

01-05: Histograms

1 - Purpose

- Creating histograms and stacked histograms
- place multiple histograms on one plot
- Add lines to a plot
- Creating new data columns using *which()*

2 - Get data

Like last lesson, we are going to:

- get weather data from the file [LansingNOAA2016.csv](#)
- save it to a data frame called **weatherData**
- reformat the date to add the year (*fig 1*)
 - Note how R displays the dates in the default format (%Y-%m-%d), not the format used in the script (%m-%d-%Y)

```
1 source( file="scripts/reference.R" );
2 weatherData = read.csv( file="data/LansingNOAA2016.csv",
3 stringsAsFactors = FALSE );
4
5 ##### Part 1: Add year to date vector and save back to data frame
6 theDate = weatherData[ , "date"];          # save date column to vector
7 theDate = paste(theDate, "-2016", sep="");  # append -2016 to vector
8 theDate = as.Date(theDate, format="%m-%d-%Y"); # format vector as Date
9 weatherData$dateYr = theDate;              # save vector to Data Frame
10 ## equivalent to previous line for data frames -- but not tibbles
11 # weatherData[, "dateYr"] = theDate;
```

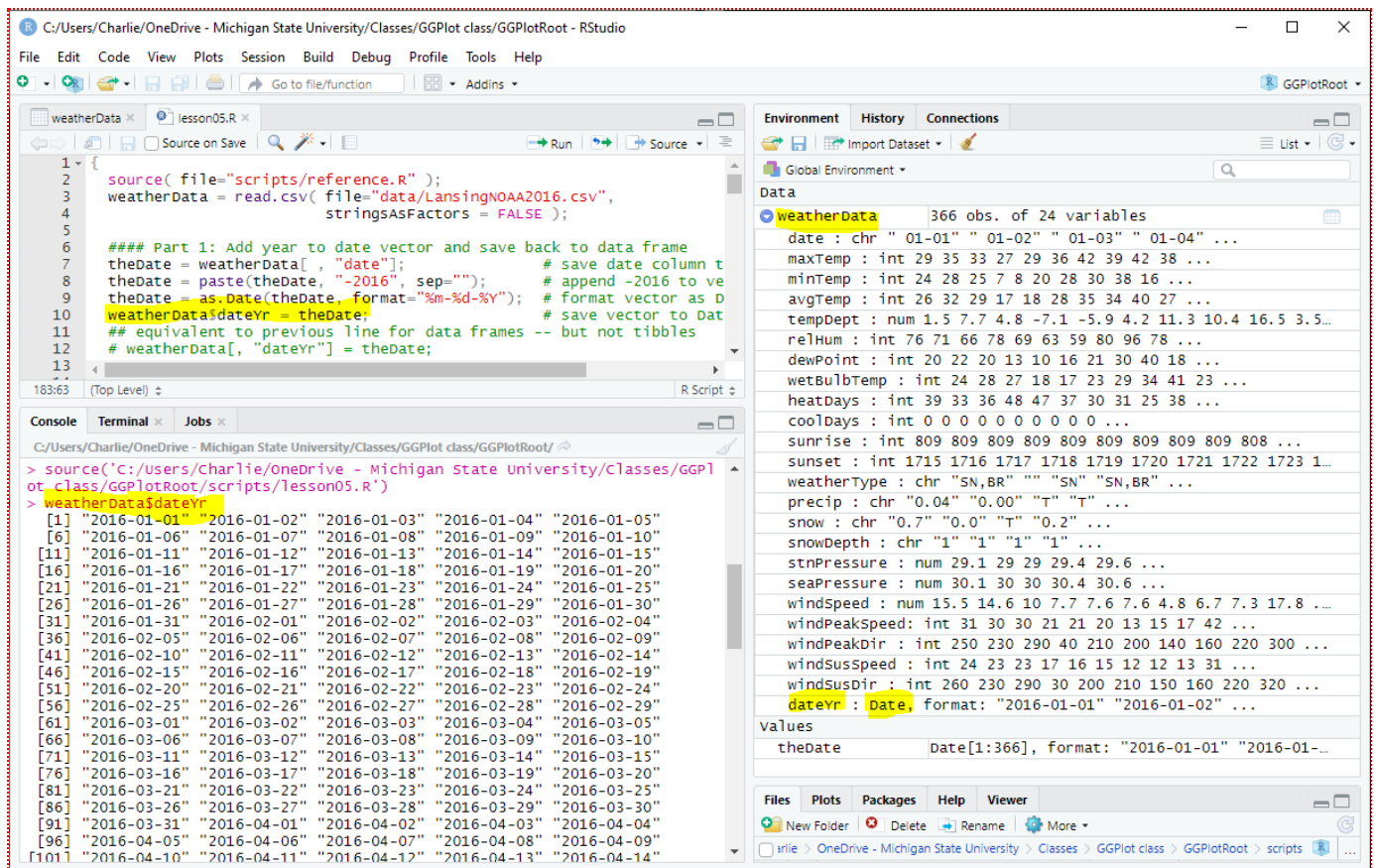


Fig 1: A couple of different ways to see the new **dateYr** column in **weatherData**

2.1 - Saving the new data frame

We do not need to repeat the **date** conversion each time we read the data from the CSV file. Instead, using **write.csv()**, we can save the **weatherData** data frame with the reformatted year to a new CSV file. In this case, we will save the data frame to the file **LansingNOAA2016-2.csv** in the **data** directory.

We are going to put this command at the end of the script because we will also be using the **season** column, created later in this lesson, in future lessons.

```
#### End of Code: Save the modified data frame to a new CSV file
write.csv(weatherData, file="data/LansingNOAA2016-2.csv");
```

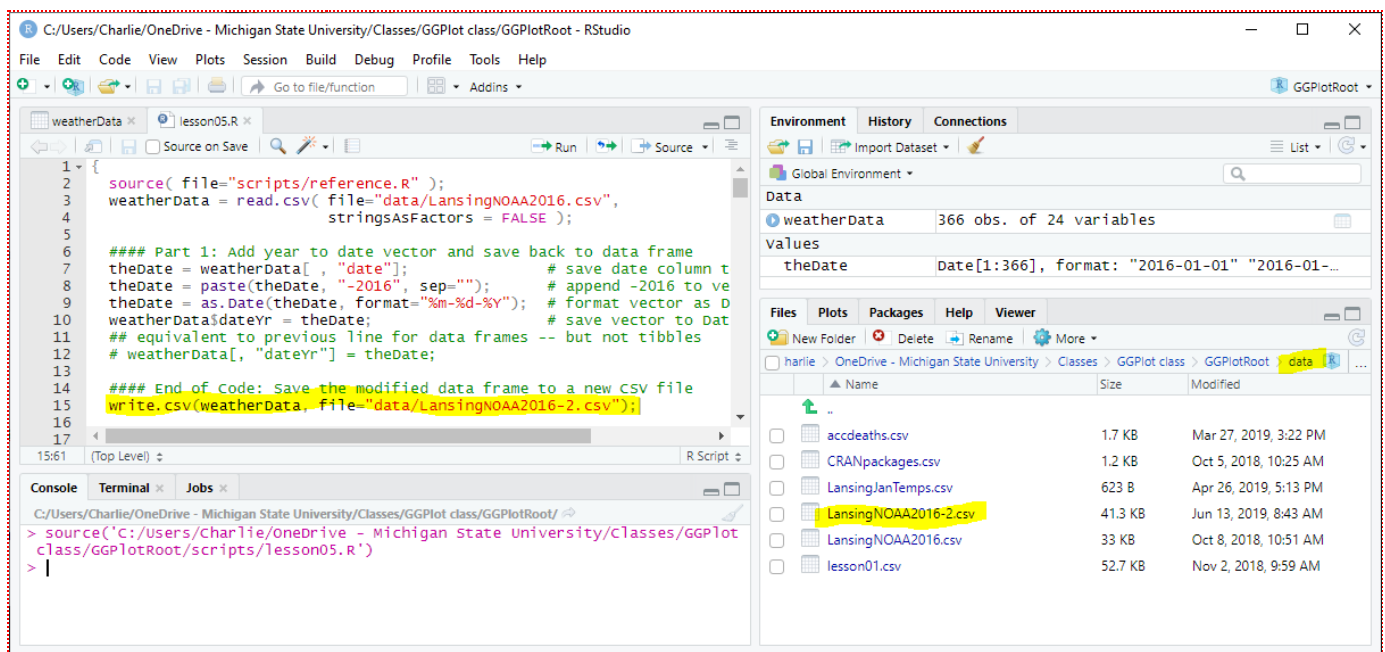


Fig 2: The new CSV file, **LansingNOAA2016-2.csv** was save to the **data** folder

2.2 - Using the new data frame

In the next lesson, we will read from the new CSV file by replacing line 3:

```

3 weatherData = read.csv(file="data/LansingNOAA2016.csv",
4 stringsAsFactors = FALSE);

```

with:

```

3 weatherData = read.csv(file="data/LansingNOAA2016-2.csv",
4 stringsAsFactors = FALSE);

```

3 - Creating a new data column: season

In this lesson, we will look at histograms that show seasonal temperatures. To do this we need to create a vector with 366 values that maps each day of the year (**dateYr** column) to a season. So, every value in this vector will be one of the four seasons.

3.1 - Create the season vector

We will start by create a vector that holds all the season values. We will use **vector()** to explicitly create a vector that contain character values (**mode="character"**) and has the same length as **theDate** vector. This create a vector with **366** values -- all of them "" (**fig 3**).

```

1 ##### Part 2: Set up the season and date variables
2 # create a season vector that has the same length as theDate vector
3 season = vector(mode="character", length=length(theDate));

```

To help us with the conditional statements used to find the season, we will create variables that hold the start date for each season. We need to be explicit that the variable is a date, using ***as.Date()***, and the ***format*** the date is in.

```
1 # create date variables for the beginning of each season
2 springStart = as.Date("03-21-2016", format="%m-%d-%Y");
3 summerStart = as.Date("06-21-2016", format="%m-%d-%Y");
4 fallStart = as.Date("09-21-2016", format="%m-%d-%Y");
5 winterStart = as.Date("12-21-2016", format="%m-%d-%Y");
```

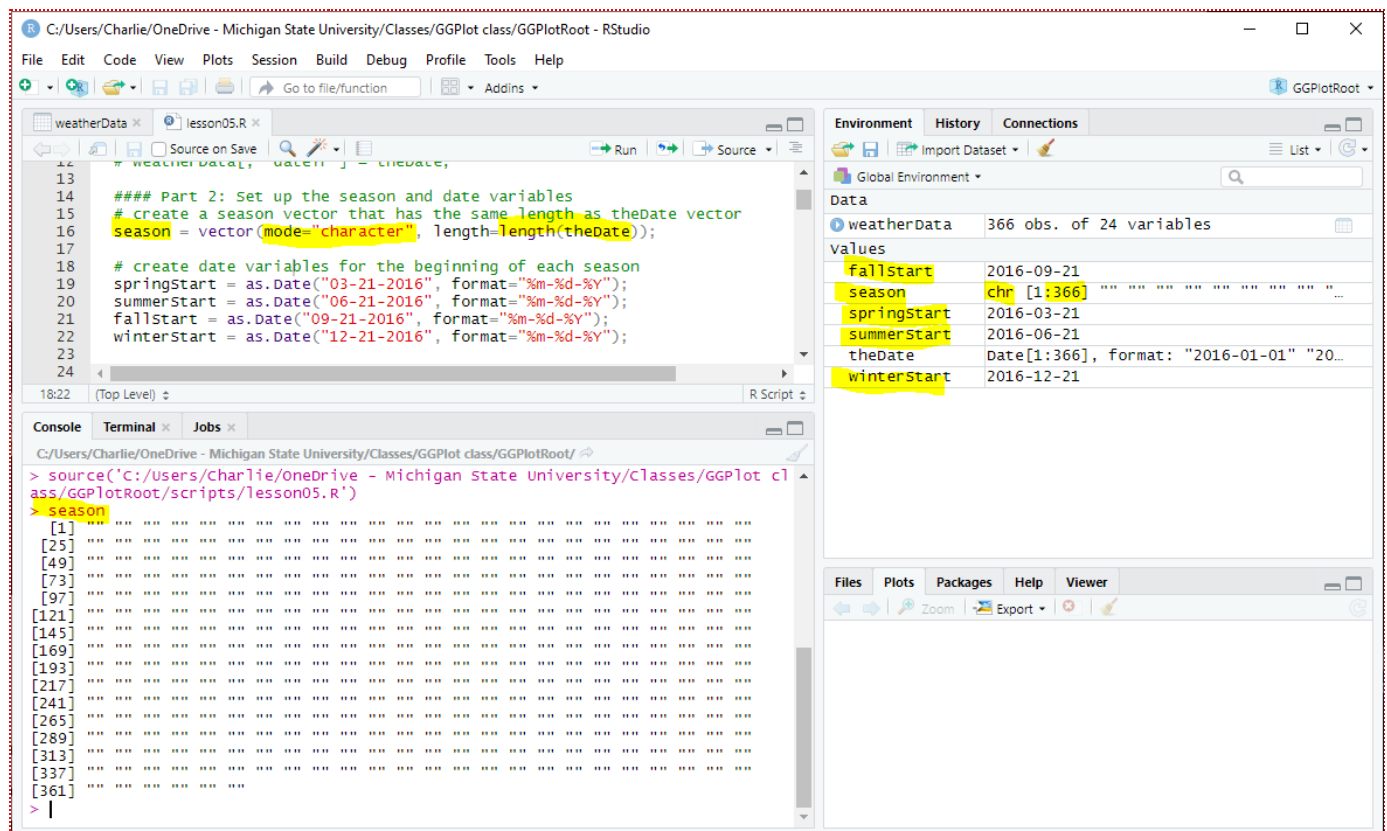


Fig 3: The initial `season` vector along with the variables that hold the season start dates

3.2 - Using logical operations to find the seasons

We need to go through each date value in the ***theDate*** and associate a season in ***season***.

To go through each value, we use a **for()** that goes from **1** to the length, 366, of **theDate**.

```
for(i in 1:length(theDate)) # go through each date
```

We can use logical operators on dates to see if a date is earlier (less than) or later (greater than) another date. Let's start with spring. Spring dates are later than or equal to 3/21 (***springStart***) **AND** earlier than 6/21 (***summerStart***)

```
# if the date falls with the spring season
if(theDate[i] >= springStart && theDate[i] < summerStart)
{
```

```
    season[i] = "Spring";
}
```

Summer and fall dates are found using a similar method.

Winter is a little different because the dates are not continuous, they fall at the beginning (Jan1-21) and end (Sept 21-Dec 31) of the year. So we need to use the **OR** operator instead. Winter dates are later than or equal to 12/21 (**winterStart**) **OR** earlier than 3/21 (**springStart**)

```
else if(theDate[i] >= winterStart || theDate[i] < springStart)
{
    season[i] = "Winter";
}
```

3.3 - Populate the season vector

Putting all the code together and adding in an error condition. *Note: adding error conditions is a good programming practice -- it would catch situations where a value was spelled wrong or missing.*

```
1 ##### Part 3: Create a season vector based on theDates vector
2 for(i in 1:length(theDate)) # go through each date
3 {
4     # if the date falls with the spring season
5     if(theDate[i] >= springStart && theDate[i] < summerStart)
6     {
7         season[i] = "Spring";
8     }
9     # if the date falls with the summer season
10    else if(theDate[i] >= summerStart && theDate[i] < fallStart)
11    {
12        season[i] = "Summer";
13    }
14    # if the date falls with the fall season
15    else if(theDate[i] >= fallStart && theDate[i] < winterStart)
16    {
17        season[i] = "Fall";
18    }
19    # if the date falls with the winter season --
20    #     using || because dates are not continuous)
21    else if(theDate[i] >= winterStart || theDate[i] < springStart)
22    {
23        season[i] = "Winter";
24    }
25    else # something went wrong... always good to check
26    {
```

```

27   season[i] = "Error";
28 }
29 }

```

The screenshot shows the RStudio interface. The script editor contains the following code:

```

22 winterStart = as.Date("12-21-2016", format="%m-%d-%Y");
23
24 #### Part 3: Create a season vector based on theDates vector
25 for(i in 1:length(theDate)) # go through each date
26 {
27   # if the date falls with the spring season
28   if(theDate[i] >= springStart && theDate[i] < summerStart)
29   {
30     season[i] = "Spring";
31   }
32   # if the date falls with the summer season
33   else if(theDate[i] >= summerStart && theDate[i] < fallStart)
34   {
35     season[i] = "Summer";
36   }
37   # if the date falls with the fall season
38   else if(theDate[i] >= fallStart && theDate[i] < winterStart)
39   {
40     season[i] = "Fall";
41   }
42   # if the date falls with the winter season --
43   # using || because dates are not continuous
44   else if(theDate[i] >= winterStart || theDate[i] < springStart)
45   {
46     season[i] = "Winter";
47   }
48 }

```

The console shows the output of the `season` vector:

```

> season
[1] "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter"
[13] "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter"
[25] "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter"
[37] "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter"
[49] "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter"
[61] "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter"
[73] "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter" "winter"
[85] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[97] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[109] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[121] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[133] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[145] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[157] "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring" "spring"
[169] "spring" "spring" "spring" "spring" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer"
[181] "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer"
[193] "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer"

```

Fig 4: Populating the season vector and displaying the values

We could also use **which()** to populate the season values, for more information go to [Extension: Using which\(\) to find index values](#)

3.4 - Adding season to the data frame

We are going to be using the season vector in this lesson and in future lessons, so let's add it to the **weatherData** data frame.

```

1 # Part 4: create a new column in weatherData called season
2 weatherData$season = season;
3 ## equivalent to above line for data frames, not tibbles
4 # weatherData[, "season"];

```

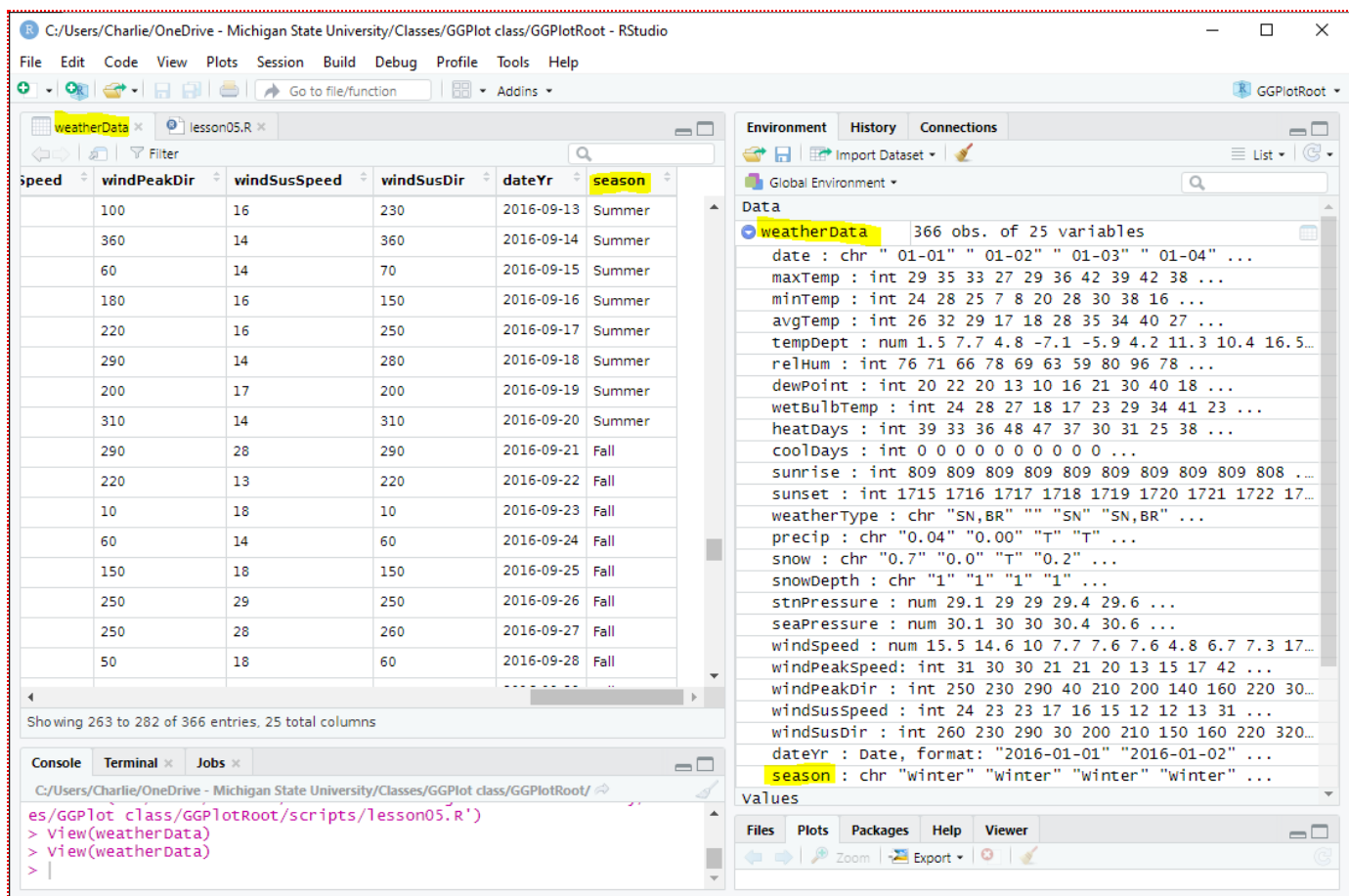


Fig 5: The season vector in the data frame

4 - Creating a histogram

We are going to plot a histogram of average temperatures by **season** but first we will plot a histogram of the whole year using **geom_histogram()**.

For the **mapping** subcomponent we use:

- **x=avgTemp**
- **y=..count..**

The 2 dots before and after **count** indicate that **count is an operation, not a variable**. In other words, it says that the **y-axis will represent a count of the x-axis values**. In **geom_histogram()**, **y=..count..** is the default. The other possible operations are: **..density..**, **..ncount..**, and **..ndensity..**, the latter two are scaled to have a maximum of 1.

```
1 ##### Part 5: Create a histogram of temperatures for the year
2 plotData = ggplot( data=weatherData ) +
3     geom_histogram( mapping=aes(x=avgTemp, y=..count..) );
4 plot(plotData);
```

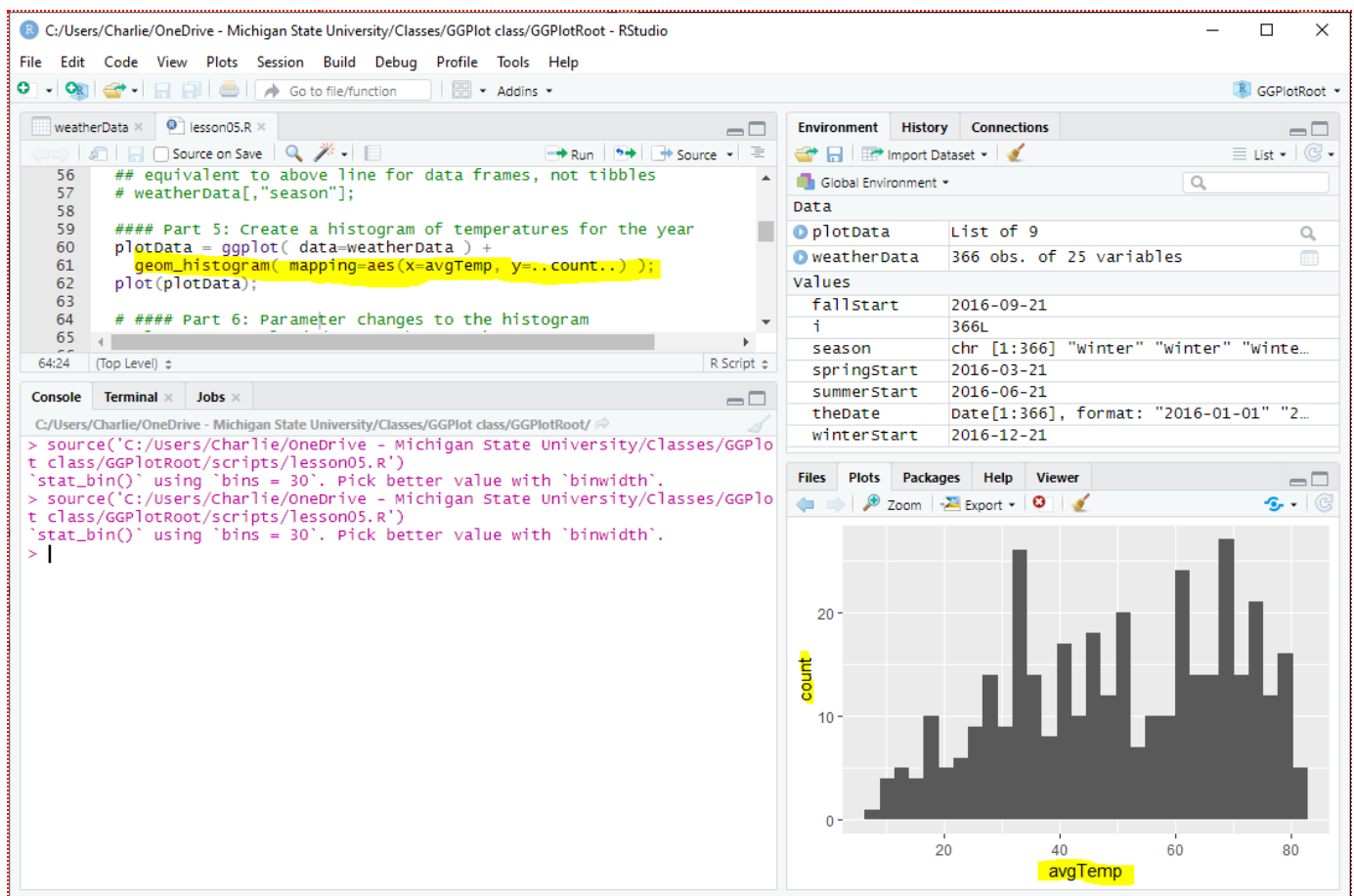



Fig 6: A histogram of *avgTemp*

4.1 - Changing bins and coloring

Next we will modify subcomponents in *geom_histogram()* to changes colors and set the number of bins.

The subcomponents we will set are:

- **bins**: the number of bins (default is 30)
- **fill**: the background color of the bins
- **color**: outline color of the bins

```

1 ##### Part 6: Parameter changes to the histogram
2 plotData = ggplot( data=weatherData ) +
3   geom_histogram( mapping=aes(x=avgTemp, y=..count..),
4     bins=40,
5     color="grey20",
6     fill="darkblue");
7
8 plot(plotData);

```

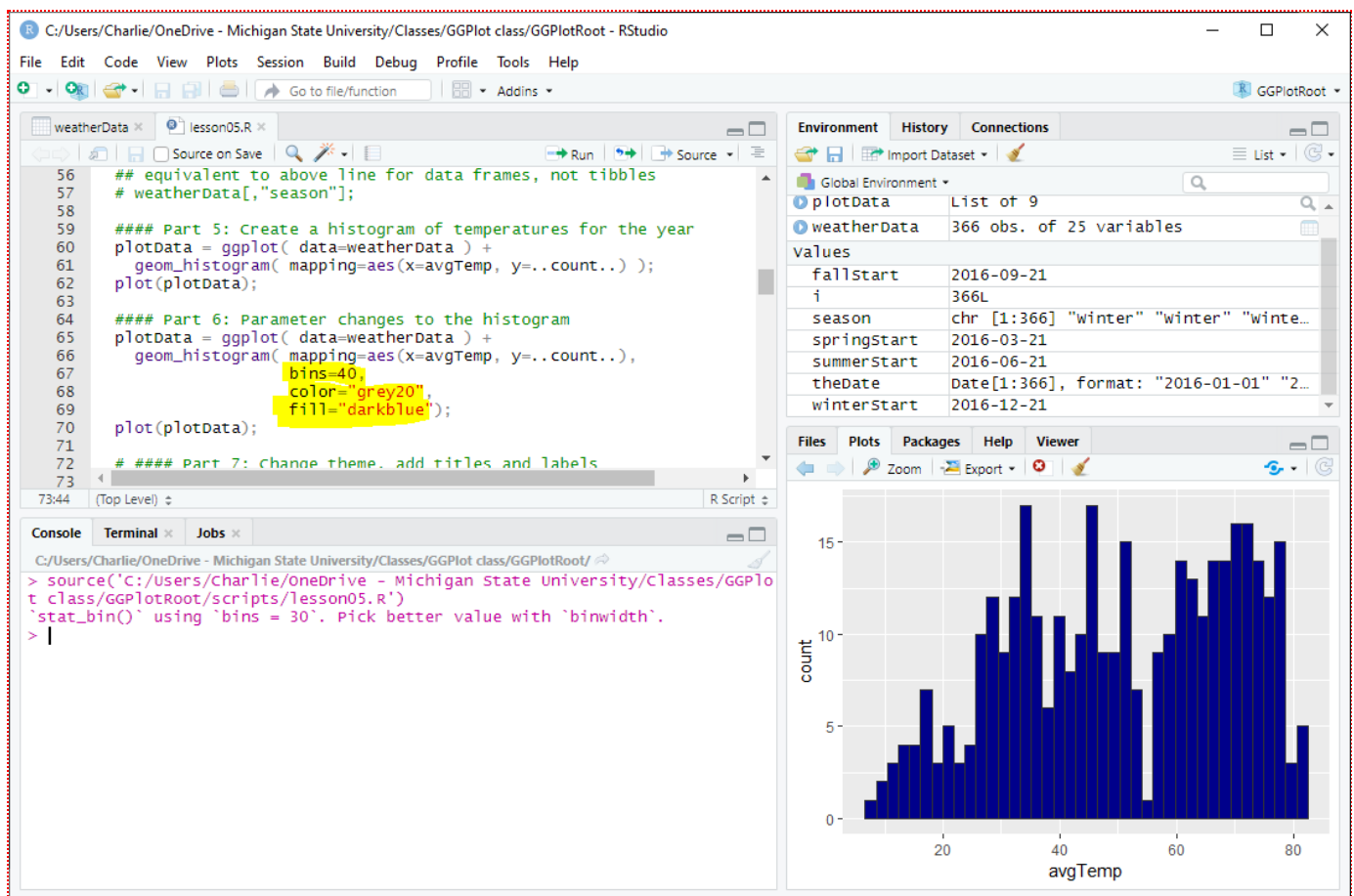



Fig 7: Changing the number of bin and the color of the bins

4.2 - Add labels and changing the theme

Next we will change the plot theme using the **theme_classic()** component and add titles and axes labels using the **labs()** component.

```

1 #### Part 7: Change theme, add titles and labels
2 plotData = ggplot( data=weatherData ) +
3   geom_histogram( mapping=aes(x=avgTemp, y=..count..),
4     bins=40,
5     color="grey20",
6     fill="darkblue") +
7   theme_classic() +
8   labs(title = "Temperature Histogram",
9     subtitle = "Lansing, Michigan: 2016",
10    x = "Average Temp (Fahrenheit)",
11    y = "Count");
12 plot(plotData);

```

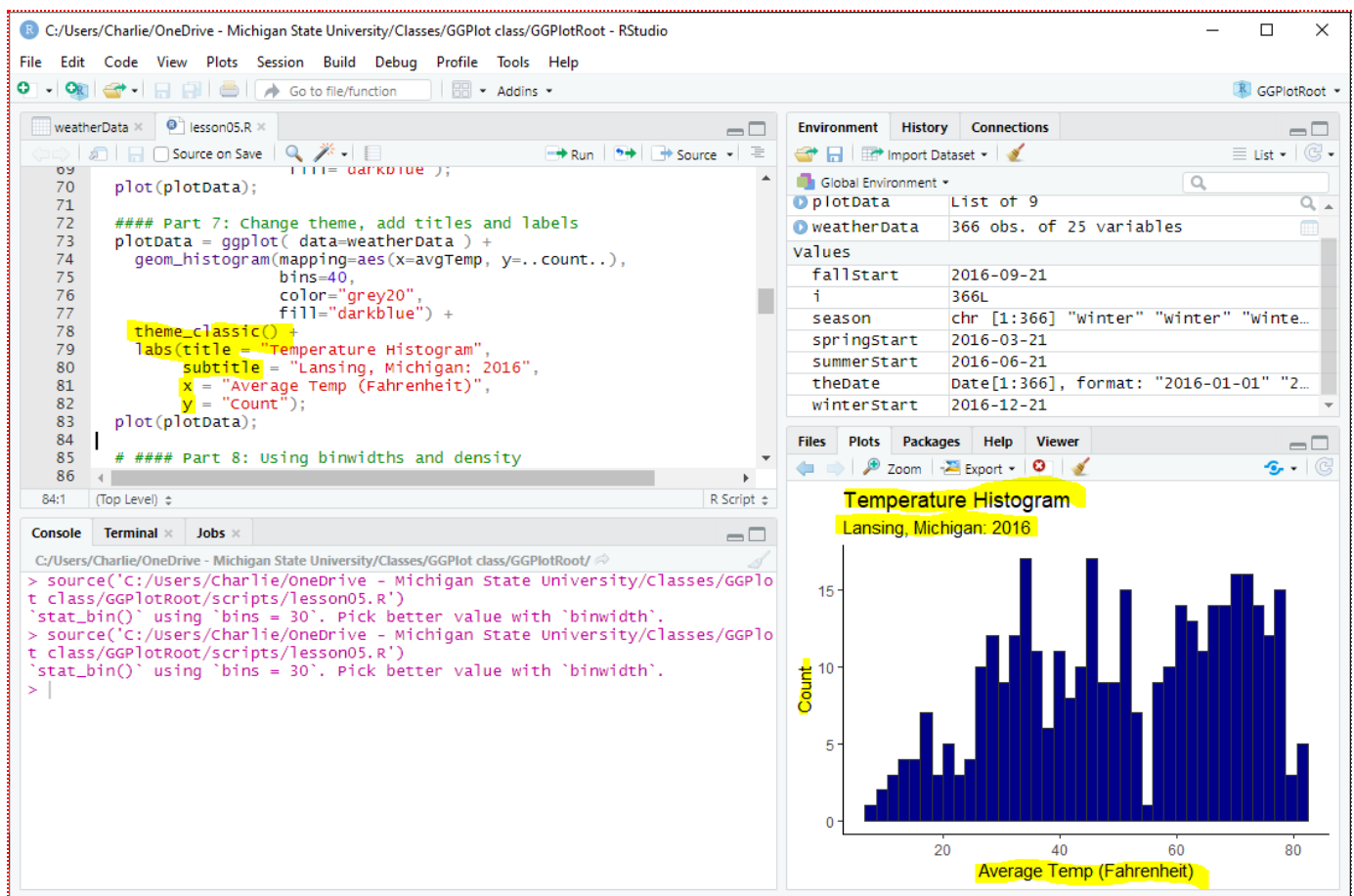


Fig 8: Titles, labels, and a new theme added to the plot

4.3 - Histogram alternative parameters

Here are a couple other values to modify a histogram:

- **binwidth** (instead of **bins**) -- **binwidth** defines the size of the bins (in this case, 4 degrees) whereas **bins** defines the number of bins
- **..density..** (instead of **..count..**) on the y-axis -- **..density..** integrates the bin areas to 1

```

1 ##### Part 8: Using binwidths and density
2 plotData = ggplot( data=weatherData ) +
3   geom_histogram(mapping=aes(x=avgTemp, y=..density..),
4                 binwidth=4,
5                 color="grey20",
6                 fill="darkblue") +
7   theme_classic() +
8   labs(title = "Temperature Histogram",
9        subtitle = "Lansing, Michigan: 2016",
10       x = "Average Temp (Fahrenheit)",
11       y = "Density");
12 plot(plotData);

```

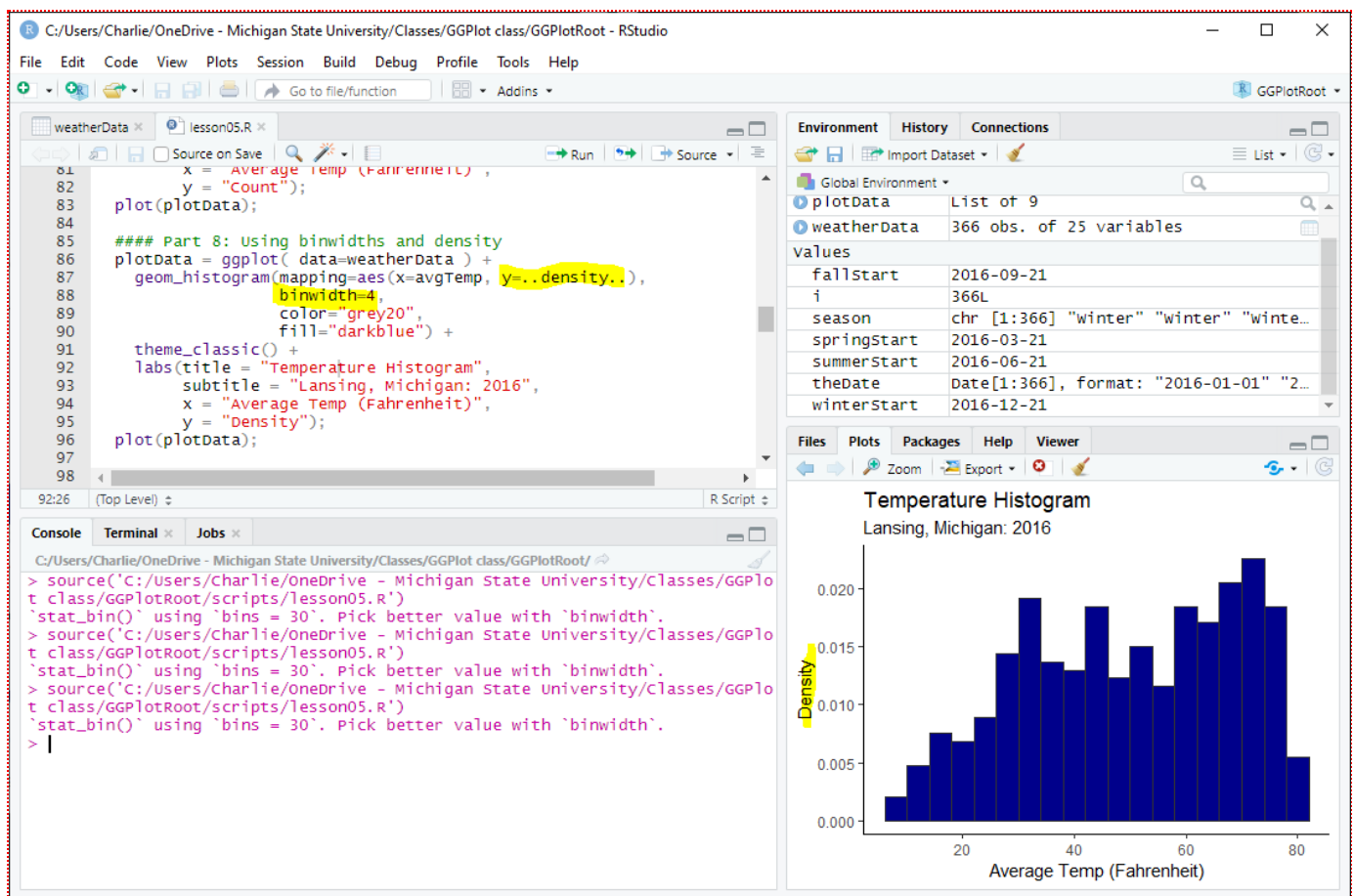


Fig 9: Density values and binwidths in a histogram

5 - Adding lines

Next, we will add two vertical lines to the plot representing the mean and median temperatures. The plotting component that adds vertical lines is **geom_vline()**.

The mapping for **geom_vline()** requires only one value to create a vertical line: the x-value of the line. In GGPlot, this parameter is called **xintercept**.

We set **xintercept** to the functions **mean()** and **median()** to plot the mean and median values of the values in **avgTemp**.

```
1 ##### Part 9: Add vertical lines representing mean and median
2 plotData = ggplot( data=weatherData ) +
3   geom_histogram(mapping=aes(x=avgTemp, y=..density..),
4     binwidth=4,
5     color="grey20",
6     fill="darkblue") +
7   geom_vline(mapping=aes(xintercept=mean(avgTemp)),
8     color="red",
9     size=1.2) +
10  geom_vline(mapping=aes(xintercept=median(avgTemp)),
11    color="green",
```

```

12         size=1.2) +
13     theme_classic() +
14     labs(title = "Temperature Histogram",
15          subtitle = "Lansing, Michigan: 2016",
16          x = "Average Temp (Fahrenheit)",
17          y = "Density");
18 plot(plotData);

```

Note: I have also set the color and size of the two vertical lines.

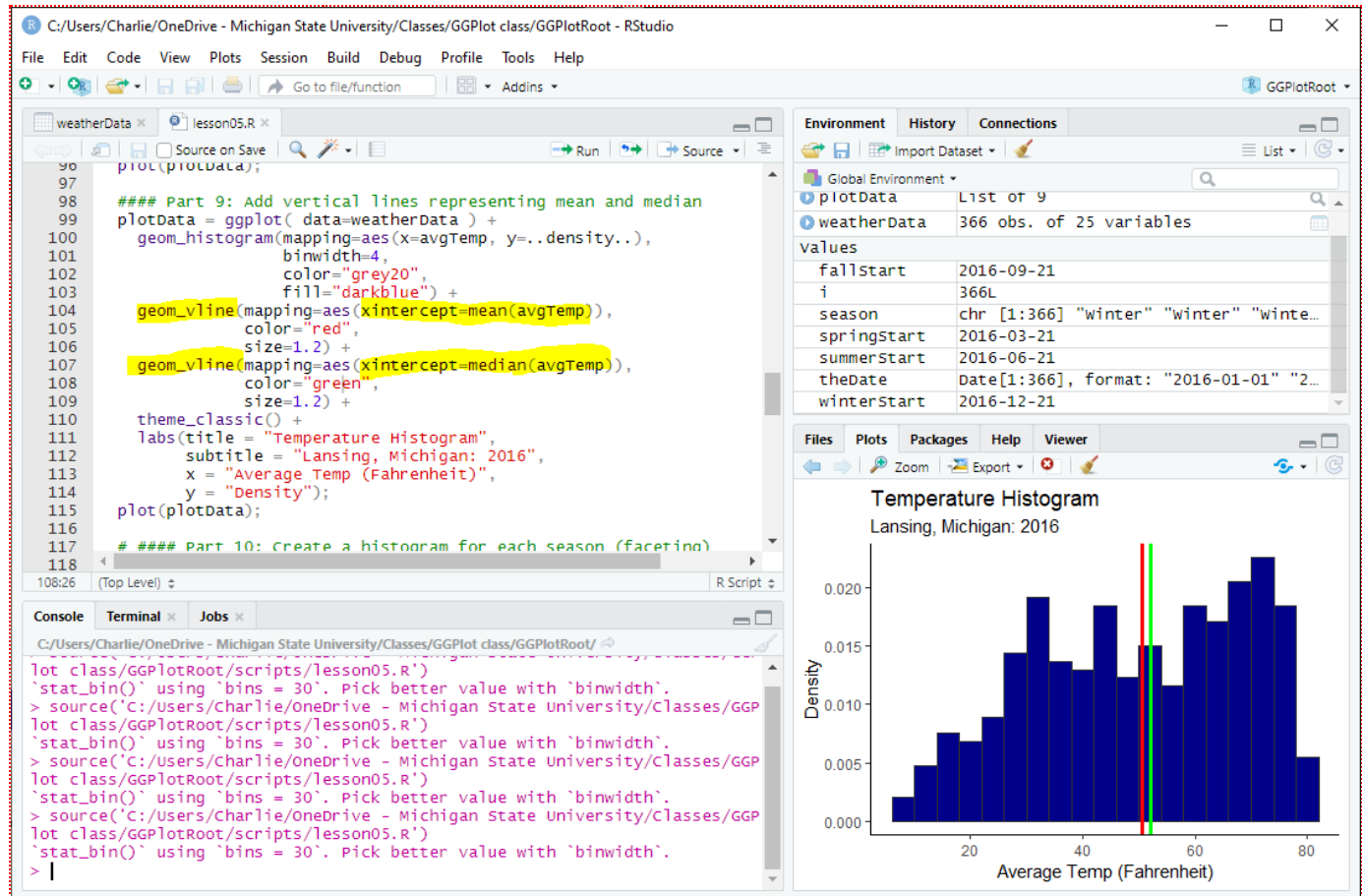


Fig 10: Vertical lines added to histogram

6 - Throwing in seasons

Now we are going to create histograms that show temperature by season. There are two ways to do this:

1. Create a histogram for each season (faceting)
2. Create a stacked histogram

6.1 - Faceting -- one plot for each season

We can instruct GGPlot to create a histogram for each season by using the component **facet_grid()**.

```

facet_grid( facet= season ~ .)

```

The subcomponent **facet** gives the grouping instructions in the form of (**y-axis variable ~ x-axis variable**). In this example we are using the variable **season** to group the data and we are putting **season** on the y-axis. The (.) in place of the x-axis variable means that there is no grouping along the x-axis.

```

1 ##### Part 10: Create a histogram for each season (faceting)
2 plotData = ggplot(data=weatherData) +
3     geom_histogram(mapping=aes(x=avgTemp, y=..count..),
4                     bins=40,
5                     color="grey20",
6                     fill="darkblue") +
7     theme_classic() +
8     facet_grid( facet= season ~ .) +
9     labs(title = "Temperature Histogram",
10          subtitle = "Lansing, Michigan: 2016",
11          x = "Average Temp (Fahrenheit)",
12          y = "Count");
13 plot(plotData);

```

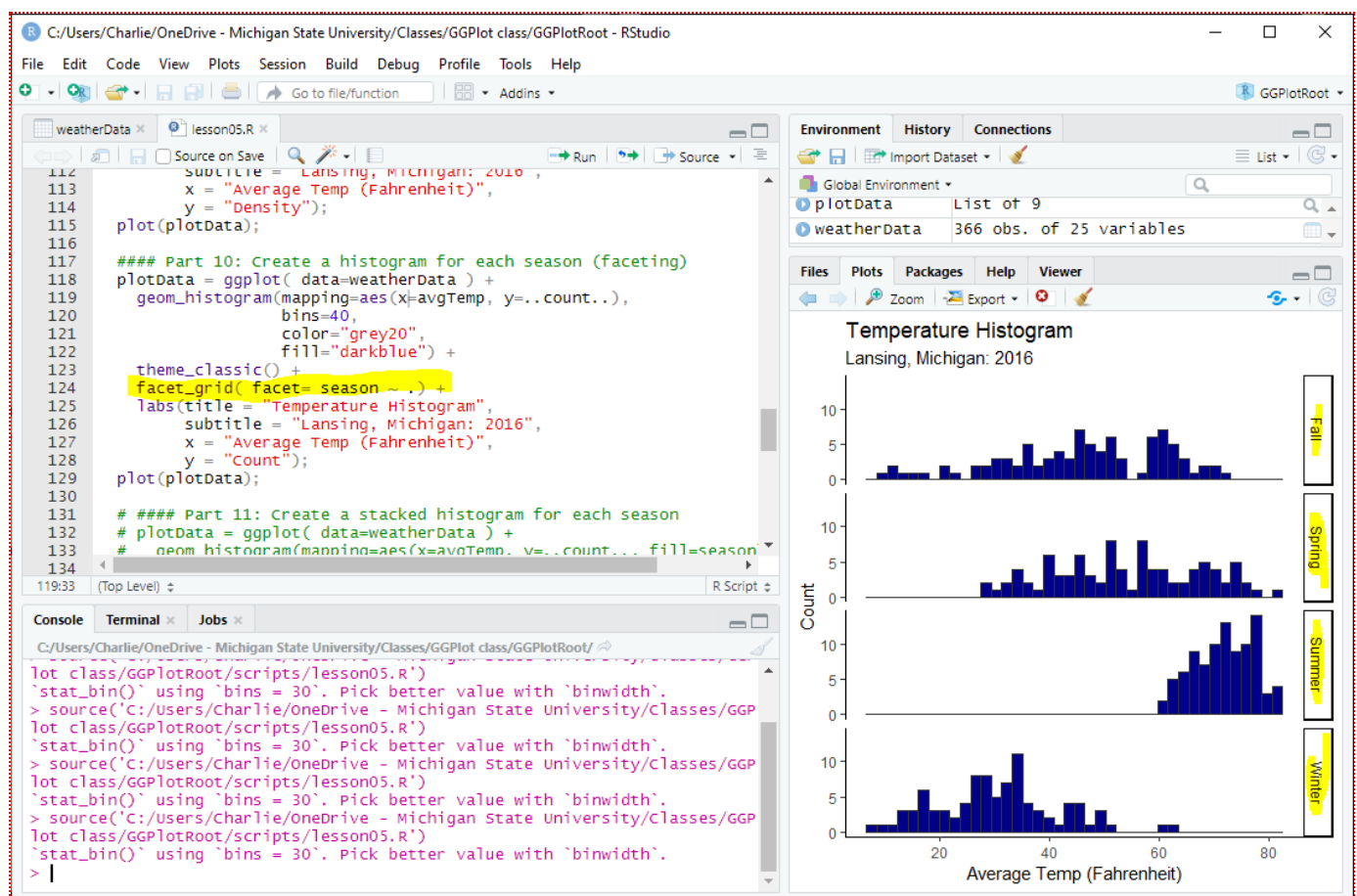


Fig 11: Displaying a histogram for each season

If you want to group by **season along the x-axis** you would use:

```
facet_grid( facet= . ~ season)
```

6.2 - Stacked the seasons on a histogram

Another option is to create a stacked histogram that shows the count for each **season** on one histogram.

To do this we make two changes to make to the **geom_histogram()** component:

1. Set **position** to **stack**
2. Add a **fill** parameter to **mapping** and set it to **season**.

```
geom_histogram(mapping=aes(x=avgTemp, y=..count.., fill = season),
               bins=40,
               position="stack",
               color="grey20")
```

The second change, **fill**, needs to be unpacked. **fill = season** means that we are filling the histogram by the categories in the **season** variable (i.e., spring, summer, fall, winter). GGPlot will visually separate the seasons by adding color for each season.

Note: fill goes in mapping because fill is mapping the fill data (seasons) to the plot (bars)

```
1 ##### Part 11: Create a stacked histogram for each season
2 plotData = ggplot(data=weatherData) +
3     geom_histogram(mapping=aes(x=avgTemp, y=..count.., fill=season),
4                   bins=40,
5                   color="grey20",
6                   position="stack") +
7     theme_classic() +
8     labs(title = "Temperature Histogram",
9          subtitle = "Lansing, Michigan: 2016",
10         x = "Average Temp (Fahrenheit)",
11         y = "Count");
12 plot(plotData);
```

In future lessons, we will go over how to manipulate the legend. For quick changes to the legend go to

Extension: Quick Legend Changes.

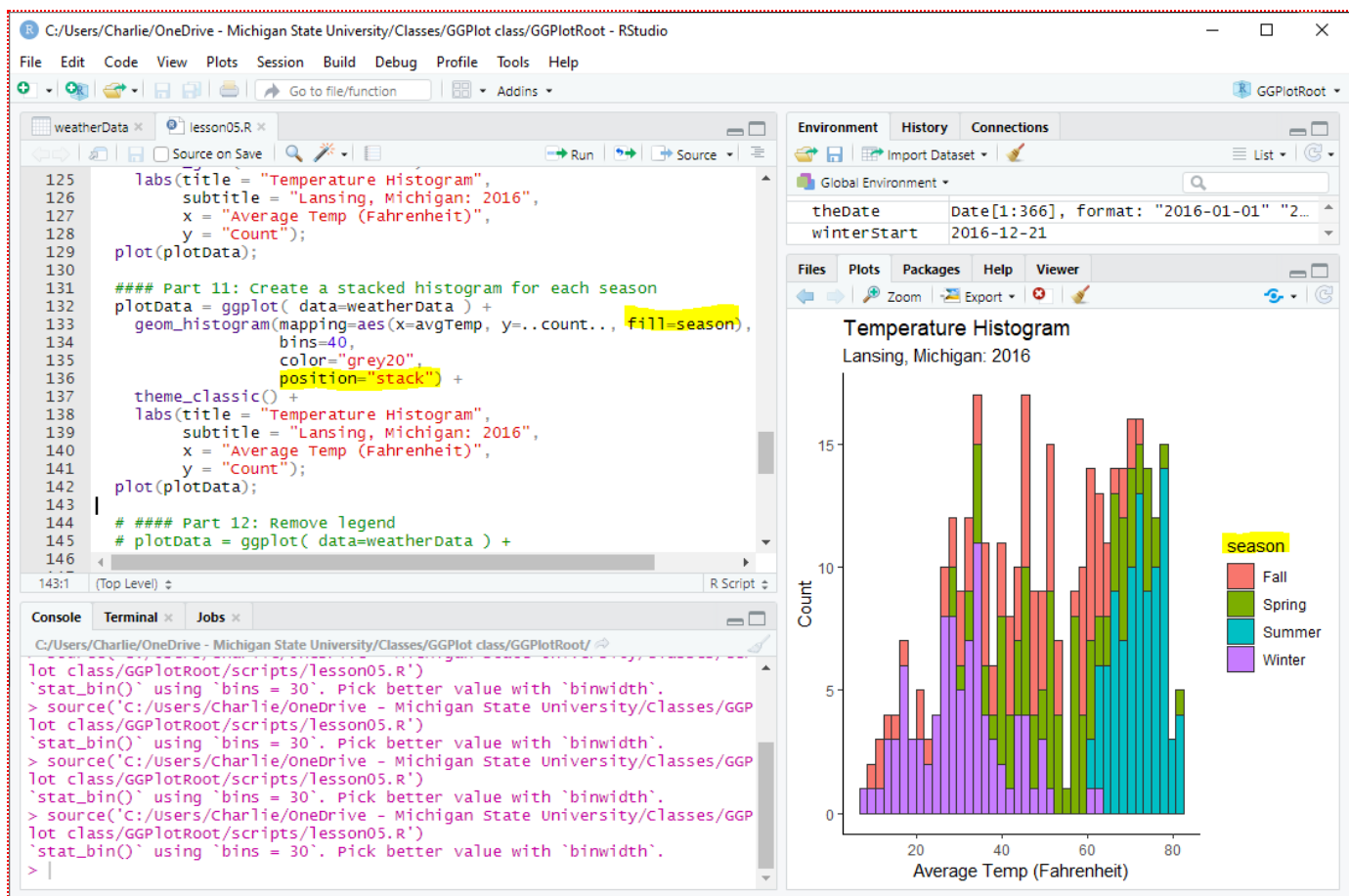


Fig 12: A histogram with stacked seasonal count

7 - Application

1. Create a script file in your **scripts** folder called **app05.r**.
2. Open the data frame created in this lesson: **LansingNOAA2016-2.csv**
3. Do a histogram of average humidity -- add titles and appropriate labels.
4. Create a column in your data frame called **biMonth** that divides the year into 6 categories: JanFeb, MarApr, MayJun, JulAug, SepOct, and NovDec
5. Using faceting, **facet_grid()**, create a separate histogram for each **biMonth** category
6. Create a stacked histogram, using **fill** subcomponent, of average humidity using **biMonth**
 - Add two vertical lines that give the mean average humidity for **JanFeb** and **JulAug**.
 - Hint: you want to find the mean of the values in the **relHum** column and **JanFeb** (**JulAug**) rows
 - Challenge: The legend is currently wasting a lot of white-space on the right side of the plot. Move the legend over the plot in a way that it does not cover the plot data. (in **extension** and lesson 7)
 - Challenge: The months are in alphabetical order instead of ordered by date. Change the order of the months by date -- this is covered in detail in the next lesson.

8 - Extension: using which() to find index values

We can use **which()** to get the index values for the dates.

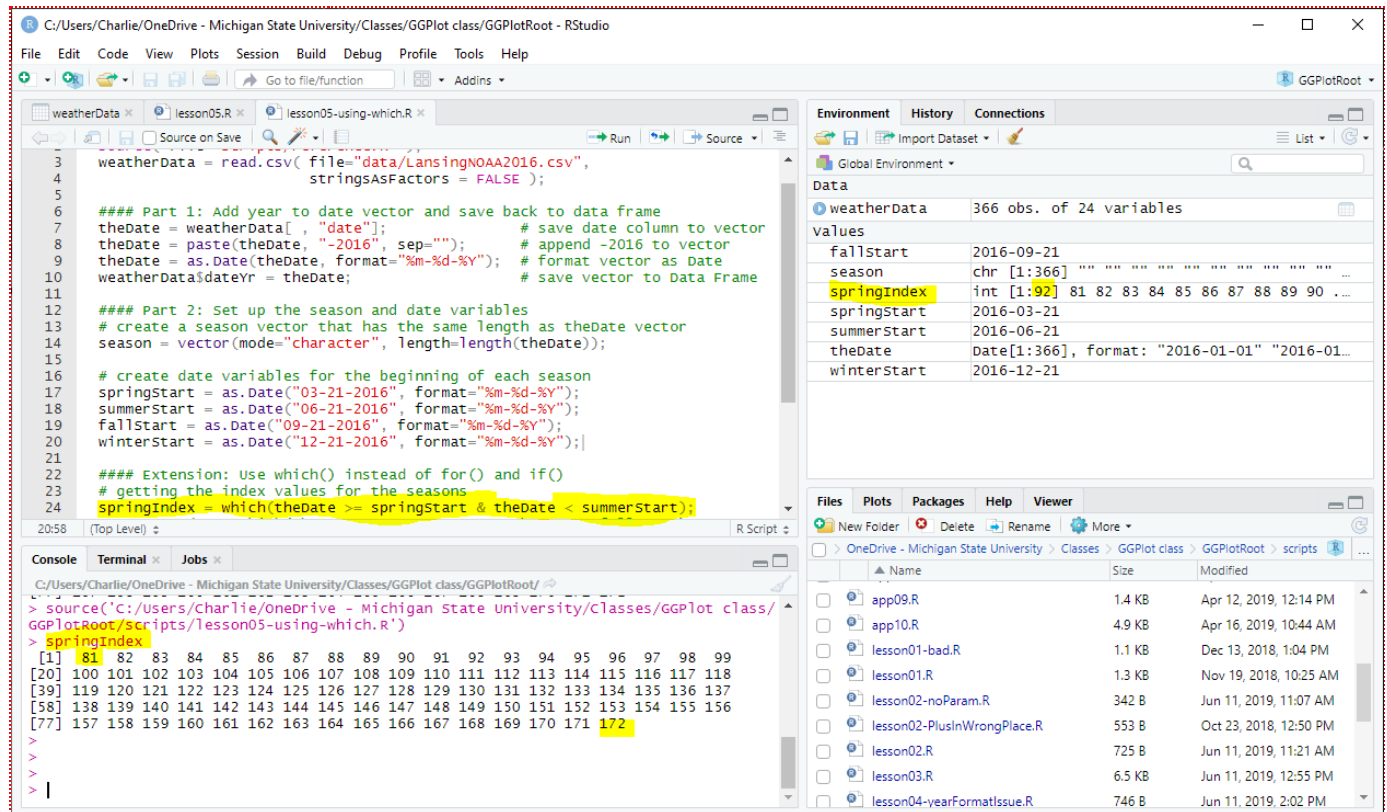
A **which()** gives the index of the values in the vector that meet a condition.

So, to find the Spring dates, we ask which dates are later than **springStart** AND earlier than **summerStart**:

```
1 | springIndex = which(theDate >= springStart & theDate < summerStart);
```


Note: the logical condition is a single & instead of a double &&

The vector, **springIndex**, is populated by the 92 index values representing the dates between March 21 (81st day of year) and June 20 (172nd day of year).



```
1 weatherData = read.csv( file="data/LansingNOAA2016.csv",
2   stringsAsFactors = FALSE );
3
4 ##### Part 1: Add year to date vector and save back to data frame
5 theDate = weatherData[, "date"]; # save date column to vector
6 theDate = paste(theDate, "-2016", sep=""); # append -2016 to vector
7 theDate = as.Date(theDate, format="%m-%d-%Y"); # format vector as Date
8 weatherData$dateYr = theDate; # save vector to Data Frame
9
10 ##### Part 2: Set up the season and date variables
11 # create a season vector that has the same length as theDate vector
12 season = vector(mode="character", length=length(theDate));
13
14 # create date variables for the beginning of each season
15 springStart = as.Date("03-21-2016", format="%m-%d-%Y");
16 summerStart = as.Date("06-21-2016", format="%m-%d-%Y");
17 fallStart = as.Date("09-21-2016", format="%m-%d-%Y");
18 winterStart = as.Date("12-21-2016", format="%m-%d-%Y");
19
20 ##### Extension: Use which() instead of for() and if()
21 # getting the index values for the seasons
22 springIndex = which(theDate >= springStart & theDate < summerStart);
```

Environment History Connections

Global Environment

Data

weatherData	366 obs. of 24 variables
fallStart	2016-09-21
season	chr [1:366] "" "" "" "" "" "" "" "" "" ...
springIndex	int [1:92] 81 82 83 84 85 86 87 88 89 90 ...
springStart	2016-03-21
summerStart	2016-06-21
theDate	Date[1:366], format: "2016-01-01" "2016-01..."
winterStart	2016-12-21

Files Plots Packages Help Viewer

New Folder Delete Rename More

OneDrive - Michigan State University > Classes > GGPlot class > GGPlotRoot > scripts

Name	Size	Modified
app09.R	1.4 KB	Apr 12, 2019, 12:14 PM
app10.R	4.9 KB	Apr 16, 2019, 10:44 AM
lesson01-bad.R	1.1 KB	Dec 13, 2018, 1:04 PM
lesson01.R	1.3 KB	Nov 19, 2018, 10:25 AM
lesson02-noParam.R	342 B	Jun 11, 2019, 11:07 AM
lesson02-PlusInWrongPlace.R	553 B	Oct 23, 2018, 12:50 PM
lesson02.R	725 B	Jun 11, 2019, 11:21 AM
lesson03.R	6.5 KB	Jun 11, 2019, 12:55 PM
lesson04-yearFormatIssue.R	746 B	Jun 11, 2019, 2:02 PM

Console Terminal Jobs

C:/Users/Charlie/OneDrive - Michigan State University/Classes/GGPlot class/GGPlotRoot/

```
> source("C:/Users/Charlie/OneDrive - Michigan State University/Classes/GGPlot class/
GGPlotRoot/scripts/lesson05-using-which.R")
> springIndex
[1] 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
[20] 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118
[39] 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137
[58] 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156
[77] 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172
```

Fig 13: **springVector** has 92 values: the 92 numbers from 81-172

The procedure is similar for summer and fall:

```
summerIndex = which(theDate >= summerStart & theDate < fallStart);
fallIndex = which(theDate >= fallStart & theDate < winterStart);
```

And, for winter, we need to switch the logical condition to **OR** because the winter date are not continuous:

```
winterIndex = which(theDate >= winterStart | theDate < springStart);
```

Unlike the three other season vectors, **winterIndex** has discontinuous values (1-80 then 356-366) representing the fact that winter days happen at both the beginning and end of the calendar year.

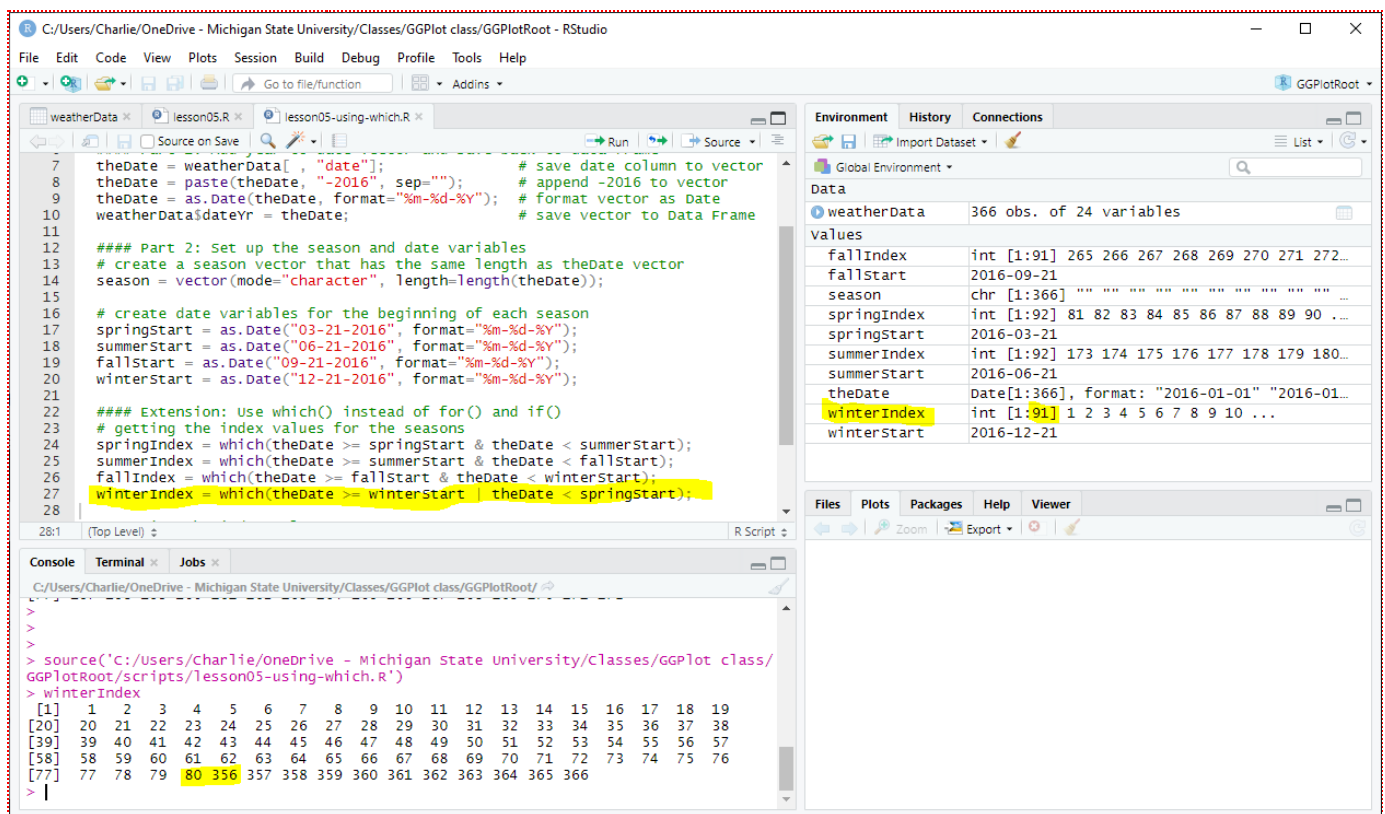


Fig 14: `winterIndex` has 91 values: 1-81 and 357-366

8.1 - Save the season to a data frame column

We have the index values for the days of each season --we can use the index value to set the season values in the **season** vector.

```
1 # Using the index values to set the season values
2 season[springIndex] = "Spring"; # sets values 81-172 to spring
3 season[summerIndex] = "Summer"; # sets values 173-264 to summer
4 season[fallIndex] = "Fall"; # sets values 265-356 to fall
5 season[winterIndex] = "Winter"; # sets values 1-80, 357-366 to winter
```

We can then save the vector to the **weatherData** data frame:

```
# Save the season vector to weatherData
weatherData$season = season;
```

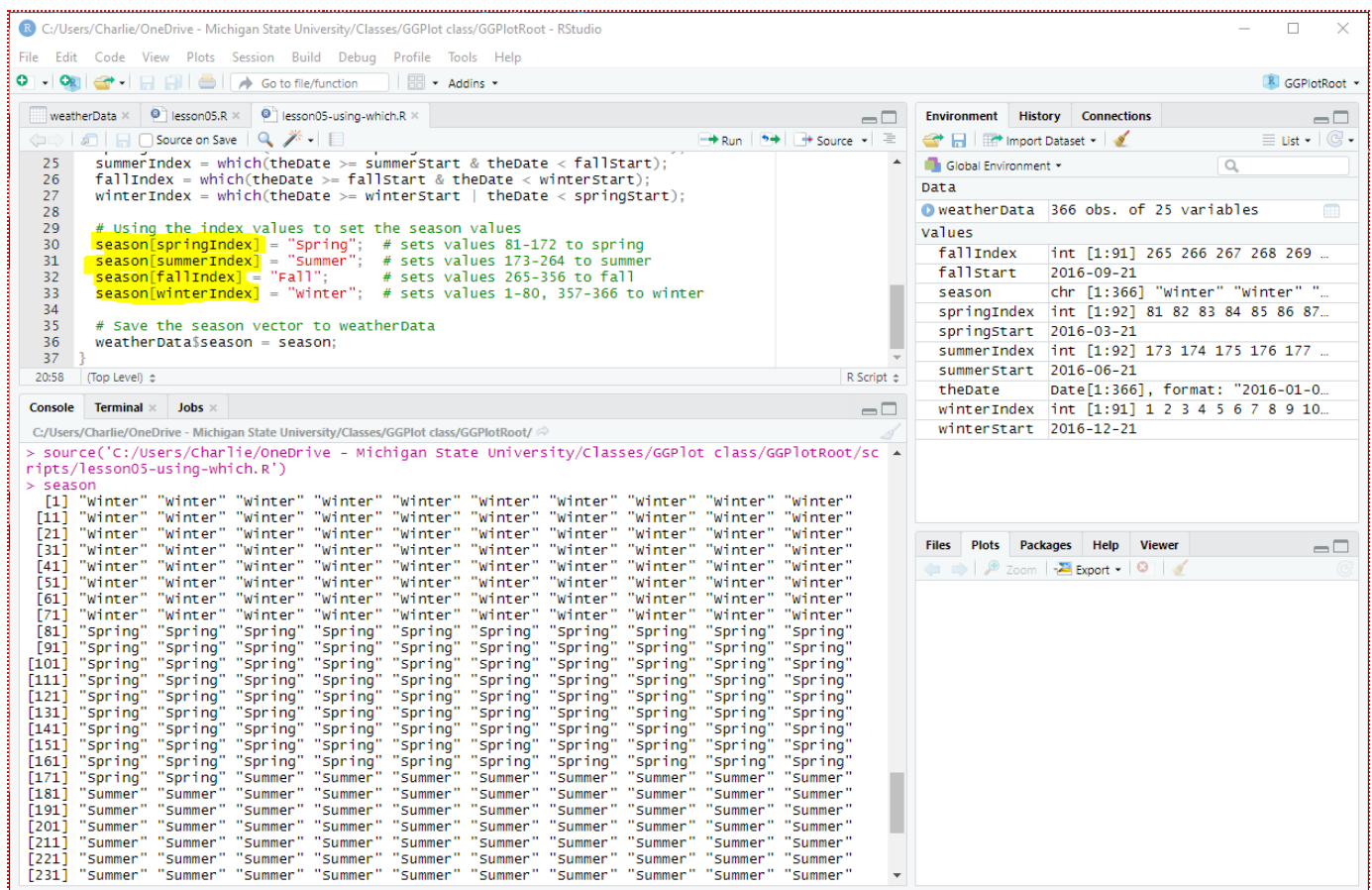


Fig 15: The new column in **weatherData** called **season** (note: the number of variables went from 23 to 24)

9 - Extension: Quick Legend changes

GGPlot automatically adds a legend when the **fill** parameter is used in **mapping**. If you do not want a legend in your plot you can modify the parameter **legend.position** in the **theme()** component:

```
theme(legend.position="none")
```

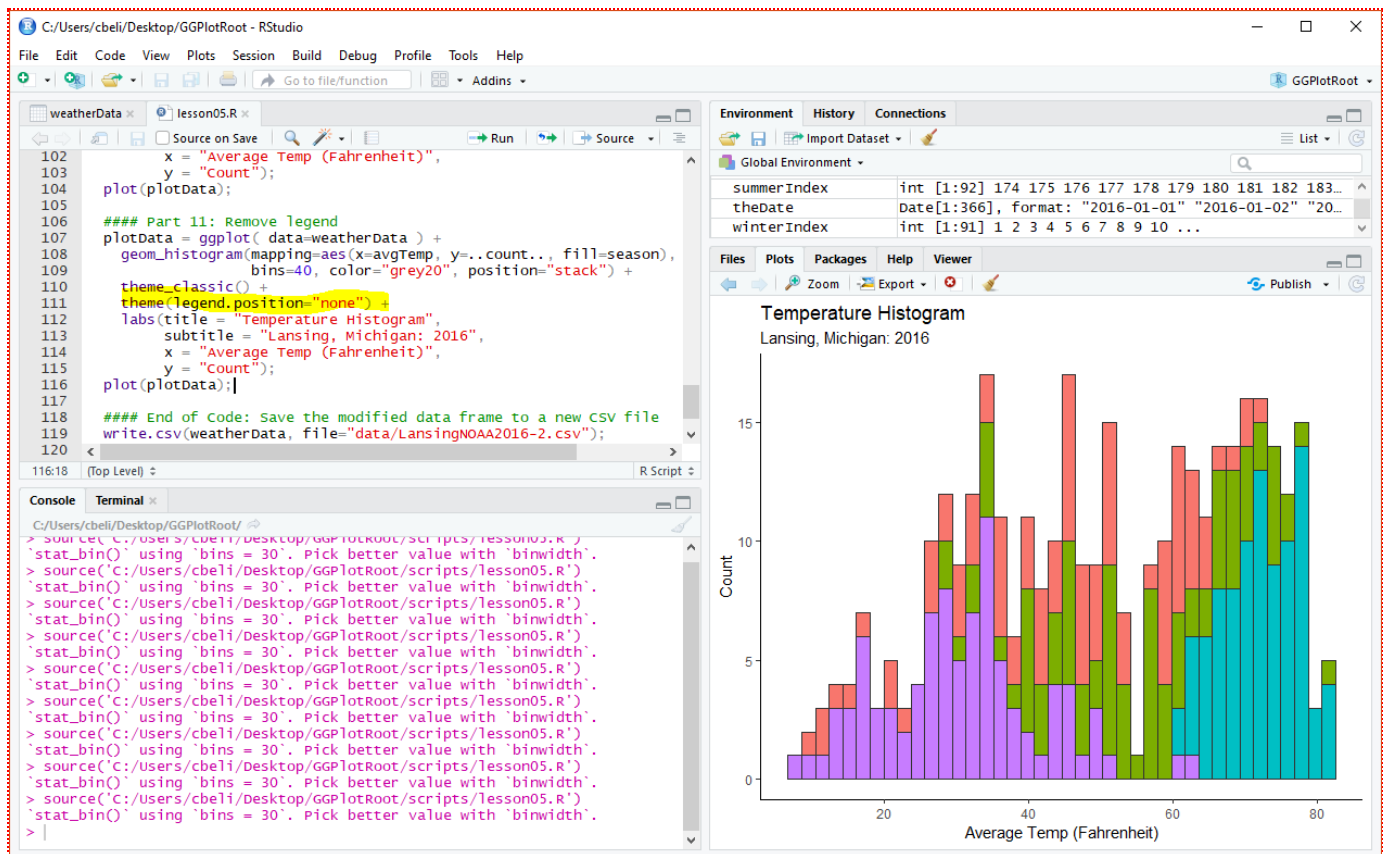


Fig 16: Removing the legend

You can also use **legend.position** to reposition the legend where **x** and **y** are both values from 0 to 1

```
theme(legend.position= c(x=0.15, y=0.75) )
```

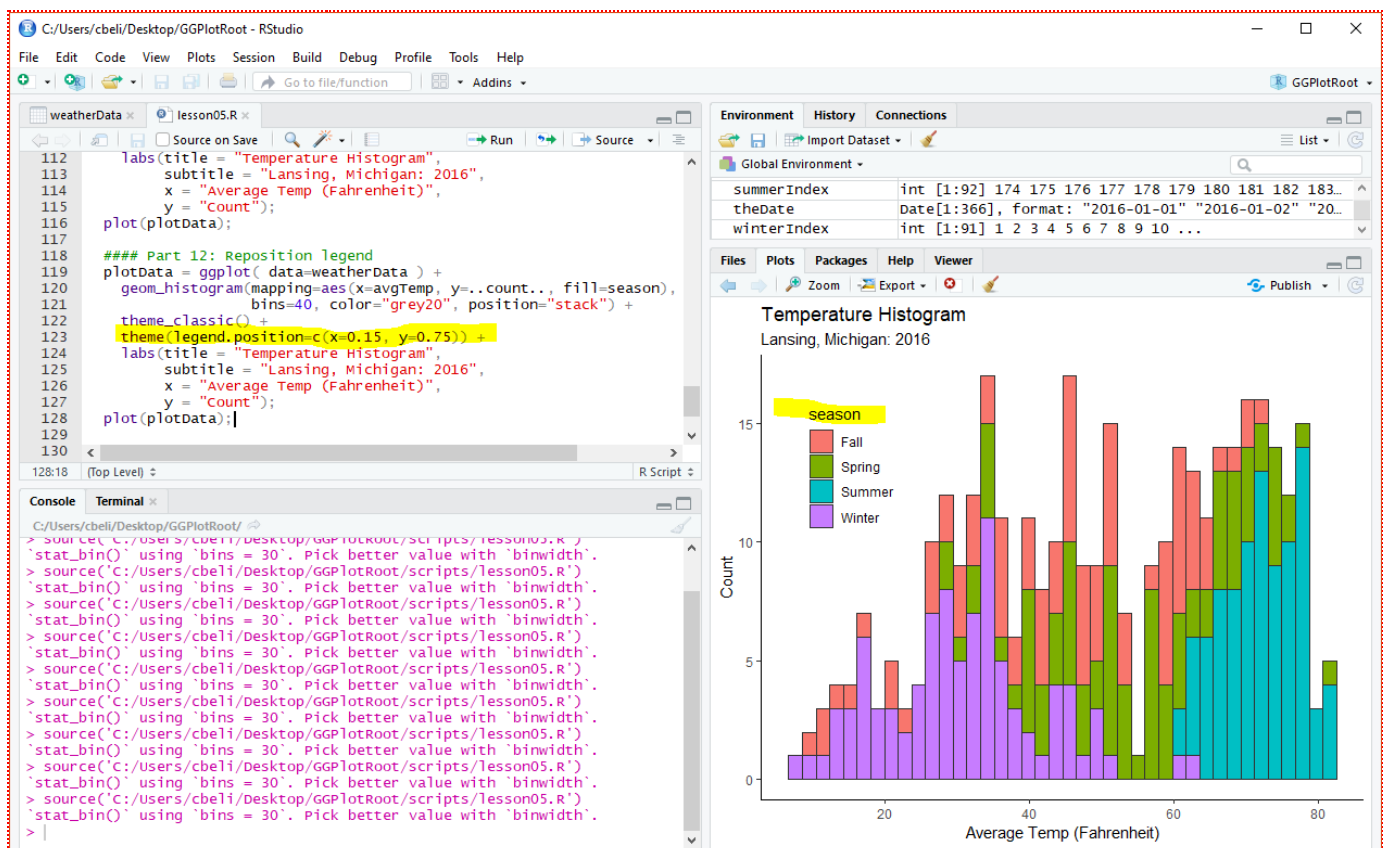


Fig 17: Repositioning the legend

