

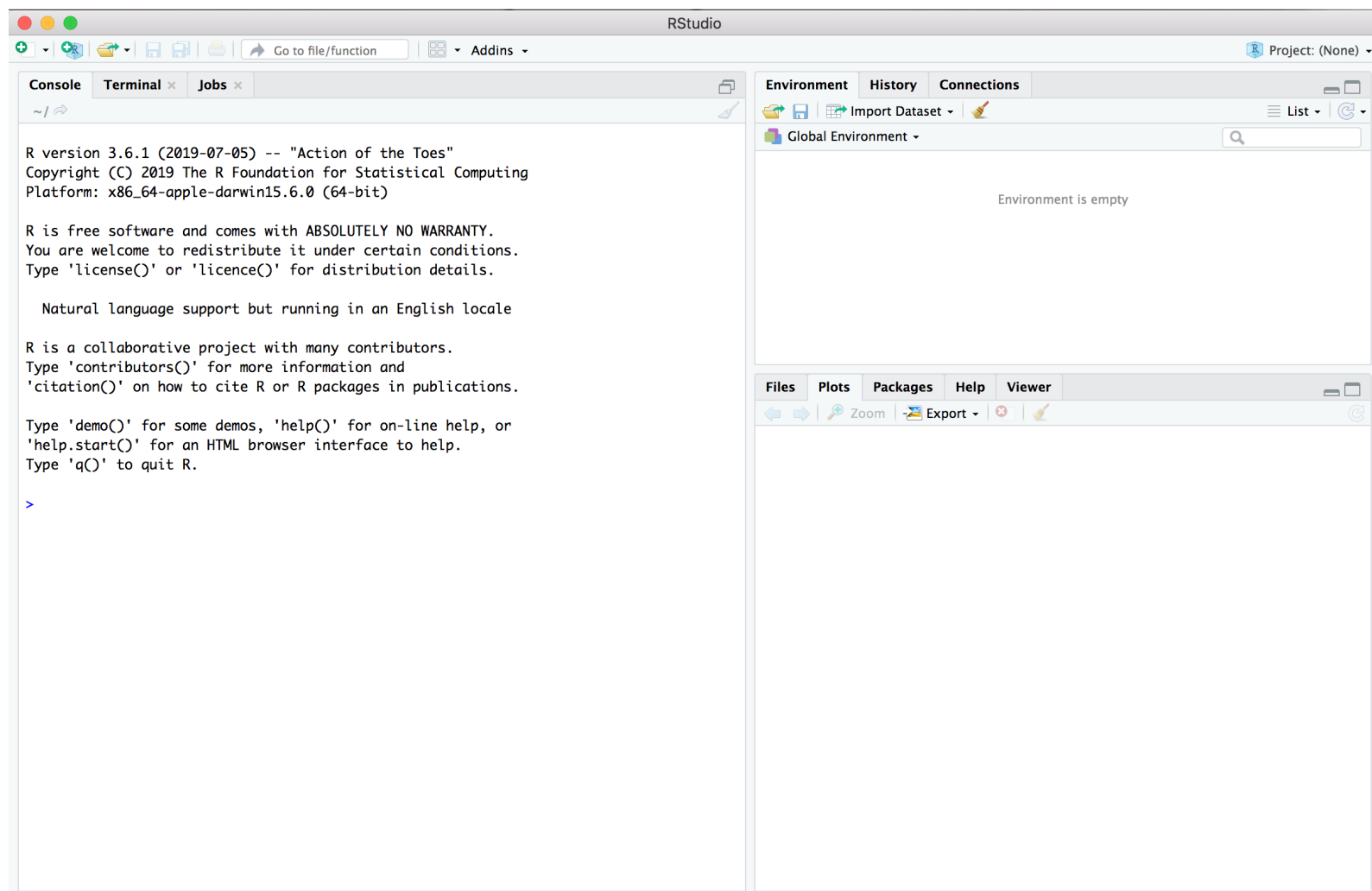


# Introduction to R

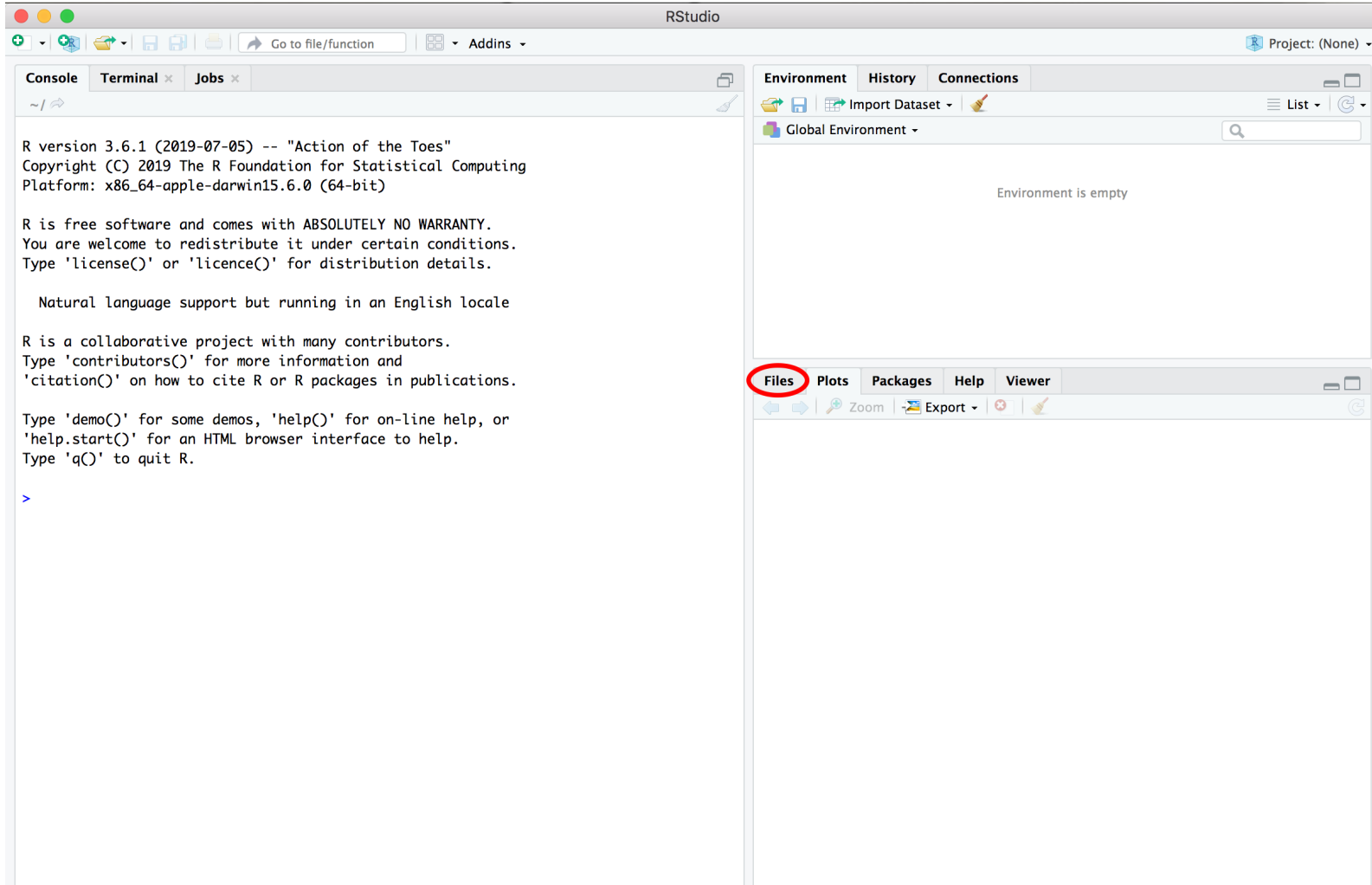
## Getting Started With RStudio

Follow the instructions in [section 1.1.1 of ModernDive](https://moderndive.com/1-getting-started.html#installing)  (<https://moderndive.com/1-getting-started.html#installing>) to install R and RStudio on your computer (alternately, if for some reason you are unable to install these on your computer, you can access RStudio through Seattle U's [virtual desktop \(http://desktop.seattleu.edu\)](http://desktop.seattleu.edu), or you can use [RStudio Cloud](https://rstudio.cloud)  (<https://rstudio.cloud>), although you will likely find things easier to work with if you can install them on your own computer).

Open RStudio. You should see a window that looks like the image below:



We should create a folder to store the files we will be using for our course. In the lower right panel, there is a tab labeled "Files":

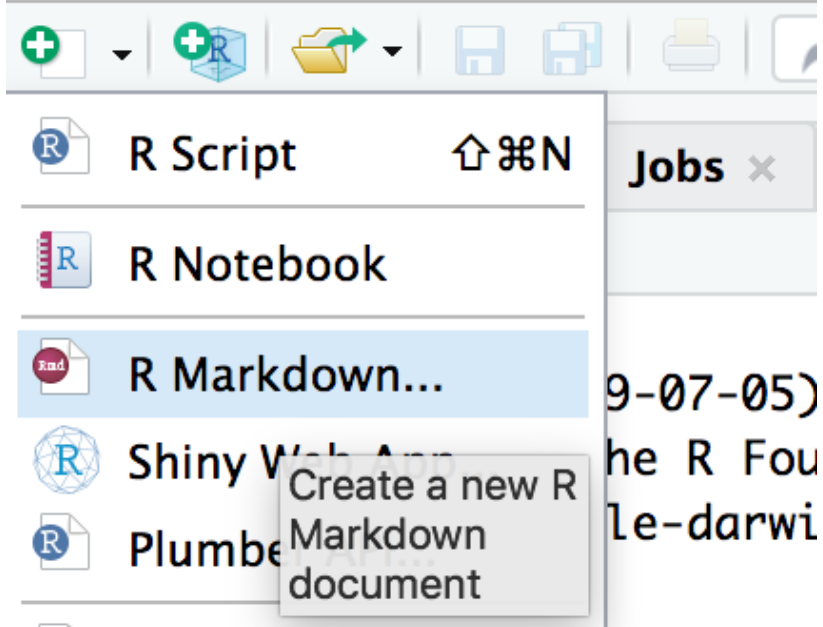


Click on the "Files" tab. Directly below it you will see a button labeled "New Folder". Click on this, and create a new folder named "DATA 5300."

## Writing Code

While we can write code directly into the R console, it is more useful to create a document with all of our code where we can go back to pieces of code we might want to re-run, where we can see all of our code written out at once, and where we can leave comments explaining what our code is doing.


We can do this in RStudio by creating an *R Markdown* document. In the upper left corner of RStudio, click on the green plus sign, then select "R Markdown" from the menu that appears:





(The first time that you do this, you might receive a prompt asking you to install or update some required packages - if so, install the needed packages.)


A new window will pop up, asking you for details on the document you want to create:

New R Markdown

 Document

 Presentation

 Shiny

 From Template

**Title:**

**Author:**

**Default Output Format:**

☒ **HTML**

Recommended format for authoring (you can switch to PDF or Word output anytime).

☐ **PDF**

PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

☐ **Word**

Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

OK

Cancel

There are options to set a title and an author's name for this document here. For the moment, we will leave these as they are. We will see how to change this after we have created the document.

There are options to create a variety of document formats. In general, HTML will be the most versatile option, so we will stick with that for the time being. Click "OK" to create the document. There is now a fourth panel in RStudio, in the upper left. This is your R Markdown document:

The screenshot shows the RStudio interface with a new R Markdown document titled "Untitled1". The editor contains the following content:

```
1 ---
2 title: "Untitled"
3 output: html_document
4 ---
5
6 ```{r setup, include=FALSE}
7 knitr::opts_chunk$set(echo = TRUE)
8 ```
9
10 ## R Markdown
11
12 This is an R Markdown document. Markdown is a simple formatting syntax for
13 authoring HTML, PDF, and MS Word documents. For more details on using R Markdown
14 see <http://rmarkdown.rstudio.com>.
15
16 When you click the **Knit** button a document will be generated that includes
17 both content as well as the output of any embedded R code chunks within the
18 document. You can embed an R code chunk like this:
19
20 ```{r cars}
21
22 ```
```

The console shows the R startup message:

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

The file explorer on the right shows the "Home" directory with the following files and folders:

Name	Size	Modified
.Rhistory	295 B	Jan 5, 2020, 4:00 PM
Applications		
Desktop		
Documents		
Downloads		
Dropbox		
gamma.txt	1.6 KB	Aug 20, 2018, 5:15 PM
Library		
Movies		
Music		
normal.txt	1.6 KB	Aug 20, 2018, 5:11 PM
Pictures		
Public		
randomization.test.dat	661 B	Sep 5, 2018, 11:26 AM
IntroDataScience		

We can make some modifications to this file.

First, let's give this a title. We will call this "Getting Started With RStudio". At the top of the document, where it currently says title: "Untitled", we can replace "Untitled" with "Getting Started With RStudio."

Next, we can add author information. We can add a new line below the "title" line, and add an "author" attribute by typing

```
author: "Your Name"
```

The screenshot shows the RStudio interface with the modified R Markdown document. The editor contains the following content:

```
1 ---
2 title: "Getting Started With RStudio"
3 author: "J. McLean Slougher"
4 output: html_document
5 ---
```

Further down in the document you will see a bunch of additional content that was included by default. This is an example document to show you what these files should look like. We will next delete all of this content, and instead

add our own content.

Delete everything below line 6.

There are two different types of content that our document will include: written content / comments, and R code and outputs.

To structure written content, we can use headers. To create a header, we can write two pound signs followed by text. For regular text, we can just write it directly in the document. Enter the following in your document:

```
## This is a header  
We can write text here.
```

## Saving Files

As we create our document, we should make sure to save it. Click "File -> Save As".

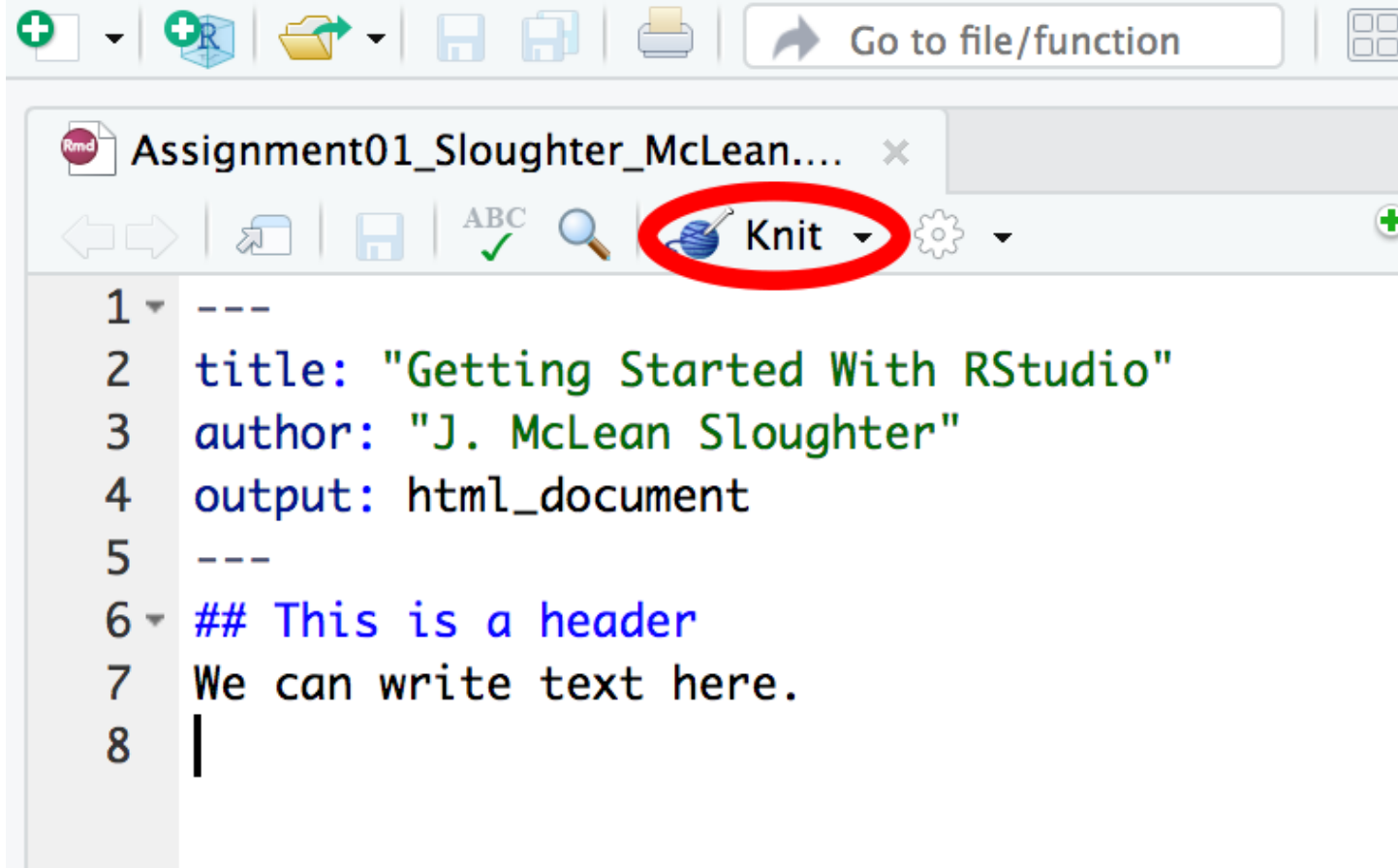
We want to save this in the "DATA 5300" folder that we created. Browse to that folder.

We need to give this document a name. Choose whatever name you feel is appropriate to help you keep track of this when you look for it later.

This file will be saved as a .Rmd file.

## Knitting

Once we are ready to generate our finished document, we click on the "knit" button. This will generate a finished HTML file that we can then use to show our work to others.



Click on the "knit" button. A new window will pop up showing the HTML file. You can see here how headers and text are formatted differently.

## Adding Code

Now let's return to our file, and add in some code as well.

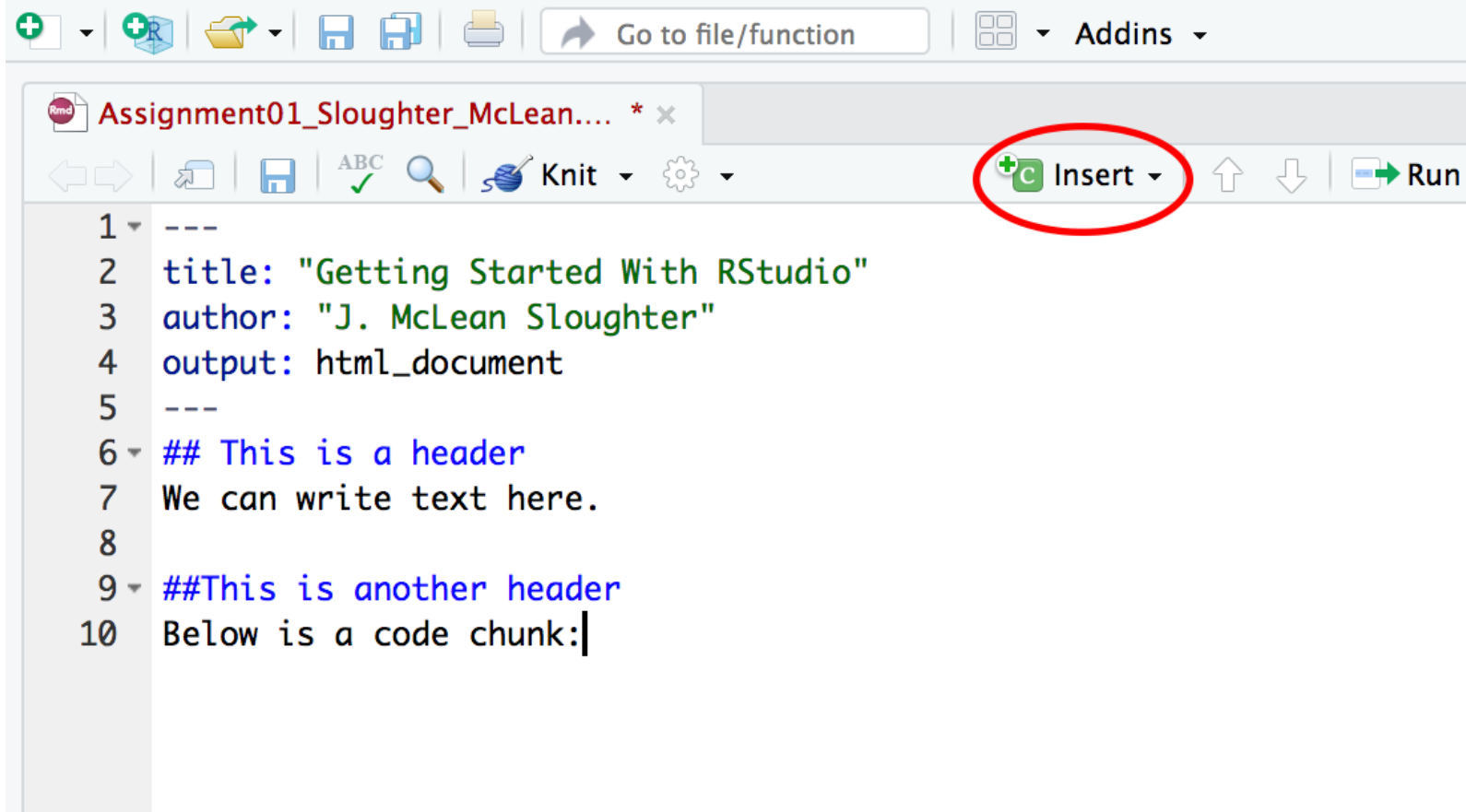
Close the window with the HTML file that popped up.

In the R Markdown panel, we will now add to our document. First, add a new header and text as follows:

```
## This is another header
Below is a code chunk:
```

Next, we will insert a section for R code.

Click on the "Insert" button, and then click on "R":



This will add a section where we can insert R code.

Let's create a variable named "x", and assign it a value of 8. We can do this with the following line of code, written into the code chunk that was created:

```
x <- 8
```

Next, let's create a variable named "y". This will be a vector (a list of several values). We will assign it the set of values 1, 7, and 20. We can do this on a new line in the code chunk, using the "c()" function in R, as follows:

```
y <- c(1, 7, 20)
```

So far, all that we are doing is displaying code. The code is not actually being run. If we want to run the code, we have a few options. We can click on the green arrow in the upper right corner of the code chunk. Or we can highlight the code, and then hit "Control-Enter" on a PC or "Command-Return" on a Mac.

Once we run the code, these variables have now been created, and are stored in our working environment. You will see them appear in the upper right panel:



The screenshot shows the RStudio interface. The main editor window displays a code chunk with the following content:

```
1 ---
2 title: "Getting Started With RStudio"
3 author: "J. McLean Sloughter"
4 output: html_document
5 ---
6 ## This is a header
7 We can write text here.
8
9 ## This is another header
10 Below is a code chunk:
11 ```{r}
12 k <- 8
13 y <- c(1, 7, 20)
14 ```
15
16
```

The Environment panel on the right shows the Global Environment with the following values:

Variable	Value
x	8
y	num [1:3] 1 7 20

The Files panel shows the project structure:

- Home > IntroDataScience
  - Assignment01\_Sloughter\_McLean.Rmd (228 B, Jan 5, 2020, 4:55 PM)
  - Assignment01\_Sloughter\_McLean.html (0 B, Jan 5, 2020, 4:55 PM)

Now that these variables are in our working environment, we can do things with them. For example, suppose the variable "y" was a set of temperatures measured in Celsius, and we wanted to create a new variable "z" that was those same temperatures converted to Fahrenheit. We could add the following code to a new line in the code chunk:

```
z <- 9/5 * y + 32
```

After we run our code with this added, we can see in our environment panel that there is now a variable "z" with values of 33.8, 44.6, and 68.

## Including Output in Documents

If you knit your document with these three lines of code included, you will see that your HTML document shows the three lines of code, but does not display the actual values of "z".

If we want to include these, we can add one additional line to our code chunk:

```
z
```

Giving a variable name without an assignment operator after it tells R that you want it to display the contents of the variable. If you knit your document with this fourth line added, you should now see a document that includes the output of displaying the "z" variable:

# Getting Started With RStudio

J. McLean Sloughter

## This is a header

We can write text here.

## This is another header

Below is a code chunk:

```
x <- 8
y <- c(1, 7, 20)
z <- 9/5 * y + 32
z
```

```
## [1] 33.8 44.6 68.0
```

You have now been introduced to the basics of generating an R Markdown document.

## Additional Mathematical Functions

You can use R code to calculate a variety of common mathematical functions. For example, if we wanted to calculate the square root of each z value, and store that as a new variable called "root.z", we could use the following code:

```
root.z <- sqrt(z)
```

Similarly, if we wanted to calculate the natural log of each z value, and store that as a new variable named "log.z", we could use the following code:

```
log.z <- log(z)
```

If we want to raise a value to an exponent, we can use "^":

```
z.squared <- z^2
```

## Packages

R's functionality can be expanded by loading packages that contain additional functions and/or data. The first time that you want to use a package, it needs to be installed. In RStudio, you can do this by clicking the "Packages" tab at the top of the lower-right pane. Then click the "Install" button. Type in the name of the package you want to install, and then click "Install".

A few packages we will make use of, that you should install, are "dplyr", "ggplot2", and "nycflights13".

Once a package has been installed, we also need to load it into our working environment. Installing a package only needs to be done once on your computer. But loading a package needs to be done every time you're working on a new project and want to use that package.

We'll add a code chunk to load the three packages we just installed, using the `library()` function:

```
library(dplyr)
library(ggplot2)
library(nycflights13)
```

## The Pipe Operator

Sometimes it useful to apply a series of functions sequentially. Suppose we create the following three mathematical functions:

```
f <- function(x)
{
  return(x^2)
}

g <- function(x)
{
  return(x+5)
}

h <- function(x)
{
  return(log(x))
}
```

Now suppose we start with a value of 3, and plug that into `f()`.

```
f(3)
```

```
## [1] 9
```

Then suppose we want to take the result, `f(3)`, and plug it into `g()`. If we had already worked out that `f(3) = 9`, we could find `g(9)`. But we could equivalently find `g(f(3))` - that is, the input to the function `g()` is the result of the function `f(3)`.

```
g(9)
```

```
## [1] 14
```

```
g(f(3))
```

```
## [1] 14
```

We could then plug this new result into `h()`, to find `h(g(f(3)))` - that is, the input to `h()` is the result of `g(f(3))`.

```
h(14)
```

```
## [1] 2.639057
```

```
h(g(f(3)))
```

```
## [1] 2.639057
```

This is a standard type of mathematical notation for “nested” functions, where we sequentially apply new functions to the output of previous functions. R can understand this notation. But it can sometimes be a bit unwieldy or unclear to read. In R, there is another option to do the same thing - we can use the pipe operator, included in the dplyr package; %>%

We use the pipe operator in R to let R know that we’re not done with our code on the current line, and want to apply some sort of additional function to the output of whatever function or object is on our current line of code.

When code is run in R, in general, it is run one line at a time, unless something in the code indicates to R that it should continue onto the next line. Adding %>% to the end of a line tells R to not just run that single line, but to look to the next line as well.

Rather than using the code f(3), we could equivalently do the following:

```
3 %>%  
  f()
```

```
## [1] 9
```

This is telling R to start with the number 3, and then to apply the function f(). This probably looks less clear than just writing f(3). But as we add more functions, it starts to make things easier to read.

If we again want to apply g() to the output of f(3), we would use this code:

```
3 %>%  
  f() %>%  
  g()
```

```
## [1] 14
```

This tells R to start with the number 3, to then apply f() to that value, and to then apply g() to the result.

We could similarly then add h() to the end as well:

```
3 %>%  
  f() %>%  
  g() %>%  
  h()
```

```
## [1] 2.639057
```

This starts with 3, applies f() to it, then applies g() to the result, and then applies h() to the result. Rather than having to read from the inside of parentheses to the outside with h(g(f(x))), we can just read the series of functions in order from one line to the next.

## Filter

Now, let’s look at how this idea applies to managing data. We will start by looking at the filter() function, also a part of the dplyr package.

In the following code, we will make use of a dataset named "flights" from the nycflights13 package. This is a data set on airline flights departing New York City in 2013.

We start by creating an object named portland\_flights. We then tell R to assign it the object "flights". But before we let R do so, the %>% at the end of the line tells is that we want it to apply a function to "flights".

The second line of code says to apply the function filter(), and specifies the argument dest == "PDX". This will filter the "flights" object to only use observations for which the destination was PDX.

To see what the resulting data object looks like, we could just type the name of the object, portland\_flights. But with large data objects, it is sometimes useful to just look at a piece of the dataset, **to see the structure without displaying every value. We can use the glimpse() function** to do this.

```
portland_flights <- flights %>%  
  filter(dest == "PDX")  
glimpse(portland_flights)
```

```
## Observations: 1,354  
## Variables: 19  
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...  
## $ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...  
## $ day       <int> 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, ...  
## $ dep_time  <int> 1739, 1805, 2052, 804, 1552, 1727, 1738, 2024, 1755, 1...  
## $ sched_dep_time <int> 1740, 1757, 2029, 805, 1550, 1720, 1740, 2029, 1745, 1...  
## $ dep_delay <dbl> -1, 8, 23, -1, 2, 7, -2, -5, 10, 47, -4, 0, 25, 16, 14...  
## $ arr_time  <int> 2051, 2117, 2349, 1039, 1853, 2042, 2028, 2314, 2110, ...  
## $ sched_arr_time <int> 2112, 2119, 2350, 1110, 1922, 2040, 2112, 2350, 2117, ...  
## $ arr_delay <dbl> -21, -2, -1, -31, -29, 2, -44, -36, -7, 19, -17, -23, ...  
## $ carrier   <chr> "DL", "UA", "B6", "UA", "DL", "UA", "DL", "B6", "DL", ...  
## $ flight    <int> 1339, 1152, 165, 423, 667, 784, 1339, 165, 1394, 1454,...  
## $ tailnum   <chr> "N3761R", "N39463", "N536JB", "N528UA", "N371DA", "N43...  
## $ origin    <chr> "JFK", "EWR", "JFK", "EWR", "JFK", "EWR", "JFK", "JFK"...  
## $ dest      <chr> "PDX", "PDX", "PDX", "PDX", "PDX", "PDX", "PDX", "PDX"...  
## $ air_time  <dbl> 341, 336, 331, 310, 305, 351, 322, 325, 325, 320, 330,...  
## $ distance  <dbl> 2454, 2434, 2454, 2434, 2454, 2434, 2454, 2454, 2454, ...  
## $ hour      <dbl> 17, 17, 20, 8, 15, 17, 17, 20, 17, 17, 20, 17, 17, 20,...  
## $ minute    <dbl> 40, 57, 29, 5, 50, 20, 40, 29, 45, 27, 29, 45, 30, 29,...  
## $ time_hour <dtm> 2013-01-01 17:00:00, 2013-01-01 17:00:00, 2013-01-01 ...
```

We can create more complicated filter rules. We can compare variables to particular values by using == to test for equality, or <, or >, or <=, or >=, or != (not equal).

And we can combine multiple comparison rules by using | for "or" and & for "and".

In the code below, we again filter the "flights" object. But now our filter is more complicated: we are looking for any observations in which the origin was JFK, the destination was either BTV or SEA, and the month was October or later:

```
btv_sea_flights_fall <- flights %>%  
  filter(origin == "JFK" & (dest == "BTV" | dest == "SEA") & month >= 10)  
glimpse(btv_sea_flights_fall)
```

```
## Observations: 815  
## Variables: 19  
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...  
## $ month     <int> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10...  
## $ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, ...  
## $ dep_time  <int> 729, 853, 916, 1216, 1452, 1459, 1754, 1825, 1925, 223...  
## $ sched_dep_time <int> 735, 900, 925, 1221, 1459, 1500, 1800, 1830, 1930, 224...  
## $ dep_delay <dbl> -6, -7, -9, -5, -7, -1, -6, -5, -5, -7, 0, -2, -7, -4,...  
## $ arr_time  <int> 1049, 1217, 1016, 1326, 1602, 1817, 2102, 2159, 2227, ...  
## $ sched_arr_time <int> 1040, 1157, 1033, 1328, 1622, 1829, 2103, 2150, 2250, ...  
## $ arr_delay <dbl> 9, 20, -17, -2, -20, -12, -1, 9, -23, -5, -14, 15, -7,...  
## $ carrier   <chr> "DL", "B6", "B6", "B6", "B6", "DL", "B6", "DL", "AA", ...
```

```
## $ flight      <int> 183, 63, 1634, 34, 1734, 161, 263, 442, 235, 234, 183,...
## $ tailnum     <chr> "N721TW", "N807JB", "N192JB", "N318JB", "N258JB", "N16...
## $ origin      <chr> "JFK", "JFK", "JFK", "JFK", "JFK", "JFK", "JFK", "JFK"...
## $ dest        <chr> "SEA", "SEA", "BTV", "BTV", "BTV", "SEA", "SEA", "SEA"...
## $ air_time    <dbl> 352, 362, 48, 49, 46, 348, 338, 366, 332, 48, 330, 344...
## $ distance    <dbl> 2422, 2422, 266, 266, 266, 2422, 2422, 2422, 2422, 266...
## $ hour        <dbl> 7, 9, 9, 12, 14, 15, 18, 18, 19, 22, 7, 9, 9, 12, 14, ...
## $ minute      <dbl> 35, 0, 25, 21, 59, 0, 0, 30, 30, 45, 35, 0, 25, 21, 59...
## $ time_hour   <dtm> 2013-10-01 07:00:00, 2013-10-01 09:00:00, 2013-10-01 ...
```

We can equivalently give it a list of conditions separated by commas, and it will filter for situations where all of the conditions are met:

```
btv_sea_flights_fall <- flights %>%
  filter(origin == "JFK", (dest == "BTV" | dest == "SEA"), month >= 10)
glimpse(btv_sea_flights_fall)
```

```
## Observations: 815
## Variables: 19
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
## $ month     <int> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10...
## $ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, ...
## $ dep_time  <int> 729, 853, 916, 1216, 1452, 1459, 1754, 1825, 1925, 223...
## $ sched_dep_time <int> 735, 900, 925, 1221, 1459, 1500, 1800, 1830, 1930, 224...
## $ dep_delay <dbl> -6, -7, -9, -5, -7, -1, -6, -5, -5, -7, 0, -2, -7, -4,...
## $ arr_time  <int> 1049, 1217, 1016, 1326, 1602, 1817, 2102, 2159, 2227, ...
## $ sched_arr_time <int> 1040, 1157, 1033, 1328, 1622, 1829, 2103, 2150, 2250, ...
## $ arr_delay <dbl> 9, 20, -17, -2, -20, -12, -1, 9, -23, -5, -14, 15, -7,...
## $ carrier   <chr> "DL", "B6", "B6", "B6", "B6", "DL", "B6", "DL", "AA", ...
## $ flight    <int> 183, 63, 1634, 34, 1734, 161, 263, 442, 235, 234, 183,...
## $ tailnum   <chr> "N721TW", "N807JB", "N192JB", "N318JB", "N258JB", "N16...
## $ origin    <chr> "JFK", "JFK", "JFK", "JFK", "JFK", "JFK", "JFK", "JFK"...
## $ dest      <chr> "SEA", "SEA", "BTV", "BTV", "BTV", "SEA", "SEA", "SEA"...
## $ air_time  <dbl> 352, 362, 48, 49, 46, 348, 338, 366, 332, 48, 330, 344...
## $ distance  <dbl> 2422, 2422, 266, 266, 266, 2422, 2422, 2422, 2422, 266...
## $ hour      <dbl> 7, 9, 9, 12, 14, 15, 18, 18, 19, 22, 7, 9, 9, 12, 14, ...
## $ minute    <dbl> 35, 0, 25, 21, 59, 0, 0, 30, 30, 45, 35, 0, 25, 21, 59...
## $ time_hour <dtm> 2013-10-01 07:00:00, 2013-10-01 09:00:00, 2013-10-01 ...
```

In R (as in many other coding languages), an exclamation mark is used to represent “not”. So if we set our filter condition as follows, we are now filtering for all flights where the destination is not BTV or SEA:

```
not_BTV_SEA <- flights %>%
  filter(!(dest == "BTV" | dest == "SEA"))
glimpse(not_BTV_SEA)
```

```
## Observations: 330,264
## Variables: 19
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
## $ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ dep_time  <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558,...
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600,...
## $ dep_delay <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, ...
## $ arr_time  <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849...
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851...
## $ arr_delay <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -...
## $ carrier   <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", ...
## $ flight    <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, ...
## $ tailnum   <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N39...
## $ origin    <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA"...
## $ dest      <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD"...
## $ air_time  <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, ...
## $ distance  <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733,...
## $ hour      <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, ...
## $ minute    <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 59, ...
## $ time_hour <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 ...
```

Note the parentheses above. R first looks at the condition inside the parentheses to find which observations have a destination of either BTV or SEA. Then the exclamation mark in front of the parentheses tells R to find all the situations where that combined condition is not met.

Without the parentheses, we would have a different filter. In the code below, we will get all observations for which either the destination was not BTV, or for which the destination was SEA:

```
not_BTV_SEA <- flights %>%  
  filter(!dest == "BTV" | dest == "SEA")  
glimpse(not_BTV_SEA)
```

```
## Observations: 334,187  
## Variables: 19  
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...  
## $ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...  
## $ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...  
## $ dep_time  <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ...  
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ...  
## $ dep_delay <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -2, ...  
## $ arr_time  <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849, ...  
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851, ...  
## $ arr_delay <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -2, ...  
## $ carrier   <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", ...  
## $ flight    <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, ...  
## $ tailnum   <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N39...  
## $ origin    <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA"...  
## $ dest      <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD"...  
## $ air_time  <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, ...  
## $ distance  <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ...  
## $ hour      <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, ...  
## $ minute    <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 59, ...  
## $ time_hour <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 ...
```

Suppose we wanted to get a subset of our data that included several destinations. We could use `|` repeatedly to say we want the destination to be SEA or SFO or PDX or BTV or BDL:

```
many_airports <- flights %>%  
  filter(dest == "SEA" | dest == "SFO" | dest == "PDX" |  
         dest == "BTV" | dest == "BDL")  
glimpse(many_airports)
```

```
## Observations: 21,640  
## Variables: 19  
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...  
## $ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...  
## $ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...  
## $ dep_time  <int> 558, 611, 655, 724, 729, 734, 743, 745, 746, 803, 826, ...  
## $ sched_dep_time <int> 600, 600, 700, 725, 730, 737, 730, 745, 746, 800, 817, ...  
## $ dep_delay <dbl> -2, 11, -5, -1, -1, -3, 13, 0, 0, 3, 9, 6, -2, -1, 1, ...  
## $ arr_time  <int> 923, 945, 1037, 1020, 1049, 1047, 1059, 1135, 1119, 11...  
## $ sched_arr_time <int> 937, 931, 1045, 1030, 1115, 1113, 1056, 1125, 1129, 11...  
## $ arr_delay <dbl> -14, 14, -8, -10, -26, -26, 3, 10, -10, -12, -13, -25, ...  
## $ carrier   <chr> "UA", "UA", "DL", "AS", "VX", "B6", "DL", "AA", "UA", ...  
## $ flight    <int> 1124, 303, 1865, 11, 11, 643, 495, 59, 1668, 223, 1480...  
## $ tailnum   <chr> "N53441", "N532UA", "N705TW", "N594AS", "N635VA", "N62...  
## $ origin    <chr> "EWR", "JFK", "JFK", "EWR", "JFK", "JFK", "JFK", "JFK"...  
## $ dest      <chr> "SFO", "SFO", "SFO", "SEA", "SFO", "SFO", "SEA", "SFO"...  
## $ air_time  <dbl> 361, 366, 362, 338, 356, 350, 349, 378, 373, 369, 357, ...  
## $ distance  <dbl> 2565, 2586, 2586, 2402, 2586, 2586, 2422, 2586, 2565, ...  
## $ hour      <dbl> 6, 6, 7, 7, 7, 7, 7, 7, 8, 8, 8, 9, 10, 10, 10, 11, ...  
## $ minute    <dbl> 0, 0, 0, 25, 30, 37, 30, 45, 46, 0, 17, 51, 10, 30, 30...  
## $ time_hour <dtm> 2013-01-01 06:00:00, 2013-01-01 06:00:00, 2013-01-01 ...
```

Or we could use a different type of comparison. Instead of checking for equality against five different options, we could check each observation to see if its destination was in a specified list of options, using `%in%`



```
many_airports <- flights %>%
  filter(dest %in% c("SEA", "SFO", "PDX", "BTV", "BDL"))
glimpse(many_airports)
```

```
## Observations: 21,640
## Variables: 19
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
## $ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ dep_time  <int> 558, 611, 655, 724, 729, 734, 743, 745, 746, 803, 826,...
## $ sched_dep_time <int> 600, 600, 700, 725, 730, 737, 730, 745, 746, 800, 817,...
## $ dep_delay  <dbl> -2, 11, -5, -1, -1, -3, 13, 0, 0, 3, 9, 6, -2, -1, 1, ...
## $ arr_time   <int> 923, 945, 1037, 1020, 1049, 1047, 1059, 1135, 1119, 11...
## $ sched_arr_time <int> 937, 931, 1045, 1030, 1115, 1113, 1056, 1125, 1129, 11...
## $ arr_delay  <dbl> -14, 14, -8, -10, -26, -26, 3, 10, -10, -12, -13, -25,...
## $ carrier    <chr> "UA", "UA", "DL", "AS", "VX", "B6", "DL", "AA", "UA", ...
## $ flight     <int> 1124, 303, 1865, 11, 11, 643, 495, 59, 1668, 223, 1480...
## $ tailnum    <chr> "N53441", "N532UA", "N705TW", "N594AS", "N635VA", "N62...
## $ origin     <chr> "EWR", "JFK", "JFK", "EWR", "JFK", "JFK", "JFK", "JFK"...
## $ dest       <chr> "SFO", "SFO", "SFO", "SEA", "SFO", "SFO", "SEA", "SFO"...
## $ air_time   <dbl> 361, 366, 362, 338, 356, 350, 349, 378, 373, 369, 357,...
## $ distance   <dbl> 2565, 2586, 2586, 2402, 2586, 2586, 2422, 2586, 2565, ...
## $ hour       <dbl> 6, 6, 7, 7, 7, 7, 7, 7, 8, 8, 8, 9, 10, 10, 10, 11,...
## $ minute     <dbl> 0, 0, 0, 25, 30, 37, 30, 45, 46, 0, 17, 51, 10, 30, 30...
## $ time_hour  <dtm> 2013-01-01 06:00:00, 2013-01-01 06:00:00, 2013-01-01 ...
```

## Summarize

We can **create numerical summaries using the `summarize()` function**, also part of the dplyr package. Suppose we were working with the weather dataset included in `nycflights13`, and wanted **to give the mean and standard deviation** for temperature. We could use the following code:

```
summary_temp <- weather %>%
  summarize(mean = mean(temp), std_dev = sd(temp))
summary_temp
```

```
## # A tibble: 1 x 2
##   mean std_dev
##   <dbl>   <dbl>
## 1    NA      NA
```

This output doesn't look very useful. **"NA" is used by R to denote missing values**. Any time we try to do a numerical calculation on a set of data with missing values, by default, R will give a result of NA.

If we want it to apply the calculation only to the non-missing values, we have to specify that, by telling R to remove any NA values in the data set before conducting its calculations:

```
summary_temp <- weather %>%
  summarize(mean = mean(temp, na.rm = TRUE),
            std_dev = sd(temp, na.rm = TRUE))
summary_temp
```

```
## # A tibble: 1 x 2
##   mean std_dev
##   <dbl>   <dbl>
## 1  55.3    17.8
```

Now we get values that can tell us something - the average temperature in our data set was around 55 degrees, and a typical observation would be about 18 degrees away from the average.



While the mean and standard deviation are two of the most common numerical summaries that someone might be interested in, the `summarize()` function can make us of any function that takes an input of a set of data and returns a single numerical summary.

## Group

We know that temperatures change over the course of the year. Rather than just summarizing all temperatures, we might want to group our data by month, and then calculate numerical summaries for each month.

We can use the `group_by()` function, also in the `dplyr` package, to first split our data into groups before we apply the `summarize()` function:

```
summary_monthly_temp <- weather %>%
  group_by(month) %>%
  summarize(mean = mean(temp, na.rm = TRUE),
            std_dev = sd(temp, na.rm = TRUE))
summary_monthly_temp
```

```
## # A tibble: 12 x 3
##   month mean std_dev
##   <int> <dbl> <dbl>
## 1     1  35.6  10.2
## 2     2  34.3   6.98
## 3     3  39.9   6.25
## 4     4  51.7   8.79
## 5     5  61.8   9.68
## 6     6  72.2   7.55
## 7     7  80.1   7.12
## 8     8  74.5   5.19
## 9     9  67.4   8.47
## 10    10  60.1   8.85
## 11    11  45.0  10.4
## 12    12  38.4   9.98
```

When we use the `group_by()` function, it doesn't change our actual data, but it changes the meta-data, the information stored about the data set. Compare the results of the following two pieces of code:

```
weather
```

```
## # A tibble: 26,115 x 15
##   origin year month day hour temp dewp humid wind_dir wind_speed
##   <chr>   <int> <int> <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 EWR    2013     1     1     1  39.0  26.1  59.4    270    10.4
## 2 EWR    2013     1     1     2  39.0  27.0  61.6    250     8.06
## 3 EWR    2013     1     1     3  39.0  28.0  64.4    240    11.5
## 4 EWR    2013     1     1     4  39.9  28.0  62.2    250    12.7
## 5 EWR    2013     1     1     5  39.0  28.0  64.4    260    12.7
## 6 EWR    2013     1     1     6  37.9  28.0  67.2    240    11.5
## 7 EWR    2013     1     1     7  39.0  28.0  64.4    240    15.0
## 8 EWR    2013     1     1     8  39.9  28.0  62.2    250    10.4
## 9 EWR    2013     1     1     9  39.9  28.0  62.2    260    15.0
## 10 EWR   2013     1     1    10  41    28.0  59.6    260    13.8
## # ... with 26,105 more rows, and 5 more variables: wind_gust <dbl>, precip <dbl>,
## #   pressure <dbl>, visib <dbl>, time_hour <dtm>
```

```
weather %>%
  group_by(month)
```

```
## # A tibble: 26,115 x 15
## # Groups:   month [12]
##   origin year month day hour temp dewp humid wind_dir wind_speed
```

```
##      <chr>      <int> <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 EWR      2013      1      1      1 39.0 26.1 59.4      270      10.4
## 2 EWR      2013      1      1      2 39.0 27.0 61.6      250       8.06
## 3 EWR      2013      1      1      3 39.0 28.0 64.4      240      11.5
## 4 EWR      2013      1      1      4 39.9 28.0 62.2      250      12.7
## 5 EWR      2013      1      1      5 39.0 28.0 64.4      260      12.7
## 6 EWR      2013      1      1      6 37.9 28.0 67.2      240      11.5
## 7 EWR      2013      1      1      7 39.0 28.0 64.4      240      15.0
## 8 EWR      2013      1      1      8 39.9 28.0 62.2      250      10.4
## 9 EWR      2013      1      1      9 39.9 28.0 62.2      260      15.0
## 10 EWR     2013      1      1     10 41    28.0 59.6      260      13.8
## # ... with 26,105 more rows, and 5 more variables: wind_gust <dbl>, precip <dbl>,
## #   pressure <dbl>, visib <dbl>, time_hour <dtm>
```

The data is the same in each case, but in the second situation, there is now an additional row at the top of our summary:

```
# Groups: month [12]
```

This change to the meta-data means that any resulting operation performed on the data will be based on the specified grouping. If we want to remove the grouping from the meta-data, we can use the `ungroup()` function:

```
weather %>%
  group_by(month) %>%
  ungroup()
```

```
## # A tibble: 26,115 x 15
##   origin year month   day hour temp dewp humid wind_dir wind_speed
##   <chr>   <int> <int> <int> <int> <dbl> <dbl> <dbl>   <dbl>    <dbl>
## 1 EWR    2013      1      1      1 39.0 26.1 59.4      270      10.4
## 2 EWR    2013      1      1      2 39.0 27.0 61.6      250       8.06
## 3 EWR    2013      1      1      3 39.0 28.0 64.4      240      11.5
## 4 EWR    2013      1      1      4 39.9 28.0 62.2      250      12.7
## 5 EWR    2013      1      1      5 39.0 28.0 64.4      260      12.7
## 6 EWR    2013      1      1      6 37.9 28.0 67.2      240      11.5
## 7 EWR    2013      1      1      7 39.0 28.0 64.4      240      15.0
## 8 EWR    2013      1      1      8 39.9 28.0 62.2      250      10.4
## 9 EWR    2013      1      1      9 39.9 28.0 62.2      260      15.0
## 10 EWR   2013      1      1     10 41    28.0 59.6      260      13.8
## # ... with 26,105 more rows, and 5 more variables: wind_gust <dbl>, precip <dbl>,
## #   pressure <dbl>, visib <dbl>, time_hour <dtm>
```

One common application of this grouping structure could be to count how many observations are in each group. The function `n()` counts how many data points are in a particular object. Suppose we wanted to count how many data points are in our flights data, grouped by origin airport:

```
by_origin <- flights %>%
  group_by(origin) %>%
  summarize(count = n())
by_origin
```

```
## # A tibble: 3 x 2
##   origin count
##   <chr>   <int>
## 1 EWR    120835
## 2 JFK    111279
## 3 LGA    104662
```

We see that our dataset includes 120,835 flights from EWR, 111,279 flights from JFK, and 104,662 flights from LGA.

In some cases, we might want to group by more than one variable. Maybe we want to look at numbers of flights per origin airport per month. We can give multiple arguments to the `group_by()` function:

```
by_origin_monthly <- flights %>%
  group_by(origin, month) %>%
  summarize(count = n())
by_origin_monthly
```

```
## # A tibble: 36 x 3
## # Groups:   origin [3]
##   origin month count
##   <chr>   <int> <int>
## 1 EWR      1  9893
## 2 EWR      2  9107
## 3 EWR      3 10420
## 4 EWR      4 10531
## 5 EWR      5 10592
## 6 EWR      6 10175
## 7 EWR      7 10475
## 8 EWR      8 10359
## 9 EWR      9  9550
## 10 EWR     10 10104
## # ... with 26 more rows
```

## Mutate

In many situations, it is useful to be able to create a new variable based on one or more existing variables. This can be done in R via the `mutate()` function, again part of the `dplyr` package.

Our temperature data is currently measured in Fahrenheit. Suppose we wanted to create a new variable measuring temperature in Celsius. We could use the following code:

```
weather <- weather %>%
  mutate(temp_in_C = (temp - 32) / 1.8)
```

This creates a new variable, `temp_in_C`, and adds it to our data frame.

We might also want to create a variable based on more than one existing variable. For our flight data, we might be interested in the gain for each flight - that is, how much quicker the flight ended up being than planned. We can find the net gain by looking at the difference in departure and arrival delays:

```
flights <- flights %>%
  mutate(gain = dep_delay - arr_delay)
```

We can create more than one new variable in the same `mutate()` function, and we can make use of a newly created variable in creating other variables:

```
flights <- flights %>%
  mutate(
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours
  )
```

What are these new variables telling us?

## Arrange

It is quite common to want to sort our data set based on one of the variables.

Suppose we are examining how frequently each destination airport appears in our data set. We could create a new object recording these frequencies:

```
freq_dest <- flights %>%  
  group_by(dest) %>%  
  summarize(num_flights = n())  
freq_dest
```

```
## # A tibble: 105 x 2  
##   dest num_flights  
##   <chr>      <int>  
## 1 ABQ         254  
## 2 ACK         265  
## 3 ALB         439  
## 4 ANC          8  
## 5 ATL       17215  
## 6 AUS       2439  
## 7 AVL         275  
## 8 BDL         443  
## 9 BGR         375  
## 10 BHM        297  
## # ... with 95 more rows
```

When we look at this new `freq_dest` object, we see that it is sorted alphabetically by airport name. In some situations, that might be the most useful way to sort it. But at other times, we might want to sort from most frequent destination to least frequent destination.

The `arrange()` function lets us sort a data frame based on a specified variable. Consider the following code:

```
freq_dest %>%  
  arrange(num_flights)
```

```
## # A tibble: 105 x 2  
##   dest num_flights  
##   <chr>      <int>  
## 1 LEX          1  
## 2 LGA          1  
## 3 ANC          8  
## 4 SBN         10  
## 5 HDN         15  
## 6 MTJ         15  
## 7 EYW         17  
## 8 PSP         19  
## 9 JAC         25  
## 10 BZN        36  
## # ... with 95 more rows
```

Did this give us what we want? Almost, but not quite. The data was arranged by frequency, but it was arranged from least frequent to most frequent.

If we want to reverse that, we can specify to use descending order rather than ascending order:

```
freq_dest %>%  
  arrange(desc(num_flights))
```

```
## # A tibble: 105 x 2  
##   dest num_flights  
##   <chr>      <int>  
## 1 ORD       17283  
## 2 ATL       17215  
## 3 LAX       16174  
## 4 BOS       15508  
## 5 MCO       14082  
## 6 CLT       14064  
## 7 SFO       13331
```

```
## 8 FLL 12055
## 9 MIA 11728
## 10 DCA 9705
## # ... with 95 more rows
```

## Join

Another common data wrangling task is to combine information from multiple data frames into a single data frame.

We can do this when we have two data frames that have a variable in common that can be used to match entries between the two frames. Our flights data includes carrier codes, but not the full names of the airlines. The “airlines” data frame also includes carrier codes, but in addition it has full airline names.

```
flights_joined <- flights %>%
  inner_join(airlines, by = "carrier")
glimpse(flights)
```

```
## Observations: 336,776
## Variables: 22
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
## $ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ dep_time  <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558,...
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600,...
## $ dep_delay <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -...
## $ arr_time  <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849...
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851...
## $ arr_delay <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -...
## $ carrier   <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", ...
## $ flight    <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, ...
## $ tailnum   <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N39...
## $ origin    <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA"...
## $ dest      <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD"...
## $ air_time  <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, ...
## $ distance  <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733,...
## $ hour      <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, ...
## $ minute    <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 59, ...
## $ time_hour <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 ...
## $ gain      <dbl> -9, -16, -31, 17, 19, -16, -24, 11, 5, -10, 0, 1, -9, ...
## $ hours     <dbl> 3.7833333, 3.7833333, 2.6666667, 3.0500000, 1.9333333,...
## $ gain_per_hour <dbl> -2.3788546, -4.2290749, -11.6250000, 5.5737705, 9.8275...
```

```
glimpse(flights_joined)
```

```
## Observations: 336,776
## Variables: 23
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
## $ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ dep_time  <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558,...
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600,...
## $ dep_delay <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -...
## $ arr_time  <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849...
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851...
## $ arr_delay <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -...
## $ carrier   <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", ...
## $ flight    <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, ...
## $ tailnum   <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N39...
## $ origin    <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA"...
## $ dest      <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD"...
## $ air_time  <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, ...
## $ distance  <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733,...
## $ hour      <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, ...
## $ minute    <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 59, ...
## $ time_hour <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 ...
## $ gain      <dbl> -9, -16, -31, 17, 19, -16, -24, 11, 5, -10, 0, 1, -9, ...
## $ hours     <dbl> 3.7833333, 3.7833333, 2.6666667, 3.0500000, 1.9333333,...
```

```
## $ gain_per_hour <dbl> -2.3788546, -4.2290749, -11.6250000, 5.5737705, 9.8275...
## $ name <chr> "United Air Lines Inc.", "United Air Lines Inc.", "Ame...
```

In the above code, we create a new data frame named `flights_joined`. We start with the `flights` data frame, and then perform an “inner join” with the `airlines` data frame, looking for matching values of “carrier”. This results in a new data frame that combines the information from the `flights` and `airlines` data frames.

There are several types of joins. An “inner join” means to only include instances in which the matching term appears in both data sets. If we had an airline code included in “airlines” that did not have any flights out of New York City in 2013 (and therefore was not included in the “flights” data set), then it would not appear in our new “`flights_joined`” dataset.

A join is easiest to do if both data frames use the same name for the shared variable. But if the names are different, we can still join them:

```
flights_with_airport_names <- flights %>%
  inner_join(airports, by = c("dest" = "faa"))
glimpse(flights_with_airport_names)
```

```
## Observations: 329,174
## Variables: 29
## $ year <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
## $ month <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ dep_time <int> 517, 533, 542, 554, 554, 555, 557, 557, 558, 558, 558, 558, 558, 558, ...
## $ sched_dep_time <int> 515, 529, 540, 600, 558, 600, 600, 600, 600, 600, 600, 600, 600, 600, ...
## $ dep_delay <dbl> 2, 4, 2, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1, 0...
## $ arr_time <int> 830, 850, 923, 812, 740, 913, 709, 838, 753, 849, 853, 853, 853, ...
## $ sched_arr_time <int> 819, 830, 850, 837, 728, 854, 723, 846, 745, 851, 856, 856, 856, ...
## $ arr_delay <dbl> 11, 20, 33, -25, 12, 19, -14, -8, 8, -2, -3, 7, -14, 3...
## $ carrier <chr> "UA", "UA", "AA", "DL", "UA", "B6", "EV", "B6", "AA", ...
## $ flight <int> 1545, 1714, 1141, 461, 1696, 507, 5708, 79, 301, 49, 7...
## $ tailnum <chr> "N14228", "N24211", "N619AA", "N668DN", "N39463", "N51...
## $ origin <chr> "EWR", "LGA", "JFK", "LGA", "EWR", "EWR", "LGA", "JFK"...
## $ dest <chr> "IAH", "IAH", "MIA", "ATL", "ORD", "FLL", "IAD", "MCO"...
## $ air_time <dbl> 227, 227, 160, 116, 150, 158, 53, 140, 138, 149, 158, ...
## $ distance <dbl> 1400, 1416, 1089, 762, 719, 1065, 229, 944, 733, 1028, ...
## $ hour <dbl> 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, ...
## $ minute <dbl> 15, 29, 40, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0, ...
## $ time_hour <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 ...
## $ gain <dbl> -9, -16, -31, 19, -16, -24, 11, 5, -10, 0, 1, -9, 12, ...
## $ hours <dbl> 3.7833333, 3.7833333, 2.6666667, 1.9333333, 2.5000000, ...
## $ gain_per_hour <dbl> -2.3788546, -4.2290749, -11.6250000, 9.8275862, -6.400...
## $ name <chr> "George Bush Intercontinental", "George Bush Intercont...
## $ lat <dbl> 29.98443, 29.98443, 25.79325, 33.63672, 41.97860, 26.0...
## $ lon <dbl> -95.34144, -95.34144, -80.29056, -84.42807, -87.90484, ...
## $ alt <dbl> 97, 97, 8, 1026, 668, 9, 313, 96, 668, 19, 26, 126, 13...
## $ tz <dbl> -6, -6, -5, -5, -6, -5, -5, -5, -6, -5, -5, -8, -8, -6...
## $ dst <chr> "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", ...
## $ tzone <chr> "America/Chicago", "America/Chicago", "America/New_Yor...
```

Now we are joining the `flights` and `airports` data frames, matching the “dest” object in “flights” with the “faa” object in “airports”.

Consider the following code, which combines several of the things we have learned about in this section. What is this code doing?

```
named_dests <- flights %>%
  group_by(dest) %>%
  summarize(num_flights = n()) %>%
  arrange(desc(num_flights)) %>%
  inner_join(airports, by = c("dest" = "faa")) %>%
```

```
rename(airport_name = name)
named_dests
```

```
## # A tibble: 101 x 9
##   dest num_flights airport_name      lat   lon alt   tz dst tzone
##   <chr>   <int> <chr>         <dbl> <dbl> <dbl> <dbl> <chr> <chr>
## 1 ORD     17283 Chicago Ohare Intl  42.0  -87.9  668   -6 A  America...
## 2 ATL     17215 Hartsfield Jackson... 33.6  -84.4 1026   -5 A  America...
## 3 LAX     16174 Los Angeles Intl   33.9 -118.   126   -8 A  America...
## 4 BOS     15508 General Edward Law... 42.4  -71.0   19   -5 A  America...
## 5 MCO     14082 Orlando Intl       28.4  -81.3   96   -5 A  America...
## 6 CLT     14064 Charlotte Douglas ... 35.2  -80.9  748   -5 A  America...
## 7 SFO     13331 San Francisco Intl  37.6 -122.   13   -8 A  America...
## 8 FLL     12055 Fort Lauderdale Ho... 26.1  -80.2    9   -5 A  America...
## 9 MIA     11728 Miami Intl       25.8  -80.3    8   -5 A  America...
## 10 DCA     9705 Ronald Reagan Wash... 38.9  -77.0   15   -5 A  America...
## # ... with 91 more rows
```

We could also consider situations where we need to match between two data frames not just based on a single variable, but instead based on a combination of variables.

Suppose we wanted to combine weather data with flight data. We need to match based on the time (specified by year, month, day, and hour) as well as the location:

```
flights_weather_joined <- flights %>%
  inner_join(weather, by = c("year", "month", "day", "hour", "origin"))
glimpse(flights_weather_joined)
```

```
## Observations: 335,220
## Variables: 33
## $ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
## $ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ dep_time  <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558,...
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600,...
## $ dep_delay <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -...
## $ arr_time  <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849...
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851...
## $ arr_delay <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -...
## $ carrier   <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", ...
## $ flight    <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, ...
## $ tailnum   <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N39...
## $ origin    <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA"...
## $ dest      <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD"...
## $ air_time  <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, ...
## $ distance  <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733,...
## $ hour      <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, ...
## $ minute    <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 59, ...
## $ time_hour.x <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 ...
## $ gain      <dbl> -9, -16, -31, 17, 19, -16, -24, 11, 5, -10, 0, 1, -9, ...
## $ hours     <dbl> 3.7833333, 3.7833333, 2.6666667, 3.0500000, 1.9333333,...
## $ gain_per_hour <dbl> -2.3788546, -4.2290749, -11.6250000, 5.5737705, 9.8275...
## $ temp      <dbl> 39.02, 39.92, 39.02, 39.02, 39.92, 39.02, 39.02, 37.94, 39.92...
## $ dewp      <dbl> 28.04, 24.98, 26.96, 26.96, 24.98, 28.04, 28.04, 24.98...
## $ humid     <dbl> 64.43, 54.81, 61.63, 61.63, 54.81, 64.43, 67.21, 54.81...
## $ wind_dir   <dbl> 260, 250, 260, 260, 260, 260, 240, 260, 260, 260, ...
## $ wind_speed <dbl> 12.65858, 14.96014, 14.96014, 14.96014, 16.11092, 12.6...
## $ wind_gust  <dbl> NA, 21.86482, NA, NA, 23.01560, NA, NA, 23.01560, NA, ...
## $ precip    <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ pressure  <dbl> 1011.9, 1011.4, 1012.1, 1012.1, 1011.7, 1011.9, 1012.4...
## $ visib     <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, ...
## $ time_hour.y <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 ...
## $ temp_in_C <dbl> 3.9, 4.4, 3.9, 3.9, 4.4, 3.9, 3.3, 4.4, 3.3, 4.4, 3.3,...
```

## Select



We might want to select only a subset of our variables, to give us something simpler to look at.

Suppose we wanted to just look at the carrier and flight variables in the flights data frame:

```
flights %>%  
  select(carrier, flight)
```

```
## # A tibble: 336,776 x 2  
##   carrier flight  
##   <chr>   <int>  
## 1 UA      1545  
## 2 UA      1714  
## 3 AA      1141  
## 4 B6       725  
## 5 DL       461  
## 6 UA      1696  
## 7 B6       507  
## 8 EV      5708  
## 9 B6        79  
## 10 AA      301  
## # ... with 336,766 more rows
```

Or suppose we wanted to leave out one variable:

```
flights_no_year <- flights %>% select(-year)
```

## Importing Data

So far, we have either used data frames that were part of a package, or we have created our own data frames to work with. But much of the time, in the real world, we will need to read in data from a file. Data files could be formatted in a variety of ways. Two of the most common are an Excel spreadsheet, or a .csv file (“csv” stands for “comma separated values”, and refers to a standard formatting convention where data is stored as a text file with commas to separate entries).

If we want to read in a .csv file, we can use the function `read_csv()` in the `readr` package. We need to tell the `read_csv()` function where to find the file. In our first example, suppose we are using a file that is available online, and we have the web address for it:

```
library(readr)  
dem_score <- read_csv("https://moderndive.com/data/dem_score.csv")
```

```
## `curl` package not installed, falling back to using `url()`
```

```
## Parsed with column specification:  
## cols(  
##   country = col_character(),  
##   `1952` = col_double(),  
##   `1957` = col_double(),  
##   `1962` = col_double(),  
##   `1967` = col_double(),  
##   `1972` = col_double(),  
##   `1977` = col_double(),  
##   `1982` = col_double(),  
##   `1987` = col_double(),  
##   `1992` = col_double()  
## )
```




```
dem_score
```

```
## # A tibble: 96 x 10
##   country    `1952` `1957` `1962` `1967` `1972` `1977` `1982` `1987` `1992`
##   <chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Albania    -9     -9     -9     -9     -9     -9     -9     -9     5
## 2 Argentina  -9     -1     -1     -9     -9     -9     -8     8     7
## 3 Armenia    -9     -7     -7     -7     -7     -7     -7     -7     7
## 4 Australia  10     10     10     10     10     10     10     10    10
## 5 Austria    10     10     10     10     10     10     10     10    10
## 6 Azerbaijan -9     -7     -7     -7     -7     -7     -7     -7     1
## 7 Belarus    -9     -7     -7     -7     -7     -7     -7     -7     7
## 8 Belgium    10     10     10     10     10     10     10     10    10
## 9 Bhutan     -10    -10    -10    -10    -10    -10    -10    -10   -10
## 10 Bolivia    -4     -3     -3     -4     -7     -7     8     9     9
## # ... with 86 more rows
```

This file gives the democracy ratings of various countries across time.

We could also load data from a file stored locally on our computer. Go to

[https://moderndive.com/data/dem\\_score.xlsx](https://moderndive.com/data/dem_score.xlsx)  ([https://moderndive.com/data/dem\\_score.xlsx](https://moderndive.com/data/dem_score.xlsx)) and save the file onto your computer. Then in the “files” pane of RStudio (the lower right pane), find the directory where you saved the file, click on it, and click “Import Dataset”. A preview window will pop up, giving you some options to specify how the file should be imported. Once you click the button in the lower right hand corner marked “Import”, it will load the data into an object in R.

## Tidy Data

Let's load in some additional packages. Make sure to install them first.

```
library(tidyr)
```

```
library(fivethirtyeight)
```

“Tidy” data refers to particular formatting conventions for storing data such that your data frames will behave correctly with a variety of functions.

We will illustrate what tidy data looks like using an example dataset from the fivethirtyeight package, a collection of datasets used in various reports from the data journalism website FiveThirtyEight.com

In particular, here, we will use the “drinks” data set from this package. This data set contains measurements of typical quantities of beer, wine, and spirits consumed in a variety of countries.

```
glimpse(drinks)
```

```
## Observations: 193
## Variables: 5
## $ country      <chr> "Afghanistan", "Albania", "Algeria", "An...
## $ beer_servings <int> 0, 89, 25, 245, 217, 102, 193, 21, 261, ...
## $ spirit_servings <int> 0, 132, 0, 138, 57, 128, 25, 179, 72, 75...
## $ wine_servings <int> 0, 54, 14, 312, 45, 45, 221, 11, 212, 19...
## $ total_litres_of_pure_alcohol <dbl> 0.0, 4.9, 0.7, 12.4, 5.9, 4.9, 8.3, 3.8,...
```

Let's start by using some of the techniques we have just learned to filter the data. Let's look at just data from the U.S.A., China, Italy, and Saudi Arabia. Let's exclude the total litres of pure alcohol variable. And let's rename the other variables to just “beer”, “spirit”, and “wine”:

```
drinks_smaller <- drinks %>%
  filter(country %in% c("USA", "China", "Italy", "Saudi Arabia")) %>%
```

```
select(-total_litres_of_pure_alcohol) %>%
  rename(beer = beer_servings, spirit = spirit_servings, wine = wine_servings)
drinks_smaller
```

```
## # A tibble: 4 x 4
##   country      beer spirit  wine
##   <chr>      <int> <int> <int>
## 1 China         79   192     8
## 2 Italy          85    42   237
## 3 Saudi Arabia    0     5     0
## 4 USA          249   158    84
```

Note that we don't have drink type stored as a variable here - instead, we have separate variables for each drink type.

This is the key idea of tidy data. Rather than having a separate column / variable for each drink type, there should be one variable to record the typical number of servings, and another variable to record which drink type is being measured in a given row. That is, rather than each row representing everything about a country, each row will represent a unique drink and country combination - each row is an observation, rather than a set of observations.

Converting to tidy format can be accomplished by the `pivot_longer()` function. Consider the following code:

```
drinks_smaller_tidy <- drinks_smaller %>%
  pivot_longer(names_to = "type",
               values_to = "servings",
               cols = -country)
drinks_smaller_tidy
```

```
## # A tibble: 12 x 3
##   country      type  servings
##   <chr>      <chr>    <int>
## 1 China      beer        79
## 2 China      spirit       192
## 3 China      wine         8
## 4 Italy       beer        85
## 5 Italy       spirit        42
## 6 Italy       wine       237
## 7 Saudi Arabia beer         0
## 8 Saudi Arabia spirit         5
## 9 Saudi Arabia wine         0
## 10 USA        beer       249
## 11 USA        spirit      158
## 12 USA        wine        84
```

This tells R that we want to create a new column named “type” that will list what the name was of each column in the original data this particular observation came from, a new column named “servings” that records the numerical values from the original data, and that the columns that we want to “tidy” from the original data are everything except “country”.

## tidyverse

There are many R packages all built around the idea of working with tidy data. Many of them have been collected into a single package, called “tidyverse”. Rather than separately loading each package, if we load the tidyverse package, it will load them all at once for us, including the ggplot2, dplyr, readr, and tidyr packages that we've worked with.