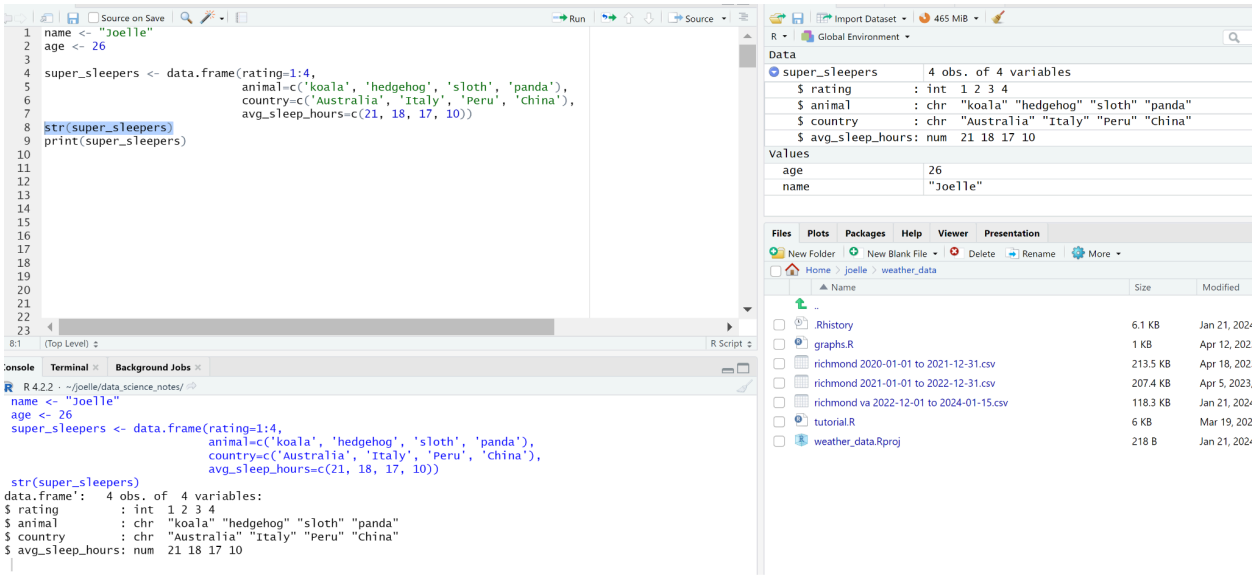
```
Console   Terminal   Background Jobs
R 4.2.2 · ~/joelle/data_science_notes/
150          5.9          3.0          5.1
> my_vector <- c(1,4,"hello", TRUE)
> str(my_vector)
 chr [1:4] "1" "4" "hello" "TRUE"
> my_list <- list(1,4,"hello", TRUE)
> str(my_list)
List of 4
 $ : num 1
 $ : num 4
 $ : chr "hello"
 $ : logi TRUE
>
```

- When you attempt to create a similar grouping as a vector, you can see that R understands them all as strings instead of numbers, a character, and a boolean.

- Matrix: 2D data structure, think of it as a table; data are arranged into rows and columns

- Dataframe: similar to a matrix except you can include a mix of data types. Data types in one column all have to be the same, but you can have a column for “names” that contains strings and then a column for “ages” that contains numbers. The difference is similar to list vs vector (vector is to list as matrix is to data frame)
    - Typically we are using vectors and data frames

Here I have created a simple dataframe called “super_sleepers”. A dataframe is basically a table. You can see that in the environment, clicking on the blue arrow will show more information about the dataframe. I also ran str(super_sleepers) which will show the structure of the variable. It can be very useful to see a quick snapshot of your data.



Running print() will show your entire variable. But for variables containing a lot of information this can be less than ideal. For the small “super_sleepers” dataframe, everything is easily displayed. But iris has many more rows and it runs off the screen.

```
> print(super_sleepers)
  rating   animal   country avg_sleep_hours
1      1    koala Australia              21
2      2 hedgehog     Italy              18
3      3    sloth      Peru              17
4      4    panda     China              10
>
```

```
> print(iris)
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
3            4.7         3.2          1.3         0.2     setosa
4            4.6         3.1          1.5         0.2     setosa
5            5.0         3.6          1.4         0.2     setosa
6            5.4         3.9          1.7         0.4     setosa
7            4.6         3.4          1.4         0.3     setosa
8            5.0         3.4          1.5         0.2     setosa
9            4.4         2.9          1.4         0.2     setosa
10           4.9         3.1          1.5         0.1     setosa
11           5.4         3.7          1.5         0.2     setosa
12           4.8         3.4          1.6         0.2     setosa
13           4.8         3.0          1.4         0.1     setosa
```

Another good way to quickly view your data is by taking a look at the first few rows of a dataframe. You can do this by using head() to show the first 6 rows, or tail() to show the last 6. Adding a number after a comma will allow you to choose how many rows are displayed.

```
> tail(iris, 10)
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
141          6.7         3.1          5.6         2.4 virginica
142          6.9         3.1          5.1         2.3 virginica
143          5.8         2.7          5.1         1.9 virginica
144          6.8         3.2          5.9         2.3 virginica
145          6.7         3.3          5.7         2.5 virginica
146          6.7         3.0          5.2         2.3 virginica
147          6.3         2.5          5.0         1.9 virginica
148          6.5         3.0          5.2         2.0 virginica
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
>
```

Operators:

- You know all the usual ones (+,-,*,/)
- You can also use exponents
- You can round
- There are comparisons like >,<,=
    - Some new ones might be
    - => and <= these are less than OR equal to and greater than OR equal to. So 5 >= 5 is true while if you only had > you'd need 6 > 5 to make this statement true
    - == is checking if two values are the same. So 5 == 5 is true

Assignment 1:

- Add comments to code sample
- Practice investigating variables
    - use str() a LOT
    - bonus: can you find other ways to investigate your variables? Can you figure out how to use attributes()?
- Find an interesting R package and try reading some documentation
    - Do not expect to fully understand it! Looking at documentation sucks! Did you use a different site from rdrr.io or cram?

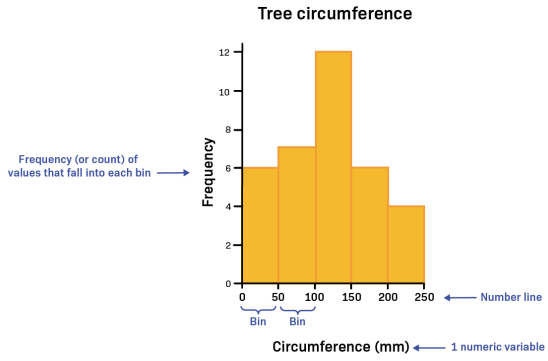
Session 2

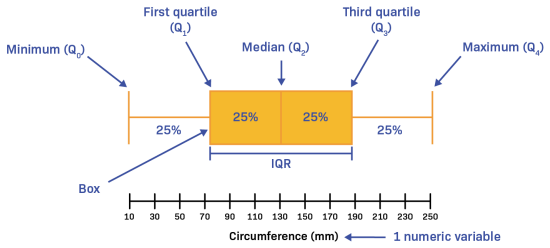
Graphs

There are many different types of graphs and each is best used for a specific purpose. Visualizations in general are the best way to start understanding your data. You can look for relationships in the data (does height seem to change as weight does?) or look at the scale of

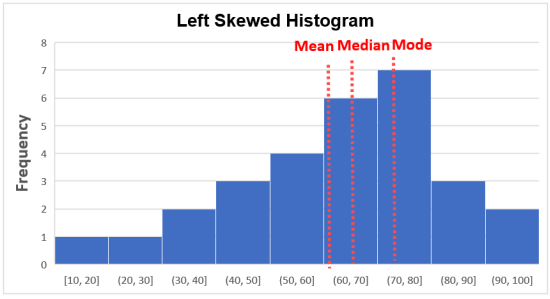
your variables (height could be 4ft-6ft, weight 80 lbs to 200 lbs) or look for outliers (is there someone who is recorded as being 12 ft tall? That is a mistake. Sometimes outliers are less obvious for example if there is someone 80 lbs but they are 6ft. That is probably incorrect but both weight and height are within the expected ranges.)

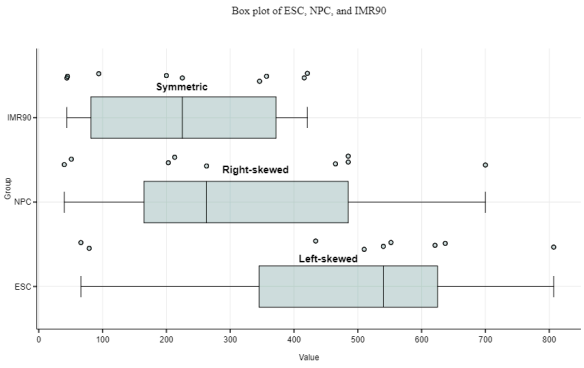
Histograms and **Boxplots** are generally good places to start when first looking at data. A histogram shows how many of a certain type of variable occur within a specific range. For example, it will show that there are 20 people 0-5 years old, 23 people 6-10, and so on. A boxplot shows the five-number summary: the minimum, the maximum, the sample median, and the first and third quartiles (25% of the data falls below or above these values).





Both provide a visual means to assess the central tendency (average), the amount of variation in the data, as well as the presence of gaps, outliers or unusual data points. Histograms are particularly useful for determining the distribution of the data.

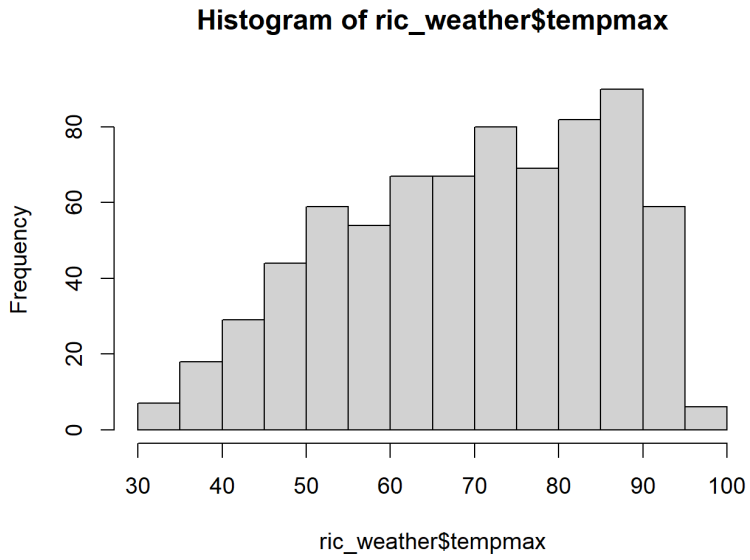




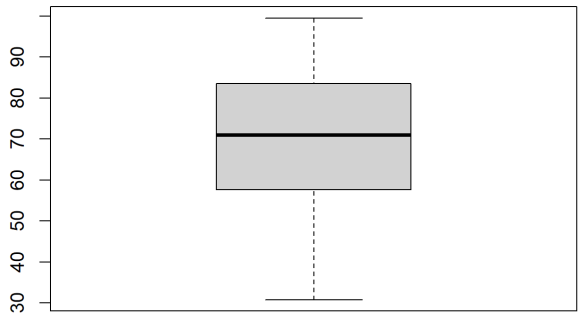
From this histogram you can tell that the mean is lower than the median. Since there are a few “extreme” values that are low, this pulls the mean down. The median is not as affected by outliers so it stays higher. Boxplots similarly show unevenness in data.

Examples in R using the weather data:

```r
# make a histogram of max temps using base R
hist(ric_weather$tempmax)
```



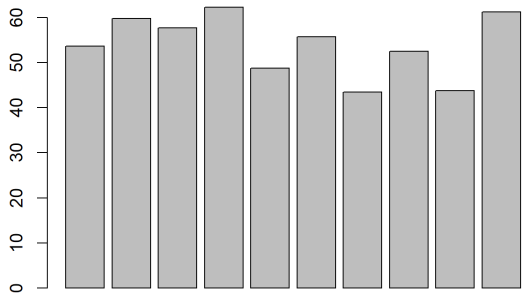

```r
# make boxplot of max temps
boxplot(ric_weather$tempmax)
```



Barcharts are useful for showing differences between categorical data. It presents categorical data with rectangular bars with heights (or lengths) proportional to the values they represent.

Examples in R using the weather data:

```r
# make barplot of max temps, only first 10 rows
barplot(ric_weather[1:10, "tempmax"])
```



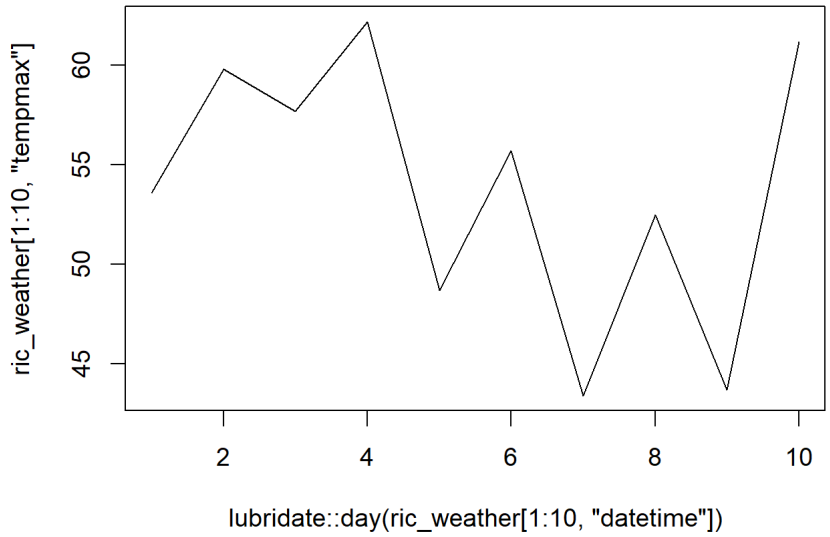

```
# look at the data the barplot is using
ric_weather[1:10, "tempmax"]
head(ric_weather, 10)
```

```
> ric_weather[1:10, "tempmax"]
 [1] 53.6 59.8 57.7 62.2 48.7 55.7 43.4 52.5 43.7 61.2
> head(ric_weather, 10)
       name   datetime tempmax tempmin temp feelslikemax f
1  richmond 2020-01-01    53.6    34.3 43.5         53.6
2  richmond 2020-01-02    59.8    31.3 46.4         59.8
3  richmond 2020-01-03    57.7    45.2 52.0         57.7
4  richmond 2020-01-04    62.2    46.1 56.3         62.2
5  richmond 2020-01-05    48.7    34.6 42.6         44.1
6  richmond 2020-01-06    55.7    31.9 42.1         55.7
7  richmond 2020-01-07    43.4    29.1 35.9         41.7
8  richmond 2020-01-08    52.5    31.5 39.6         52.5
9  richmond 2020-01-09    43.7    24.5 34.2         42.4
10 richmond 2020-01-10    61.2    33.0 47.2         61.2
```

You can see that this is only using the first 10 rows from the “tempmax” column to create this barplot.

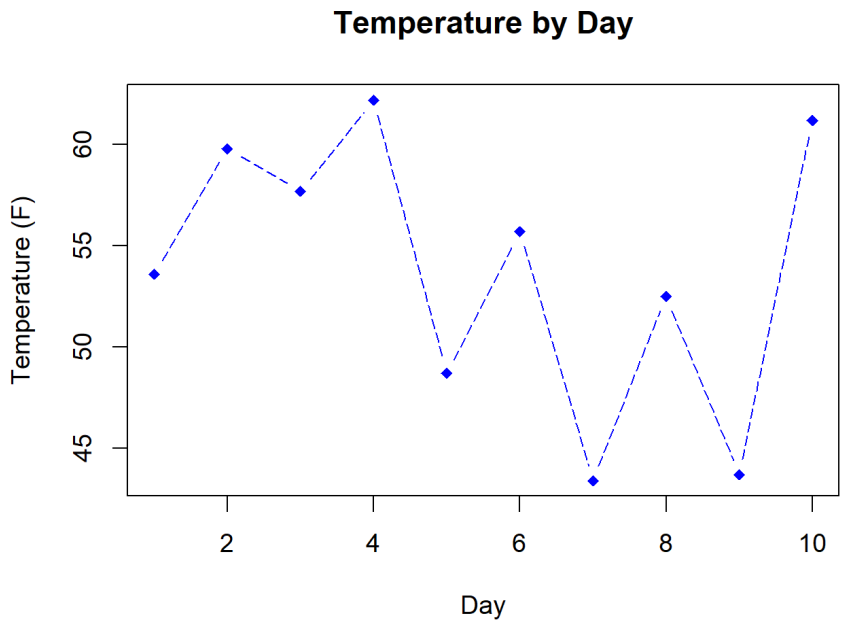
Line graphs show how information changes over time. The x-axis is usually some measure of time (in hours, days, years, etc).

```
# make line graph
library(lubridate)
plot( lubridate::day(ric_weather[1:10, "datetime"]), ric_weather[1:10, "tempmax"], type = "l")
```



These graphs show the information, but they do not present it nicely. Luckily even base R provides lots of options to customize your graphs. The most important things to add are axis titles and main title. I added some things such as type, pch, col, lty.

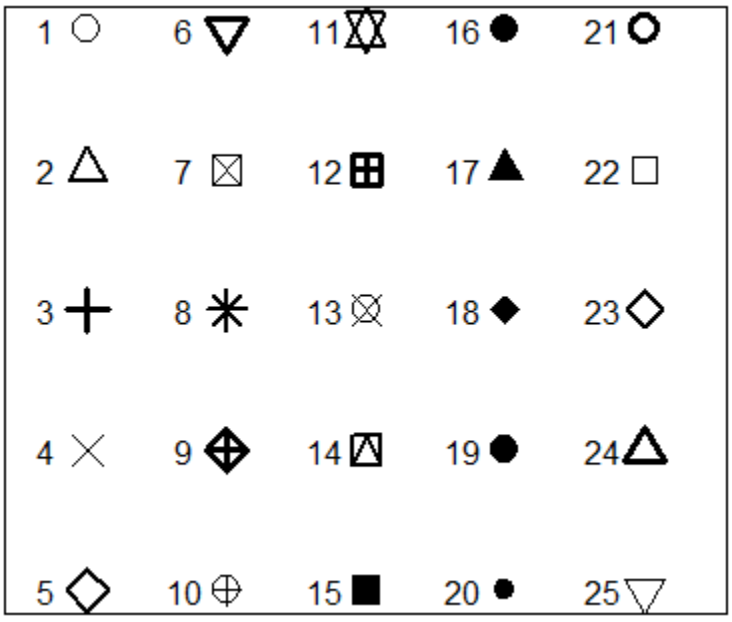
```
# make line graph
library(lubridate)
plot(lubridate::day(ric_weather[1:10, "datetime"]), ric_weather[1:10, "tempmax"],
     xlab = "Day", ylab = "Temperature (F)", main = "Temperature by Day",
     type = "b", pch = 18, col = "blue", lty = 5)
```

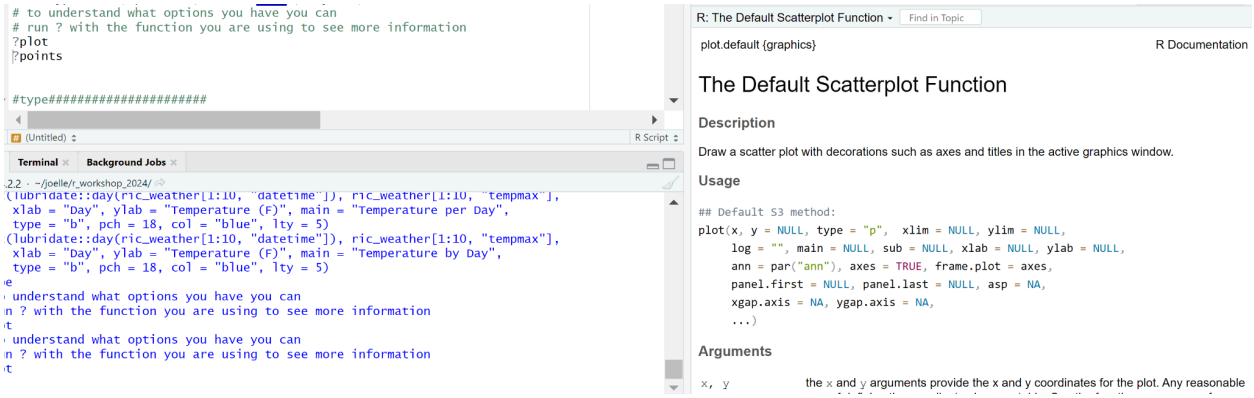


This is some information on the attributes (parts) of the graph I changed. If you want to know more about the options, you can run ? with whatever function you need to know about (ex: ?plot). In the bottom right section, R will provide the documentation on that function. This is a good place to start when you have questions. After that, try googling!

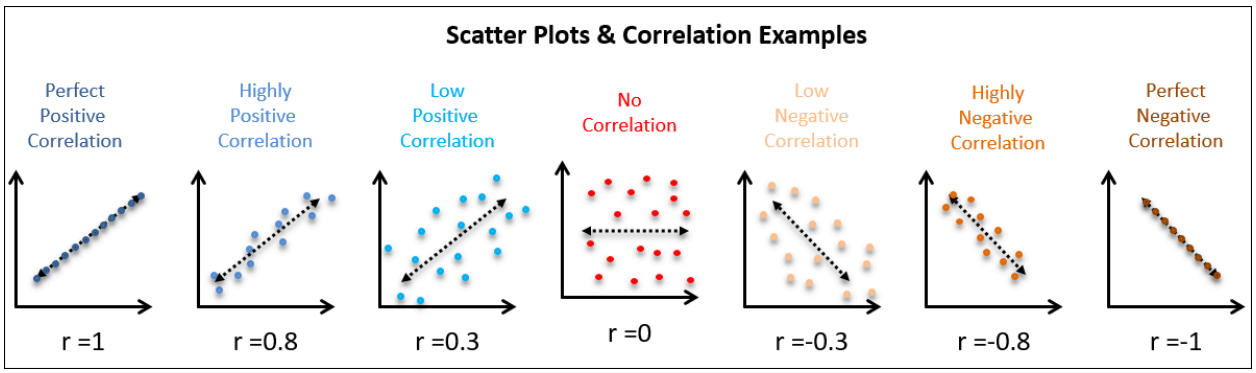
Below are more descriptions for plot type and pch.

Plot type	Description
p	Points plot (default)
l	Line plot
b	Both (points and line)
o	Both (overplotted)
s	Stairs plot
h	Histogram-like plot
n	No plotting

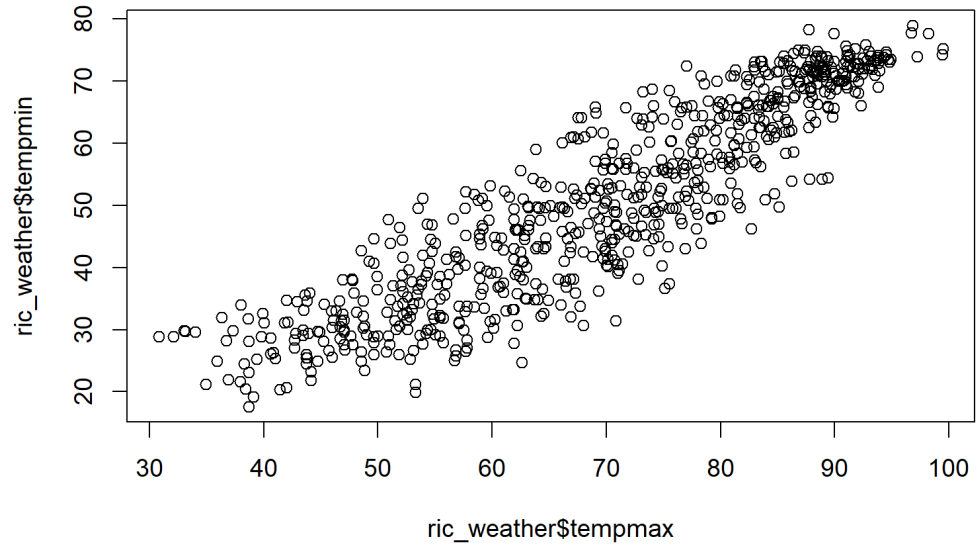




Scatter plots are used when you are looking at relationships between two numerical attributes. This can show whether the variables are correlated or not.

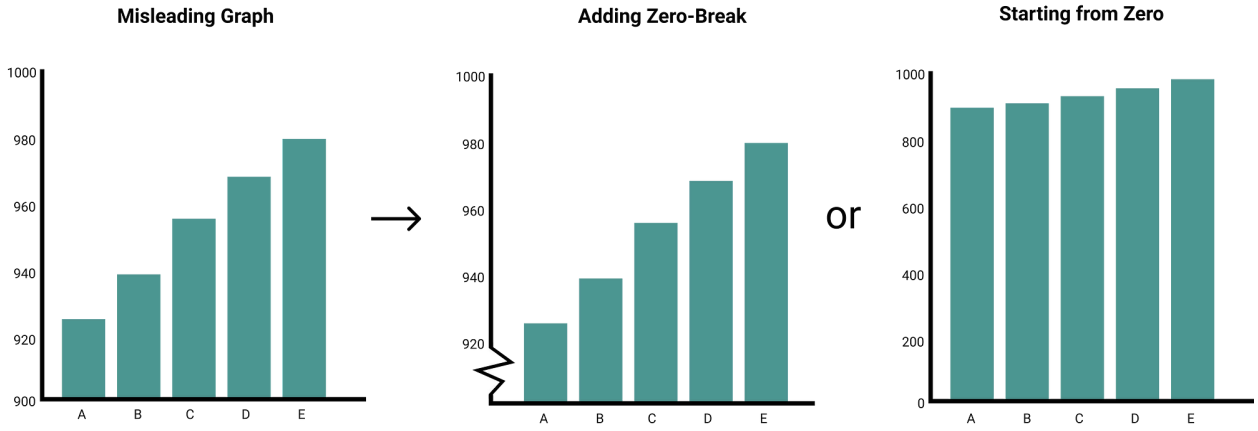


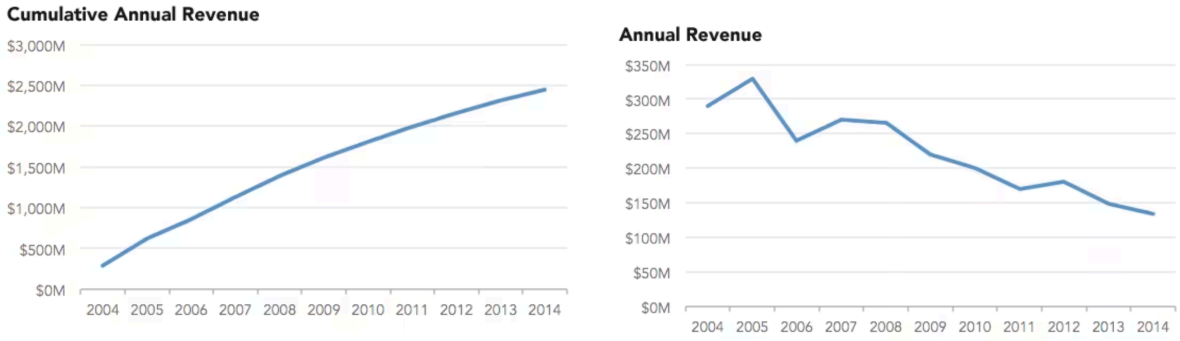

```r
# make scatterplot of max temps and min temps
plot(ric_weather$tempmax, ric_weather$tempmin)
```



Interpretation

It is important to remember that every graph can be misleading. Remember to keep scale in mind and try to think about what biases may be present in a graph. What is the creator of the graph trying to tell you and how? All of these graphs should have clearly labeled axes, titles, and units.





What issues can you see by using cumulative revenue vs annual revenue?

Ggplot

Ggplot is a useful R package and commonly used in ecology to create graphs. It provides a standardized way to specify what variables to plot, how they are displayed, and general visual properties. Therefore, we only need minimal changes if the underlying data change or if we decide to change from a bar plot to a scatterplot. This helps in creating publication quality plots with minimal amounts of adjustments and tweaking. There are a lot of things you can do with it and we will be using it for the rest of this class in the coding section.

Assignment 2:

Download your own weather data:

- https://www.visualcrossing.com/weather-data
- You must create an account but it is free

    - If you don't want to download data just use the example data!
- Create some more graphs!
    - Try to make at least two different kinds of graphs
    - Use base R and ggplot
    - Change titles, colors, plot types, etc

Session 3

Discrete vs Continuous

Discrete variables are countable and distinct values. Continuous variables can take on any value within a data set. For example, you cannot buy 1.5 chickens so a chicken is a discrete variable because they cannot be subdivided. Age can be considered continuous because you can be 20 years, 3 months, and 14 days old.

It is important to note that the distinction between discrete and continuous is sometimes just how the person chooses to measure a variable. For example, salinity could be a measure on a continuous scale OR it could be classified as low, medium, and high.

Whichever your data follow will determine what types of analyses you are able to do.

Normal distribution

Normal distribution or Gaussian distribution is a type of continuous probability distribution for a random variable. It describes a symmetric, bell-shaped probability distribution characterized by its mean (average) and standard deviation. In normal distribution:

1. **Symmetry**: The distribution is symmetric around its mean, with the highest point of the curve occurring at the mean.
2. **Bell Shape**: The shape of the distribution forms a bell curve, tapering off symmetrically on both sides of the mean.
3. **Mean and Standard Deviation**: The mean (μ) determines the central location of the distribution, while the standard deviation (σ) determines the spread or variability of the data points around the mean.

In ecology normal distribution can be seen in various aspects of populations and communities:

1. Population Density: In many ecological studies, populations of organisms often exhibit a normal distribution in terms of their density across a habitat. For example, if you were to study the distribution of a particular plant species in a forest, you might find that most individuals are concentrated around the optimal conditions for growth (e.g., near a water source or in a specific light condition), with fewer individuals occurring in less favorable areas. This creates a bell-shaped curve of population density across the habitat.

2. Trait Variation: The normal distribution also appears when examining trait variation within populations. Traits like body size, leaf area, or metabolic rates often follow a normal distribution within a population. For instance, in a population of birds, you might find that most individuals have an average wingspan, with fewer birds having wingspans significantly smaller or larger, resulting in a bell-shaped curve of trait variation.

3. Response to Environmental Factors: Ecological responses to environmental factors can also be described using normal distributions. For example, the response of a species to temperature, precipitation, or resource availability often follows a bell-shaped curve. This means that most species have an optimal range of environmental conditions where they thrive, with fewer species able to tolerate extreme conditions.

4. Biodiversity Patterns: In community ecology, the distribution of species abundances within a community often approximates a log-normal distribution, which is related to the normal distribution. In this case, most species are relatively rare, while a few species are very abundant. This pattern of species abundance is commonly observed in various ecosystems and can have important implications for ecosystem stability and function.

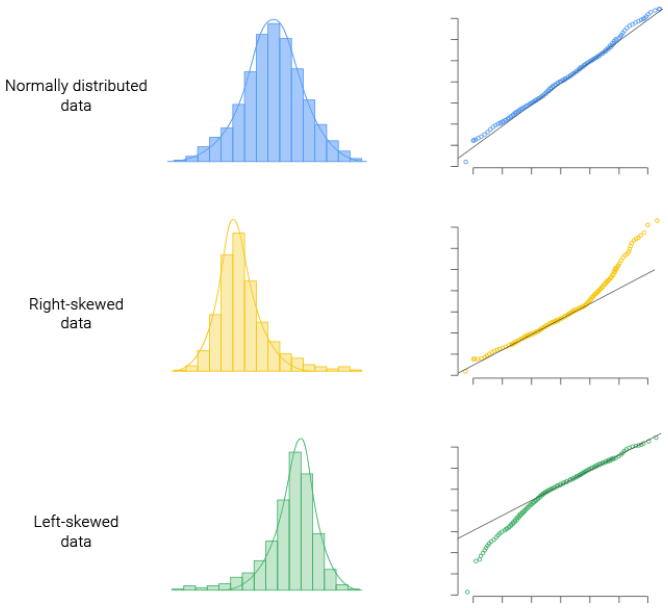
There are many other distributions. Just know that others exist that may fit your data better. The best fit will determine which types of analysis are best to understand your data. Remember that the purpose of distributions and their accompanying analyses are to allow us to make inferences about our data set (“our data” could be the population of robins or the 20 lab rats we are studying). Once we have established the connection between our data and a particular distribution, we can say anything that applies to the distribution probably applies to the real world situation we are looking at as well. It will not be perfect because we aren’t putting EVERY condition into this model, but it will allow us to generalize our findings to a larger group (even more robins or maybe 50 lab rats instead of the smaller number we gathered data on). If we know 10 of the sampled lab rats have a tail length 6-8 inches we may be able to infer that 25 of 50 lab rats also have a tail length between 6-8 inches.

Qqnorm and Shapiro-Wilk

The **quantile-quantile plot** is a graphical tool to help us assess if our data may be normally distributed. The more exactly the points fit along the bottom left to top right line, the more closely it follows normal distribution. The **Shapiro-Wilk test** is a test of normality. It assumes normality and then provides a p value which tells you the likelihood that any deviations from normal distribution were due to chance. The generally accepted p value is 0.05. If your p value is 0.05, there is a 95% chance that any deviations from the expected result (normal distribution) were due to chance and you can still assume the data are normally distributed. This is slightly different from saying 95% chance the data are normally distributed.

Here are some examples of qq norm plots and an output of a Shapiro-Wilk test. You can easily see that the blue data appear more normally distributed than the yellow or green because it

follows that straight line better. In the sample output, the p value is over 0.05 so we accept that the data tested do not differ from normal distribution.




```
> shapiro.test(ric_weather$sealevelpressure)

        Shapiro-Wilk normality test

data:  ric_weather$sealevelpressure
W = 0.99621, p-value = 0.07632
```

There is some controversy over using p values and an attempt to shift away from using them as they are often misinterpreted and may prevent important studies from being published simply because they do not meet this somewhat arbitrary 0.05 threshold. Some additional reading: 1, 2, 3, 4. However, it is still widely used and an important metric to evaluate research results.

Important terms:

- Discrete
- Continuous
- P value
- Bayesian statistics

T-test

There are many different ways to do t tests but we will be using a **two sample t-test** to look at the difference between two means. The null hypothesis is that there is no difference between means. If the means are different and we can reject the null hypothesis, the p value has to be less than 0.05. This means that there is a 5% chance of getting results that are different from expected (the null, that there is no difference in means) and we can reject the null.

For smaller sample sizes, the variable you are measuring needs to be normally distributed. However, for sample sizes over 50, you can still run t tests even if your data do not follow normal distribution. This is because of the central limit theorem. Basically as long as the sample size is large enough, the sample will be normally distributed even if the population you sample from is not.

Next steps (beyond this class): **paired t test**. In this analysis, each subject is measured twice resulting in pairs of observations. This could be used to see the impact of a breeding program or supplemental feeding. Even further: the t test only allows us to compare two groups, an **ANOVA** (analysis of variance) allows us to compare 3 or more groups.

Important terms:

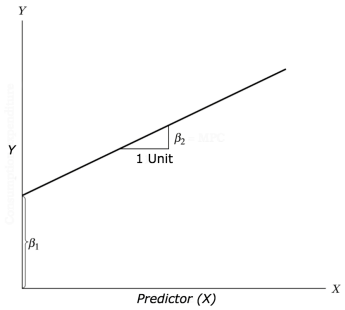
- One sample t test
- Two sample t test
- Paired t test
- Central limit theorem

Sources:

- https://www.datacamp.com/tutorial/t-tests-r-tutorial
- https://www.statisticssolutions.com/free-resources/directory-of-statistical-analyses/paired-sample-t-test/
- https://miroslavtushev.medium.com/does-my-sample-have-to-be-normally-distributed-for-a-t-test-7ee91aaaca2a
- https://stats.stackexchange.com/questions/9573/t-test-for-non-normal-when-n50

Linear Regression

Linear regression tries to find the straight line equation that approximates the relationship between two variables. One is the **explanatory/independent variable** and the other is the **dependent variable**. The equation for the line is `y' = b0 + b1*x` where y' is the predicted value, b0 is the intercept, and b1 is the angle of the slope. The **intercept** is the predicted value when the independent variable is 0. The **slope** tells you for each change in the explanatory variable the dependent variable is affected by this amount.



You can calculate the slope and intercept by hand: This example is looking at how the mass of a chemical is related to how long it takes in seconds for a reaction to use all of the chemical. We understand that if there is more of the chemical it will take more time for the reaction to complete.

Time, x (seconds)	5	7	12	16	20	
Mass, y (grams)	40	120	180	210	240	

$$\begin{aligned}\bar{x} &= \frac{\sum x}{n} \\ &= \frac{5+7+12+16+20}{5} \\ &= \frac{60}{5} \\ &= 12, \\ \bar{y} &= \frac{\sum y}{n} \\ &= \frac{40+120+180+210+240}{5} \\ &= \frac{790}{5} \\ &= 158.\end{aligned}$$

x_i	y_i	$x_i - \bar{x}$	$y_i - \bar{y}$	$(x_i - \bar{x})(y_i - \bar{y})$	$(x_i - \bar{x})^2$
5	40	$5 - 12 = -7$	$40 - 158 = -118$	$-7 \times -118 = 826$	$-7^2 = 49$
7	120	$7 - 12 = -5$	$120 - 158 = -38$	$-5 \times -38 = 190$	$-5^2 = 25$
12	180	$12 - 12 = 0$	$180 - 158 = 22$	$0 \times 22 = 0$	$0^2 = 0$
16	210	$16 - 12 = 4$	$210 - 158 = 52$	$4 \times 52 = 208$	$4^2 = 16$
20	240	$20 - 12 = 8$	$240 - 158 = 82$	$8 \times 82 = 656$	$8^2 = 64$
$\sum x = 60$	$\sum y = 790$			$\sum (x_i - \bar{x})(y_i - \bar{y}) = 1880$	$\sum (x_i - \bar{x})^2 = 154$

$$\begin{aligned}b &= \frac{S_{xy}}{S_{xx}} \\ &= \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} \\ &= \frac{1880}{154} = 12.20779\ldots \\ &= 12.208 \text{ (3.d.p.)}\end{aligned}$$

$$\begin{aligned}a &= \bar{y} - b\bar{x} \\ &= 158 - 12.208 \times 12 \\ &= 11.506\ldots \\ &= 11.506 \text{ (3.d.p.).}\end{aligned}$$

regression line is: $\hat{y} = a + bx = 11.506 + 12.208x.$

Apart from understanding how this is calculated, it is very impractical to do it this way. In R you can accomplish the same calculation by using **lm()**. First, I create the dataframe with the same information as above. Then I simply use:

```
lm(independent variable ~ the variable I am trying to predict,
dataframe I am taking the variables from)
```

By storing the results in a variable and calling summary() on it, I can see a more detailed output.

```
reaction <- data.frame(
  time = c(5,7,12,16,20),
  mass = c(40,120,180,210,240)
)
reaction_lm <- lm(mass ~ time, data = reaction)
summary(reaction_lm)
```

Output from the summary:

```
Call:
lm(formula = mass ~ time, data = reaction)

Residuals:
      1       2       3       4       5
-32.545  23.039  22.000   3.169 -15.662

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   11.506     29.687   0.388   0.7242
time          12.208      2.245   5.437   0.0122 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 27.86 on 3 degrees of freedom
Multiple R-squared:  0.9079,    Adjusted R-squared:  0.8771
F-statistic: 29.56 on 1 and 3 DF,  p-value: 0.01222
```

You can see that the **intercept** is 11.506 and the **slope** is 12.208, the same result as above. These basically mean for every gram increase in the amount of the chemical, it will take 12.208 more seconds for the reaction to complete. It also means that when there are ZERO grams of the chemical, the reaction should take 11.506 seconds to complete but clearly this is not logical. This line should start just before 0 and may continue from there. It is important to look back and make sure you understand what the numbers are telling you. In some cases, zero does not make sense and in others it does or the line can even contain negative x values.

Once you have your intercept and slope, you then need to check how different your expected values are from the real values. For this we use **R Squared**, also called the coefficient of determination. The R Squared value measures the proportion of variance explained by the independent variable. It is the performance measure for regression models. For the chemical reaction, *88% of the variance found in the reaction's time can be explained by the amount of the chemical*. You can find the r squared by pulling it directly from the summary using $ or it is in the summary output.

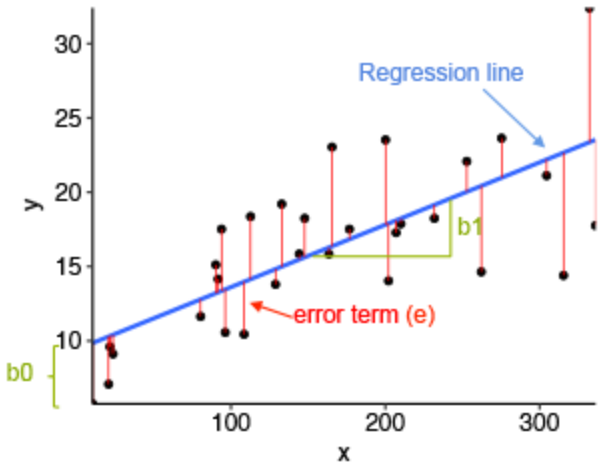
```
> summary(reaction_lm)$r.squared
[1] 0.907858
```

```
Call:
lm(formula = mass ~ time, data = reaction)

Residuals:
      1       2       3       4       5
-32.545  23.039  22.000   3.169 -15.662

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   11.506     29.687   0.388   0.7242
time          12.208      2.245   5.437   0.0122 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 27.86 on 3 degrees of freedom
Multiple R-squared:  0.9079,    Adjusted R-squared:  0.8771
F-statistic: 29.56 on 1 and 3 DF,  p-value: 0.01222
```



- the best-fit regression line is in blue
- the intercept (b0) and the slope (b1) are shown in green
- the error terms (e) are represented by vertical red lines

Sometimes, it may be appropriate to use the **adjusted r squared value**. This takes into consideration multiple independent variables and penalizes adding more independent variables. For example, if we wanted to include temperature and mass in our model to predict reaction time it would be better to use adjusted r squared to evaluate the model. This is important because there is a balance when creating models between making them generalizable and overfitting them. **Generalization** refers to how well a model can make accurate predictions on new data. Models that generalize well perform reliably when faced with new inputs. **Overfitting** occurs when a model becomes too tailored to the nuances and noise in the training data (the data used to create the model). As a result, their performance dramatically declines when evaluated on new test data.

We will use the chemical reaction example to discuss these concepts. If we put in 10 g of chemical, according to our model we expect this reaction to take 133.61 seconds to complete. However, if we test this and it actually takes 150 seconds, our model is off. If we ask the original lab that ran those first tests to try with 10 g and it takes them 133 seconds, the model may be overfitted to only be accurate under the conditions of that lab. It may be the person doing the test or the humidity in the lab or any other factor that our model is not accounting for. However, if we tested the reaction and it takes us 133 seconds, we know that the model is successfully using the underlying cause of the time and generalized well.

If there is a large difference between r squared and adjusted r squared, it is likely the model is overfitted and is only representative of your data.

Important terms:

- Dependent and explanatory/independent variables
- Lurking variable

- Residuals
- R squared value, adjusted R squared
- Generalization
- Overfit
- Extrapolation

Sources:

- http://www.stat.yale.edu/Courses/1997-98/101/linreg.htm
- http://r-statistics.co/Linear-Regression.html
- https://www.rpubs.com/Katharhy/853651

Session 4

Subsetting data

Subsetting in R is a useful feature for accessing object elements (ex: specific rows or columns of interest). It can be used to select and filter variables and observations. You can use brackets to select rows and columns from your dataframe where dataframe[rows, columns].

This pulls out the first row of the first column:

```
> ric_weather[1,1]
[1] "richmond"
```

Instead of looking at only the first column, this shows the first three columns

```
> ric_weather[1,c(1:3)]
      name   datetime tempmax
1 richmond 2020-01-01    53.6
```

If you leave the space blank, R will show you the entire range. In this case the columns were unspecified so R is showing all of them.

```
> ric_weather[c(1:3),]
      name   datetime tempmax tempmin temp feelslikemax feelslikemin feelslike  dew humidity precip precipprob precipcover preciptype snow snowdepth windgust windspeed
1 richmond 2020-01-01    53.6    34.3 43.5         53.6         29.8      40.7 28.2     56.6  0.000          0           0               0         0     20.9      14.6
2 richmond 2020-01-02    59.8    31.3 46.4         59.8         28.7      45.3 30.2     58.0  0.000          0           0               0         0     21.6      14.9
3 richmond 2020-01-03    57.7    45.2 52.0         57.7         40.0      50.2 49.4     91.4  0.204        100          50       rain    0         0     21.9      14.0
  winddir sealevelpressure cloudcover visibility solarradiation solarenergy uvindex severerisk             sunrise              sunset moonphase              conditions
1   260.5           1011.8       12.3        9.9           84.6         7.3       4         NA 2020-01-01T07:24:34 2020-01-01T17:01:53      0.21                   Clear
2   200.9           1015.5       51.0        9.9           72.3         6.3       4         NA 2020-01-02T07:24:43 2020-01-02T17:02:41      0.25        Partially cloudy
3   216.7           1011.7       74.7        8.3           37.3         3.3       2         NA 2020-01-03T07:24:50 2020-01-03T17:03:30      0.27 Rain, Partially cloudy
                                                 description              icon                                       stations
1                            Clear conditions throughout the day.         clear-day KRIC,KFCI,F2053,72401013740,KW96,KOFP,72308493775
2                               Partly cloudy throughout the day. partly-cloudy-day KRIC,KFCI,F2053,72401013740,KW96,KOFP,72308493775
3 Partly cloudy throughout the day with a chance of rain throughout the day.              rain KRIC,KFCI,F2053,72401013740,KW96,KOFP,72308493775
```

Similarly, you can call columns by name.

```
> ric_weather[1,"datetime"]
[1] "2020-01-01"
> ric_weather[1,c("name", "datetime", "tempmax")]
      name   datetime tempmax
1 richmond 2020-01-01    53.6
```

You can also use '$' to subset data by columns. You can choose which rows by adding brackets with specific rows inside. Or, if you do not specify the rows R will give you every row of that column.

```
> ric_weather$datetime[1]
[1] "2020-01-01"
> ric_weather$datetime[c(1:3)]
[1] "2020-01-01" "2020-01-02" "2020-01-03"
> ric_weather$datetime
  [1] "2020-01-01" "2020-01-02" "2020-01-03" "2020-01-04" "2020-01-05" "20
 [13] "2020-01-13" "2020-01-14" "2020-01-15" "2020-01-16" "2020-01-17" "20
 [25] "2020-01-25" "2020-01-26" "2020-01-27" "2020-01-28" "2020-01-29" "20
 [37] "2020-02-06" "2020-02-07" "2020-02-08" "2020-02-09" "2020-02-10" "20
 [49] "2020-02-18" "2020-02-19" "2020-02-20" "2020-02-21" "2020-02-22" "20
 [61] "2020-03-01" "2020-03-02" "2020-03-03" "2020-03-04" "2020-03-05" "20
 [73] "2020-03-13" "2020-03-14" "2020-03-15" "2020-03-16" "2020-03-17" "20
 [85] "2020-03-25" "2020-03-26" "2020-03-27" "2020-03-28" "2020-03-29" "20
 [97] "2020-04-06" "2020-04-07" "2020-04-08" "2020-04-09" "2020-04-10" "20
[109] "2020-04-18" "2020-04-19" "2020-04-20" "2020-04-21" "2020-04-22" "20
```

Some more interesting ways to divide data involve using conditional statements. This allows you to select rows that fulfill some requirement. For example, if you were interested in which days had a certain temperature, you would subset the data. Here, I am showing only the days where the temperature was over 85 and only the first 5 columns.

```
> ric_weather[ric_weather$temp > 85,c(1:5)]
        name   datetime tempmax tempmin temp
201 richmond 2020-07-19    99.5    75.2 86.3
202 richmond 2020-07-20    96.8    78.9 87.5
203 richmond 2020-07-21    99.4    74.3 87.2
208 richmond 2020-07-26    97.2    73.9 85.3
209 richmond 2020-07-27    96.7    77.8 87.0
210 richmond 2020-07-28    98.2    77.7 86.5
```

These are all examples using base R but we will see how to subset data using dplyr.

Transforming data

Data transformation is the process of modifying data to fit a particular purpose or context. This can involve tasks such as cleaning, standardization, or aggregation, all aimed at enhancing data quality and usability for more accurate analysis. We will focus on simple transformations such as converting units or removing outliers. Once data has been transformed, it is converted into a more usable format.

Here, we convert the temperatures in the "temp" column from Fahrenheit to Celsius.

```
> celsius <- (ric_weather$temp - 32) * 5 / 9
> celsius
  [1]  6.38888889  8.00000000 11.11111111 13.50000000  5.88888889  5.
 [14] 11.11111111 10.05555556 11.72222222  1.44444444  1.16666667  6.
 [27]  5.33333333  5.33333333  2.33333333  1.22222222  2.11111111  5.
 [40]  4.00000000  8.72222222 13.66666667  8.50000000 11.61111111  4.
 [53]  2.11111111  6.61111111  8.11111111 10.44444444 11.00000000  6.
```

With the list of numbers converted, we can then add a column to the dataframe.

```
> ric_weather$temp_c <- celsius
> head(ric_weather[,c(1:5, 34)])
      name   datetime tempmax tempmin temp    temp_c
1 richmond 2020-01-01    53.6    34.3 43.5  6.388889
2 richmond 2020-01-02    59.8    31.3 46.4  8.000000
3 richmond 2020-01-03    57.7    45.2 52.0 11.111111
4 richmond 2020-01-04    62.2    46.1 56.3 13.500000
5 richmond 2020-01-05    48.7    34.6 42.6  5.888889
6 richmond 2020-01-06    55.7    31.9 42.1  5.611111
```

Removing rows with values at a certain threshold is just subsetting. The important part is figuring out which values to subset. It could be based on an outlier calculation or any sort of significant number. In this example I chose 85 again as the cut off. You can see that there are fewer rows in the new dataframe.

```
> below_85 <- ric_weather[ric_weather$temp < 85,]
> nrow(below_85)
[1] 725
> nrow(ric_weather)
[1] 731
```

Dplyr

Dplyr is a powerful tool in R. It provides simple “verbs”, functions that correspond to the most common data manipulation tasks, to help you translate your thoughts into code. It is extremely useful to have this interface to perform data manipulations. The pipe (%>%) also makes your code much more readable. It is easy to track which manipulations you have made which is invaluable.

For example, here are the previous examples manipulating data but this time using dplyr. Although you do not know the syntax yet, it should be somewhat easy to understand that first you create the column called celsius according to the equation (red boxes) and after that, you filter only the rows where the temperature is less than 85 (blue boxes). It is tricky to understand at first, but it should be more intuitive than the way without using dplyr.

```
> below_85_dplyr <- ric_weather %>%
+   dplyr::mutate(celsius = (temp - 32) * 5 / 9) %>%
+   dplyr::filter(temp < 85)
> head(below_85_dplyr[,c(1:5, 35)])
      name   datetime tempmax tempmin temp   celsius
1 richmond 2020-01-01    53.6    34.3 43.5  6.388889
2 richmond 2020-01-02    59.8    31.3 46.4  8.000000
3 richmond 2020-01-03    57.7    45.2 52.0 11.111111
4 richmond 2020-01-04    62.2    46.1 56.3 13.500000
5 richmond 2020-01-05    48.7    34.6 42.6  5.888889
6 richmond 2020-01-06    55.7    31.9 42.1  5.611111
> nrow(below_85_dplyr)
[1] 725
```

VS

```
> ric_weather$temp_c <- (ric_weather$temp - 32) * 5 / 9
> below_85 <- ric_weather[ric_weather$temp < 85,]
> head(below_85[,c(1:5, 34)])
      name   datetime tempmax tempmin temp    temp_c
1 richmond 2020-01-01    53.6    34.3 43.5  6.388889
2 richmond 2020-01-02    59.8    31.3 46.4  8.000000
3 richmond 2020-01-03    57.7    45.2 52.0 11.111111
4 richmond 2020-01-04    62.2    46.1 56.3 13.500000
5 richmond 2020-01-05    48.7    34.6 42.6  5.888889
6 richmond 2020-01-06    55.7    31.9 42.1  5.611111
> nrow(below_85)
[1] 725
```

Now let's go into some more details of the functions most commonly used in dplyr. The main ones are:

- select(): choose certain columns
- filter(): choose certain rows
- arrange(): sorts data
- mutate(): adds a column
- group_by(): groups data, does not have any output on its own, needs to be combined with summarise() with an action to perform
    - Ex: you could find the mean of every day's temperature OR you could group the days by month and find the mean of January, February, March, etc
- summarise(): creates one row summarizing all observations for the input

The data for the next examples comes from a fake dataset of each state's income for the year from 2002-2015 (using this link https://raw.githubusercontent.com/deepanshu88/data/master/sampledata.csv). There is also an index column that is just the letter the state's name starts with.

```
> mydata = read.csv("https://raw.githubusercontent.com/deepanshu88/data/master/sampledata.csv")
> head(mydata)
  Index       State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1     A     Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661
2     A      Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143
3     A     Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724
4     A    Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341
5     C  California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607
6     C    Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387 1913275 1665877 1491604 1178355 1383978 1330736
>
```

Select()

This allows you to take a selection of the data using column names.

```
> mydata_selection <- mydata %>%
+   dplyr::select(State, Y2010, Y2015)
> head(mydata_selection)
       State   Y2010   Y2015
1    Alabama 1237582 1916661
2     Alaska 1629616 1979143
3    Arizona 1300521 1647724
4   Arkansas 1669295 1329341
5 California 1624509 1644607
6   Colorado 1913275 1330736
```

Notice the difference between creating a new dataframe called “mydata_selection” using “mydata” (above) and selecting columns without creating a dataframe (below). Notice the difference in syntax (using the <-) and how you cannot recall the transformations you did if you do not create a new variable/dataframe. It is just displayed once.

```
> mydata %>%
+   dplyr::select(State, Y2010, Y2015)
                  State   Y2010   Y2015
1               Alabama 1237582 1916661
2                Alaska 1629616 1979143
3               Arizona 1300521 1647724
4              Arkansas 1669295 1329341
5            California 1624509 1644607
6              Colorado 1913275 1330736
```

You can also choose columns to ignore by using “-” (minus sign)

```
> mydata %>%
+   dplyr::select(-c(Index, Y2002, Y2003, Y2004, Y2005))
                State   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1             Alabama 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661
2              Alaska 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143
3             Arizona 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724
4            Arkansas 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341
5          California 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607
6            Colorado 1871471 1814218 1875146 1752387 1913275 1665877 1491604 1178355 1383978 1330736
```

This time we are choosing rows Y2002 TO Y2005 using a ‘:’. We also just show the first 10 rows in a pipe by using head(). Here you can see how adding an additional “pipe” allows you to stack things. First R will select the right columns, and then it will select the first 10 rows.

```
> mydata %>%
+   dplyr::select(c(State, Y2002:Y2005)) %>%
+   head(., 10)
                   State   Y2002   Y2003   Y2004   Y2005
1                Alabama 1296530 1317711 1118631 1492583
2                 Alaska 1170302 1960378 1818085 1447852
3                Arizona 1742027 1968140 1377583 1782199
4               Arkansas 1485531 1994927 1119299 1947979
5             California 1685349 1675807 1889570 1480280
6               Colorado 1343824 1878473 1886149 1236697
7            Connecticut 1610512 1232844 1181949 1518933
8               Delaware 1330403 1268673 1706751 1403759
9  District of Columbia 1111437 1993741 1374643 1827949
10               Florida 1964626 1468852 1419738 1362787
>
```

Filter()

Filter makes it easy to subset rows based on a condition.

For example we are only choosing rows where Y2002 values are greater than 1,900,000

```
> mydata %>%
+   dplyr::filter(Y2002 > 1900000)
  Index       State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1     F     Florida 1964626 1468852 1419738 1362787 1339608 1278550 1756185 1818438 1198403 1497051 1131928 1107448 1407784 1170389
2     G     Georgia 1929009 1541565 1810773 1779091 1326846 1223770 1773090 1630325 1145473 1851245 1850111 1887157 1259353 1725470
3     M Mississippi 1983285 1292558 1631325 1943311 1354579 1731643 1428291 1568049 1383227 1629132 1988270 1907777 1649668 1991232
4     W  Washington 1977749 1687136 1199490 1163092 1334864 1621989 1545621 1555554 1179331 1150089 1775787 1273834 1387428 1377341
```

This time TWO conditions must be met: Y2002 and Y2003 must be greater than 1,500,000. We are also selecting only the columns Index, State, Y2002, and Y2003. This just makes it easier to look at since there is less on the screen.

```
> mydata %>%
+   dplyr::filter(Y2002 > 1500000 & Y2003 > 1500000) %>%
+   dplyr::select(Index, State, Y2002, Y2003)
   Index          State   Y2002   Y2003
1      A        Arizona 1742027 1968140
2      C     California 1685349 1675807
3      G        Georgia 1929009 1541565
4      I       Illinois 1508356 1527440
5      I        Indiana 1776918 1734104
6      M          Maine 1582720 1678622
7      M  Massachusetts 1647582 1686259
8      M      Minnesota 1729921 1675204
9      M        Montana 1877154 1540099
10     N   North Dakota 1618807 1510193
11     O           Ohio 1802132 1648498
12     O         Oregon 1794912 1726665
13     R   Rhode Island 1501744 1942942
14     S South Carolina 1631522 1803455
15     W     Washington 1977749 1687136
16     W      Wisconsin 1788920 1518578
```

This is selecting only rows where the Index is A, B, or C. %in% means we are looking for Index to match any of the values in the list provided (remember we need ‘c’ to specify a vector).

```
> mydata %>%
+   dplyr::filter(Index %in% c("A", "B", "C"))
  Index       State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1     A     Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661
2     A      Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143
3     A     Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724
4     A    Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341
5     C  California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607
6     C    Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387 1913275 1665877 1491604 1178355 1383978 1330736
7     C Connecticut 1610512 1232844 1181949 1518933 1841266 1976976 1764457 1972730 1968730 1945524 1228529 1582249 1503156 1718072
```

Now we are using another package “stringr” to search for a matching section of a string. The State column is character type. We are looking for states that end in “land” and then states that start with “New”.

```
> mydata %>%
+   dplyr::filter(str_detect(State, "land$"))
  Index       State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1     M    Maryland 1579713 1404700 1849798 1397738 1310270 1789128 1112765 1967225 1486246 1872327 1175819 1314343 1979529 1569566
2     R Rhode Island 1501744 1942942 1266657 1961923 1835983 1234040 1151409 1993136 1983569 1781016 1909119 1531212 1990412 1611730

> mydata %>%
+   dplyr::filter(str_detect(State, "^New"))
  Index          State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1     N New Hampshire 1419776 1854370 1195119 1990062 1645430 1286967 1762936 1763211 1265642 1704297 1131298 1197576 1242623 1963313
2     N     New Jersey 1605532 1141514 1613550 1181452 1541327 1156804 1568034 1357418 1443718 1390010 1202326 1100990 1850165 1183568
3     N     New Mexico 1819239 1226057 1935991 1124400 1723493 1475985 1237704 1820856 1801430 1653384 1475715 1623388 1533494 1868612
4     N       New York 1395149 1611371 1170675 1446810 1426941 1463171 1732098 1426216 1604531 1683687 1500089 1718837 1619033 1367705
```

Notice how if you search for “new” all lowercase, R does not find anything. This is because a computer needs explicit instructions and cannot know that New and new are functionally the same.

```
> mydata %>%
+   dplyr::filter(str_detect(State, "^new"))
 [1] Index State Y2002 Y2003 Y2004 Y2005 Y2006 Y2007 Y2008 Y2009 Y2010 Y2011 Y2012 Y2013 Y2014 Y2015
<0 rows> (or 0-length row.names)
```

To get around this you can use a lowercase string to search and change all the values in the State column to lowercase. Using tolower() like this does not change anything in the actual dataframe because we have not used the assignment operator. It is only lowercase to check for a match.

```
> mydata %>%
+   dplyr::filter(str_detect(tolower(State), "^new"))
  Index          State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1     N New Hampshire 1419776 1854370 1195119 1990062 1645430 1286967 1762936 1763211 1265642 1704297 1131298 1197576 1242623 1963313
2     N     New Jersey 1605532 1141514 1613550 1181452 1541327 1156804 1568034 1357418 1443718 1390010 1202326 1100990 1850165 1183568
3     N     New Mexico 1819239 1226057 1935991 1124400 1723493 1475985 1237704 1820856 1801430 1653384 1475715 1623388 1533494 1868612
4     N       New York 1395149 1611371 1170675 1446810 1426941 1463171 1732098 1426216 1604531 1683687 1500089 1718837 1619033 1367705
```

Arrange()

Arrange lets you sort rows. You can do this in ascending order or descending order.

In this case, we sort the columns from lowest Y2002 value to highest and also select only the State and Y2002 columns.

```
> mydata %>%
+   dplyr::arrange(Y2002) %>%
+   dplyr::select(State, Y2002)
                  State   Y2002
1  District of Columbia 1111437
2              Virginia 1134317
3               Vermont 1146902
4          South Dakota 1159037
5                Alaska 1170302
6              Oklahoma 1173918
7              Missouri 1221316
8              Michigan 1295635
9               Alabama 1296530
```

You can use desc() to sort column values the other way. Now the greatest values are at the top.

```
> mydata %>%
+   dplyr::arrange(desc(Y2002)) %>%
+   dplyr::select(State, Y2002) %>%
+   head(., 10)
         State   Y2002
1  Mississippi 1983285
2   Washington 1977749
3      Florida 1964626
4      Georgia 1929009
5     Nebraska 1885081
6      Montana 1877154
7   New Mexico 1819239
8     Kentucky 1813878
9    Tennessee 1811867
10        Ohio 1802132
```

Mutate()

Mutate creates new variables and preserves existing ones. Inside the () (because functions always have parentheses), you add whatever you want to name your new column with = to set the new column value. R will know to do this for each row. In this first example we have just created another column that is the same as Y2015.

```
> mutate_column <- mydata %>%
+   dplyr::mutate(new_column_name = Y2015)
> head(mutate_column)
  Index      State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015 new_column_name
1     A    Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661         1916661
2     A     Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143         1979143
3     A    Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724         1647724
4     A   Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341         1329341
5     C California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607         1644607
6     C   Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387 1913275 1665877 1491604 1178355 1383978 1330736         1330736
```

Mutate is more useful when you need to convert units or perform calculations. For example we have income by year but what if instead we want average income for years 2014-2015?

Here we are finding the average of Y2014 and Y2015 and storing those values in a column called ave_1415. If you are ever unsure whether you used the correct code to do what you want you can always check manually. (Try checking a few of these row averages by using R like a calculator)

```
> mutate_ave_1415 <- mydata %>%
+   dplyr::mutate(ave_1415 = rowMeans(dplyr::select(mydata, c(Y2014:Y2015))))
> head(mutate_ave_1415)
  Index      State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015 ave_1415
1     A    Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661  1737784
2     A     Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143  1779769
3     A    Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724  1586795
4     A   Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341  1345150
5     C California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607  1516534
6     C   Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387 1913275 1665877 1491604 1178355 1383978 1330736  1357357
```

Now pretend that these values are in mm but we need cm. We create a new column called cm15 to store Y2015 values divided by 10.

```
> mutate_convert_to_cm <- mydata %>%
+   dplyr::mutate(cm15 = Y2015/10)
> head(mutate_convert_to_cm)
  Index      State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015     cm15
1     A    Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661 191666.1
2     A     Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143 197914.3
3     A    Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724 164772.4
4     A   Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341 132934.1
5     C California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607 164460.7
6     C   Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387 1913275 1665877 1491604 1178355 1383978 1330736 133073.6
```

Doing one column at a time might be tedious though. To try to do the entire datatable, we could select the columns of interest (Y2002-Y2015) and perform the same calculation across them. Because these are all numbers, it is pretty easy to do this.

```
> convert_to_cm <- mydata %>% dplyr::select(c(3:16))/10
> head(convert_to_cm)
     Y2002    Y2003    Y2004    Y2005    Y2006    Y2007    Y2008    Y2009    Y2010    Y2011    Y2012    Y2013    Y2014    Y2015
1 129653.0 131771.1 111863.1 149258.3 110740.8 144013.4 194522.9 194417.3 123758.2 144075.6 118674.1 185284.1 155890.6 191666.1
2 117030.2 196037.8 181808.5 144785.2 186163.9 146584.1 155182.6 143654.1 162961.6 123086.6 151280.4 198530.2 158039.4 197914.3
3 174202.7 196814.0 137758.3 178219.9 110256.8 110938.2 175288.6 155433.0 130052.1 113070.9 190728.4 136327.9 152586.6 164772.4
4 148553.1 199492.7 111929.9 194797.9 166919.1 180121.3 118810.4 162898.0 166929.5 192823.8 121667.5 159189.6 136095.9 132934.1
5 168534.9 167580.7 188957.0 148028.0 173506.9 181254.6 148731.5 166380.9 162450.9 163967.0 192184.5 115653.6 138846.1 164460.7
6 134382.4 187847.3 188614.9 123669.7 187147.1 181421.8 187514.6 175238.7 191327.5 166587.7 149160.4 117835.5 138397.8 133073.6
```

However, notice that the State and Index columns were lost doing it this way.

A conditional mutate allows you to perform some action based on the values of a column. For example we could create a column that is the state's income IF the value is greater than

100,000 and otherwise set the value to NA. Similarly, we can use case_when to have multiple conditions. If Y2015 is 0-1300000 set the category to “low”, 1300001-1750000 is “medium”, and >1750000 is “high”. R will assign the category according to your specifications.

```
mutate_buckets <- mydata %>%
  dplyr::mutate(category_15 = case_when(Y2015 < 1300000 ~ "low",
                                        Y2015 < 1750000 ~ "medium",
                                        .default = "high"))
```

```
> head(mutate_buckets[,c(2, 16:17)], 10)
                   State   Y2015 category_15
1                Alabama 1916661        high
2                 Alaska 1979143        high
3                Arizona 1647724      medium
4               Arkansas 1329341      medium
5             California 1644607      medium
6               Colorado 1330736      medium
7            Connecticut 1718072      medium
8               Delaware 1627508      medium
9  District of Columbia 1410183      medium
10               Florida 1170389         low
```

Summarise() and Group_by()

Summarise() is a very useful function that lets you summarize data. In this first example, we find the average income for all states in the year 2002.

```
> mydata %>%
+   dplyr::summarise(all_states_2002 = mean(Y2002))
  all_states_2002
1         1566034
```

Summarise() is often used in conjunction with group_by() to summarize by row values. For example we can find the average income in 2002 for states that have the same first letter (the column “Index” for this dataset).

```
> mydata %>%
+   dplyr::group_by(Index) %>%
+   dplyr::summarise(by_index_2002 = mean(Y2002))
# A tibble: 19 × 2
   Index by_index_2002
   <chr>         <dbl>
 1 A          1423598.
 2 C          1546562.
 3 D          1220920
 4 F          1964626
 5 G          1929009
 6 H          1461570
 7 I          1534438.
 8 K          1661466
 9 L          1584734
10 M          1614666.
11 N          1598305.
12 O          1590321.
13 P          1320191
14 R          1501744
15 S          1395280.
16 T          1666229
17 U          1771096
18 V          1140610.
19 W          1804802.
```

Or in this example, we are using the groupings created earlier with mutate() in which we assigned a “low”, “medium”, or “high” value according to 2015 income. Now, we will group by the category_15 column instead of Index.

```
> mutate_buckets %>%
+   dplyr::group_by(category_15) %>%
+   dplyr::summarise(mean_15 = mean(Y2015))
# A tibble: 3 × 2
  category_15  mean_15
  <chr>          <dbl>
1 high        1897712.
2 low         1163098.
3 medium      1545073.
```

Tidyr

This package is very useful for reshaping data between wide and long format. Understanding the differences between wide and long was confusing for me and simply looking at examples and using the data differently helped me the most. Look at the following screenshots. When the data is wide, each state has multiple columns for each year’s income. When converted to long,

each state appears multiple times in each row but has a new column indicating the year, and there is only one value for income in each row.

Wide format

```
# A tibble: 51 × 16
   Index State                Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
   <chr> <chr>                <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>
 1 A     Alabama            1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661
 2 A     Alaska             1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143
 3 A     Arizona            1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724
 4 A     Arkansas           1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341
 5 C     California         1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607
 6 C     Colorado           1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387 1913275 1665877 1491604 1178355 1383978 1330736
 7 C     Connecticut        1610512 1232844 1181949 1518933 1841266 1976976 1764457 1972730 1968730 1945524 1228529 1582249 1503156 1718072
 8 D     Delaware           1330403 1268673 1706751 1403759 1441351 1300836 1762096 1553585 1370984 1318669 1984027 1671279 1803169 1627508
 9 D     District of Columbia 1111437 1993741 1374643 1827949 1803852 1595981 1193245 1739748 1707823 1353449 1979708 1912654 1782169 1410183
10 F     Florida            1964626 1468852 1419738 1362787 1339608 1278550 1756185 1818438 1198403 1497051 1131928 1107448 1407784 1170389
# … with 41 more rows
# ℹ Use `print(n = ...)` to see more rows
```

Long format

```
# A tibble: 714 × 4
   Index State   year   income
   <chr> <chr>   <chr>   <int>
 1 A     Alabama Y2002 1296530
 2 A     Alabama Y2003 1317711
 3 A     Alabama Y2004 1118631
 4 A     Alabama Y2005 1492583
 5 A     Alabama Y2006 1107408
 6 A     Alabama Y2007 1440134
 7 A     Alabama Y2008 1945229
 8 A     Alabama Y2009 1944173
 9 A     Alabama Y2010 1237582
10 A     Alabama Y2011 1440756
11 A     Alabama Y2012 1186741
12 A     Alabama Y2013 1852841
13 A     Alabama Y2014 1558906
14 A     Alabama Y2015 1916661
15 A     Alaska  Y2002 1170302
16 A     Alaska  Y2003 1960378
17 A     Alaska  Y2004 1818085
18 A     Alaska  Y2005 1447852
19 A     Alaska  Y2006 1861639
20 A     Alaska  Y2007 1465841
# … with 694 more rows
```

A wide format has values that **do not** repeat in the first column. A long format has values that **do** repeat in the first column.

We will go through the steps together. First look at the data we have:

```
> head(mydata)
  Index      State   Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1     A    Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661
2     A     Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143
3     A    Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724
4     A   Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341
5     C California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607
6     C   Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387 1913275 1665877 1491604 1178355 1383978 1330736
```

Because there is only one row for each state and multiple values for each income but at different years, we know this is in wide format. Now we use tidyr to change it into a long format so there is only one column for the income and multiple rows indicating the year. Use the “names_to” variable to indicate how to label the value (in this case the old column titles are labeling income so year is the name). Use “values_to” to indicate the value. Use “cols” to indicate which columns have the corresponding pairs of a year name and an income value. Here we are only using rows 3-16 because the first columns have index and state and not year or income. The red boxes show that we are calling this column “year” and taking the values from

the names of the old columns (Y2002, Y2003, etc). The blue boxes that we are calling this column “income” and it is coming from that country’s data. Purple indicates which rows we are pulling this from. We told R that the year/income data is in columns 3 to 16 so it knows to create 14 rows, one for each year/income recorded for each state.

This is basically turning 1 row and 14 columns into 14 rows and 1 column (split between the name/label or year and value or income) instead.

```
> mydata_long <- mydata %>%
+   tidyr::pivot_longer(names_to = "year",
+                       values_to = "income",
+                       cols = 3:16) %>%
+   print(n = 20)
# A tibble: 714 × 4
   Index State   year  income
   <chr> <chr>   <chr>  <int>
 1 A     Alabama Y2002 1296530
 2 A     Alabama Y2003 1317711
 3 A     Alabama Y2004 1118631
 4 A     Alabama Y2005 1492583
 5 A     Alabama Y2006 1107408
 6 A     Alabama Y2007 1440134
 7 A     Alabama Y2008 1945229
 8 A     Alabama Y2009 1944173
 9 A     Alabama Y2010 1237582
10 A     Alabama Y2011 1440756
11 A     Alabama Y2012 1186741
12 A     Alabama Y2013 1852841
13 A     Alabama Y2014 1558906
14 A     Alabama Y2015 1916661
15 A     Alaska  Y2002 1170302
16 A     Alaska  Y2003 1960378
17 A     Alaska  Y2004 1818085
18 A     Alaska  Y2005 1447852
19 A     Alaska  Y2006 1861639
20 A     Alaska  Y2007 1465841
```

```
> head(mydata)
  Index      State  Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
1     A    Alabama 1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661
2     A     Alaska 1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143
3     A    Arizona 1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724
4     A   Arkansas 1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341
5     C California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607
6     C   Colorado 1343824 1878473 1886149 1236697 1871471 1814218 1875146 1752387 1913275 1665877 1491604 1178355 1383978 1330736
```

Going from long to wide format is similar. We tell R which column has the column names and which column has the column values.

```
> mydata_wide <- mydata_long %>%
+   tidyr::pivot_wider(names_from = "year",
+                      values_from = "income") %>%
+   print(n = 5)
# A tibble: 51 × 16
  Index State       Y2002   Y2003   Y2004   Y2005   Y2006   Y2007   Y2008   Y2009   Y2010   Y2011   Y2012   Y2013   Y2014   Y2015
  <chr> <chr>       <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>
1 A     Alabama   1296530 1317711 1118631 1492583 1107408 1440134 1945229 1944173 1237582 1440756 1186741 1852841 1558906 1916661
2 A     Alaska    1170302 1960378 1818085 1447852 1861639 1465841 1551826 1436541 1629616 1230866 1512804 1985302 1580394 1979143
3 A     Arizona   1742027 1968140 1377583 1782199 1102568 1109382 1752886 1554330 1300521 1130709 1907284 1363279 1525866 1647724
4 A     Arkansas  1485531 1994927 1119299 1947979 1669191 1801213 1188104 1628980 1669295 1928238 1216675 1591896 1360959 1329341
5 C     California 1685349 1675807 1889570 1480280 1735069 1812546 1487315 1663809 1624509 1639670 1921845 1156536 1388461 1644607
# … with 46 more rows
```

Important terms:

- Long vs wide format

Sources:

- https://dplyr.tidyverse.org/
- https://www.stat.cmu.edu/~ryantibs/statcomp/lectures/dplyr.html
    - https://www.stat.cmu.edu/~ryantibs/statcomp/ (this whole class is a VERY good overview of R)
- https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

Resources:

- Introduction/Installation:
    - https://teacherscollege.screenstepslive.com/a/1108074-install-r-and-rstudio-for-windows (Windows)
    - https://teacherscollege.screenstepslive.com/a/1135059-install-r-and-rstudio-for-mac (Mac)
    - https://docs.posit.co/ide/user/ide/guide/ui/ui-panes.html
- Guides:
    - Hands-On Programming with R (best option!)
    - https://r4ds.hadley.nz/
    - https://bookdown.org/ndphillips/YaRrr/
    - https://www.programiz.com/r/getting-started
- General questions
    - Generally good to google: <your question> + “in r” (ex: make line graph in r)
    - https://www.w3schools.com/r/
    - https://www.geeksforgeeks.org/r-programming-language-introduction/?ref=lbp
    - https://stackoverflow.com/questions/tagged/r
- Package documentation
    - https://www.rdocumentation.org/
    - https://rdrr.io/