02-03: Sequences and For Loops

1 - Purpose

- Create various types of sequences
- Use sequences to iterate through a for()
- Use sequences to index a vector inside a for()

2 - Concepts

3 - Iteration (i.e., executing code multiple times)

Programmers often want to execute, or iterate through, similar code multiple times. The most simplistic example is a counter. If you wanted your script to count from 1 to 5, you could simply output the numbers one to five in order:

```
{
  rm(list=ls());  options(show.error.locations = TRUE);

  cat("count is ", 1, "\n");
  cat("count is ", 2, "\n");
  cat("count is ", 3, "\n");
  cat("count is ", 4, "\n");
  cat("count is ", 5, "\n");
}
```

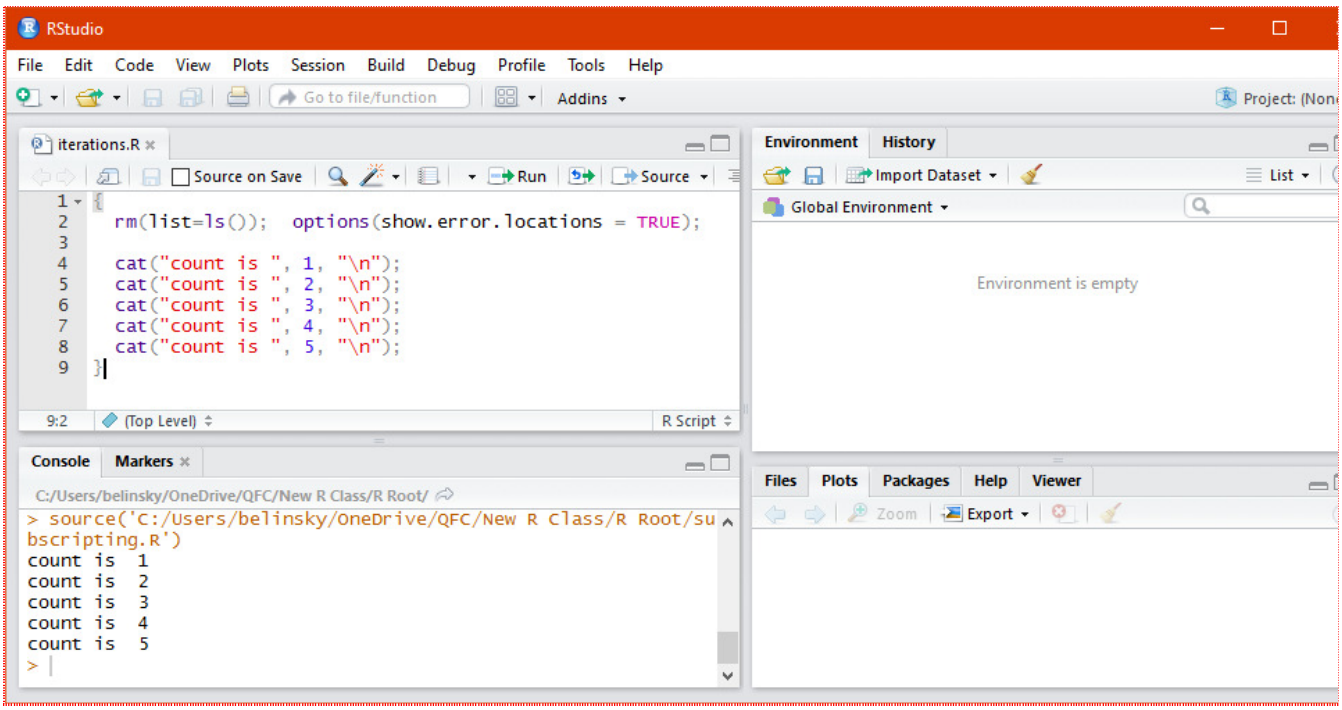

*Fig 1: Output numbers **1** through **5***

Or, perhaps you want to output the first five values from a column in a data frame (in this case, the first five temperatures from the ***highTemp*** column in the ***weatherData*** data frame). Remember: ***weatherData[3,"highTemp"]*** references the **third** value in the ***highTemp*** column.

```
{
  rm(list=ls());  options(show.error.locations = TRUE);

  weatherData = read.csv("data/LansingWeather2.csv");

  cat("1st temp is ", weatherData[1, "highTemp"], "\n");
  cat("2nd temp is ", weatherData[2, "highTemp"], "\n");
  cat("3rd temp is ", weatherData[3, "highTemp"], "\n");
  cat("4th temp is ", weatherData[4, "highTemp"], "\n");
  cat("5th temp is ", weatherData[5, "highTemp"], "\n");
}
```

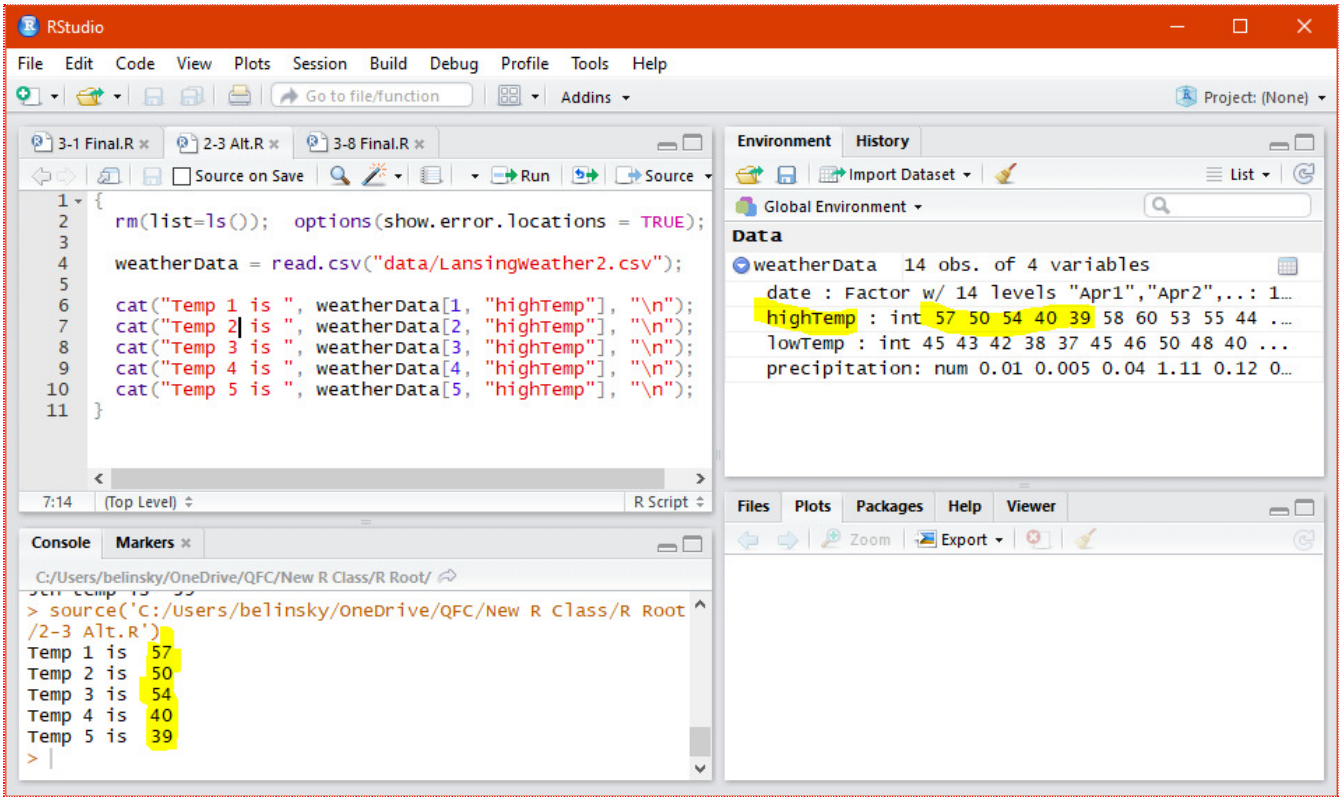

*Fig 2: Output the first five temperatures in the **highTemp** column of **weatherData**.*

While this works, you can see that this would get ugly quick if you wanted to count to 100 or get the first 100 values of a data frame column. And this method is completely impractical for 1000 or 1,000,000 values. Using programming functions we can reduce this above code to just a few lines no matter how high the count is.

Trap: string and factors in data frames

4 - Iterating through a codeblock

The two scripts above execute basically the same code 5 times -- with one small exception. In the count code the *count number* changes, and in the ***weatherData*** code the *vector's index number* changes. In both examples, both the count number and index number iterate sequentially from 1 to 5.

We can reduce the code by instructing R to iterate through a codeblock using the sequence 1 to 5. This is done using a ***for()*** loop.

```
{
  rm(list=ls());  options(show.error.locations = TRUE);

  weatherData = read.csv("data/LansingWeather2.csv");

  for(seqVal in 1:5)  # run the attached codeblock sequentially from 1 to 5
  {
    cat("the count is", seqVal, "\n");
    cat("Temp ", seqVal, " is ", weatherData[seqVal, "highTemp"], "\n");
  }
}
```

The code above outputs the values 1 through 5 and the first five values of the ***highTemp*** column -- just like the previous two scripts.

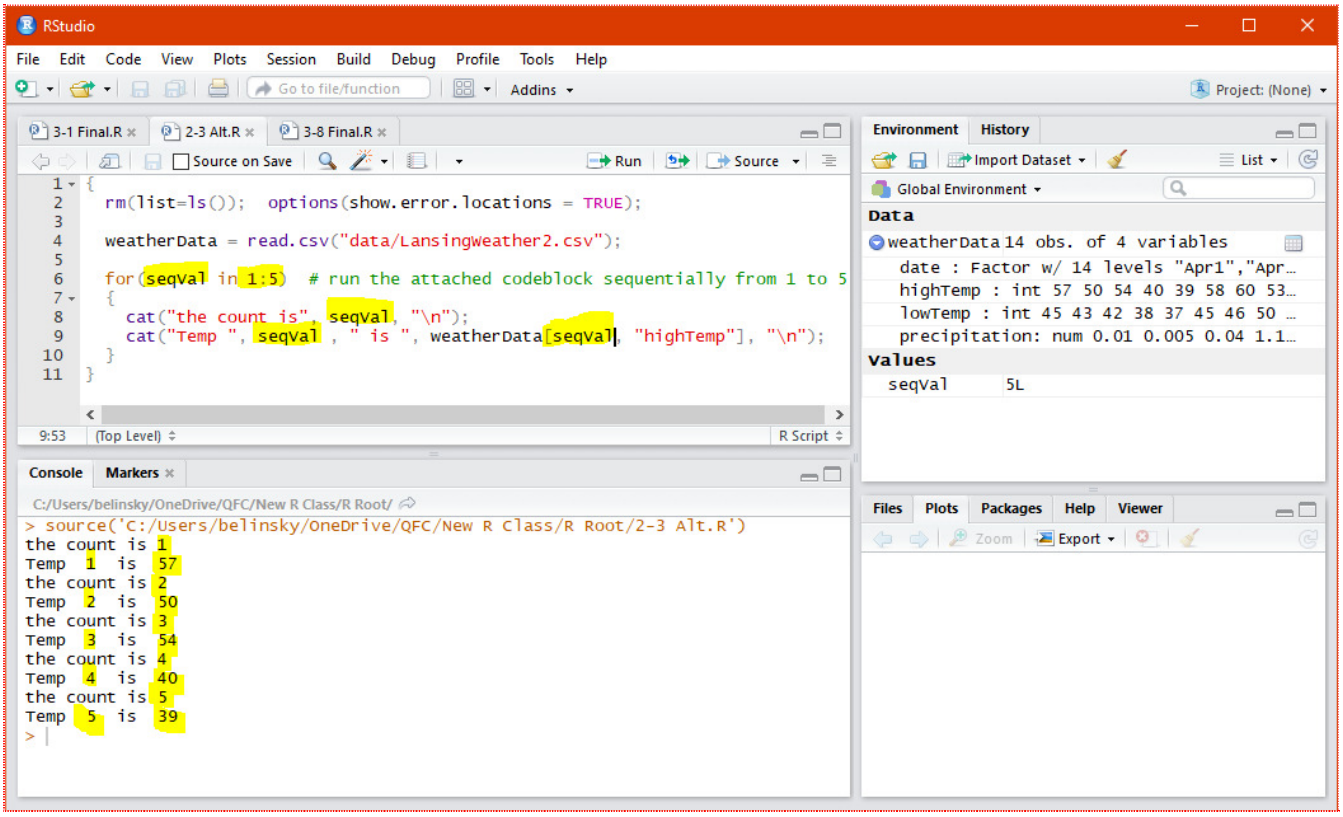

*Fig 3: Using a **for()** to output a sequence of value and vectors.*

There is a whole lot going on in the script and we will go through each part in this lesson and the next lesson but, in a nutshell:

1) ***for()*** will execute the codeblock attached to it (with curly brackets: ***{ }***) multiple times.

2) The number of times the codeblock is executed is determined by the sequence (in this case **1:5**, which is the same as **1** through **5**).
3) At the start of the iteration, the sequence value (1, 2, 3, 4, or 5) is saved to a variable (in this case: ***seqVal***) each time the codeblock is executed (i.e., the value of ***seqVal*** changes each iteration).
4) In line 9, ***seqVal*** is used as an index value to subset the ***weatherData*** data frame.

4.1 - Sequences

In the script above, ***1:5*** creates a sequence that *starts at **1***, *ends at **5***, and *increases by **1***
So, ***1:5*** is a sequence with 5 values: ***1, 2, 3, 4, and 5***. This is the same as **c(1, 2, 3, 4, 5).**

We can create a sequence using any two numbers -- including negative numbers:
1:7 is a sequence with 7 values: ***1, 2, 3, 4, 5, 6, and 7***
4307:4310 is a sequence with 4 values ***4307, 4308, 4309, and 4310***
-5:2 is a sequence with 8 values: ***-5, -4, -3, -2, -1, 0, 1, 2***

The sequence can also go backwards:
5:1 is a sequence with 5 values: ***5, 4, 3, 2, 1***
13:8 is a sequence with 6 values: ***13, 12, 11, 10, 9, 8***

Let's replace the sequence ***1:5*** in the script above with the sequence ***13:8***

```
1  {
2    rm(list=ls());  options(show.error.locations = TRUE);
3
4    weatherData = read.csv("data/LansingWeather2.csv");
5
6    for(seqVal in 13:8)  # run the attached codeblock sequentially from 13 down to 8
7    {
8      cat("the count is", seqVal, "\n");
9      cat("Temp ", seqVal, " is ", weatherData[seqVal, "highTemp"], "\n");
10   }
11 }
```

The script now outputs the count values 13 down to 8 and the script outputs 13th value in the ***highTemp*** column down to the 8th value in the ***highTemp*** column

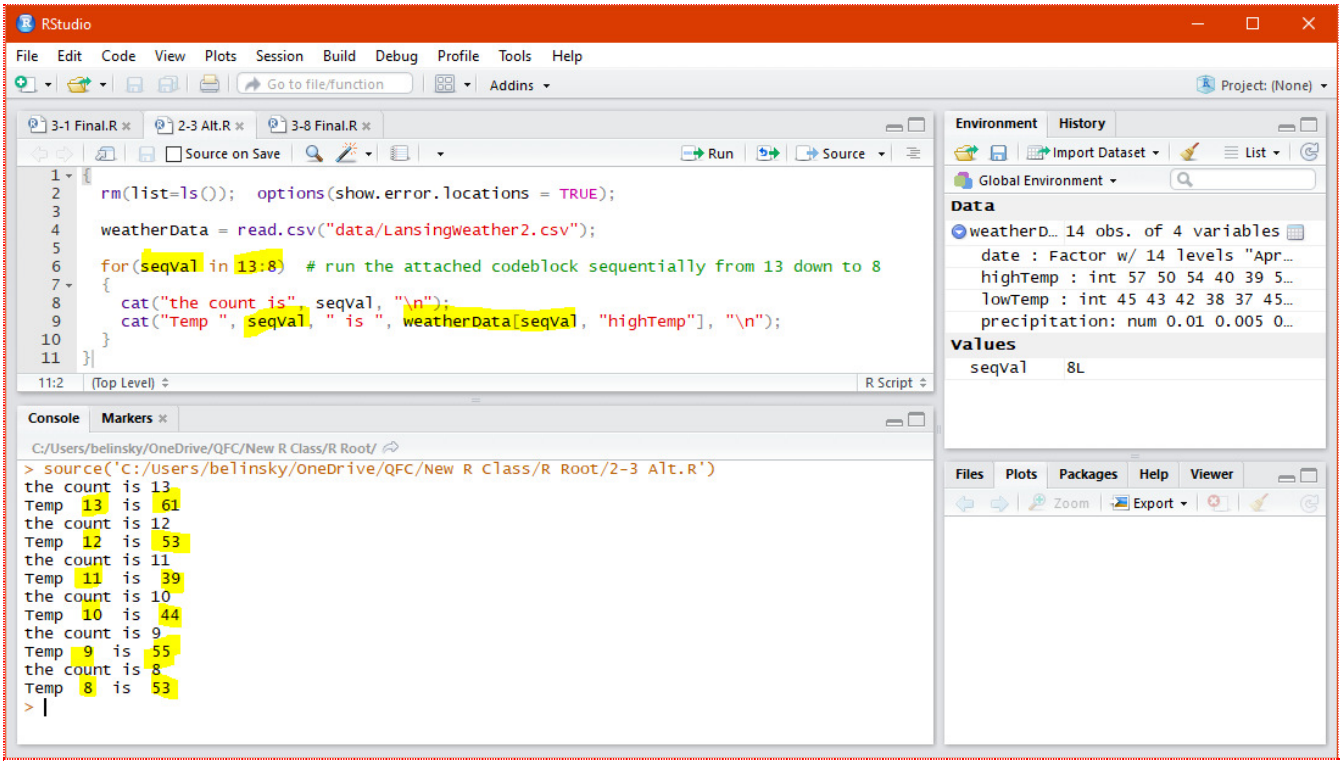

*Fig 4: Going backwards through a sequence using a **for()***

4.2 - The sequence variable

The real power in a ***for()*** is the sequence variable that is declared inside the parentheses -- in this case, ***seqVal***. The name of the variable is completely up to the programmer. For historical reasons, programmers often use the names ***i*** and ***j*** for the sequence variable.

The line

```
1  for(seqVal in 13:8)
```

says that the codeblock attached to the ***for()*** will be executed 6 times, representing the 6 different values that the variable ***seqVal*** will take, which are given by the sequence ***13:8***.

In other words, ***seqVal*** is assigned a new value from the sequence ***13:8*** each of the six times the codeblock attached to the ***for()*** is iterated.
So after the **first** iteration, ***seqVal*** will be **13**
after the **second** iteration, ***seqVal*** will be **12**,
after the **third** iteration, ***seqVal*** will be **11**...

Again, the variable name inside the ***for()*** is up to us and it is probably nice to give it a name that makes sense in context. In this case, I will call the sequence variable ***dayNum*** because I want information for the **4th** through **8th** days in the ***weatherData*** data frame.

```
1  {
2    rm(list=ls());  options(show.error.locations = TRUE);
3
4    weatherData = read.csv("data/LansingWeather2.csv");
5
6    for(dayNum in 4:8)  # run the attached codeblock sequentially from 4 to 8
7    {
```

```
    cat("On day", dayNum, "the high temp was",
        weatherData[dayNum, "highTemp"], "\n");
  }
}
```

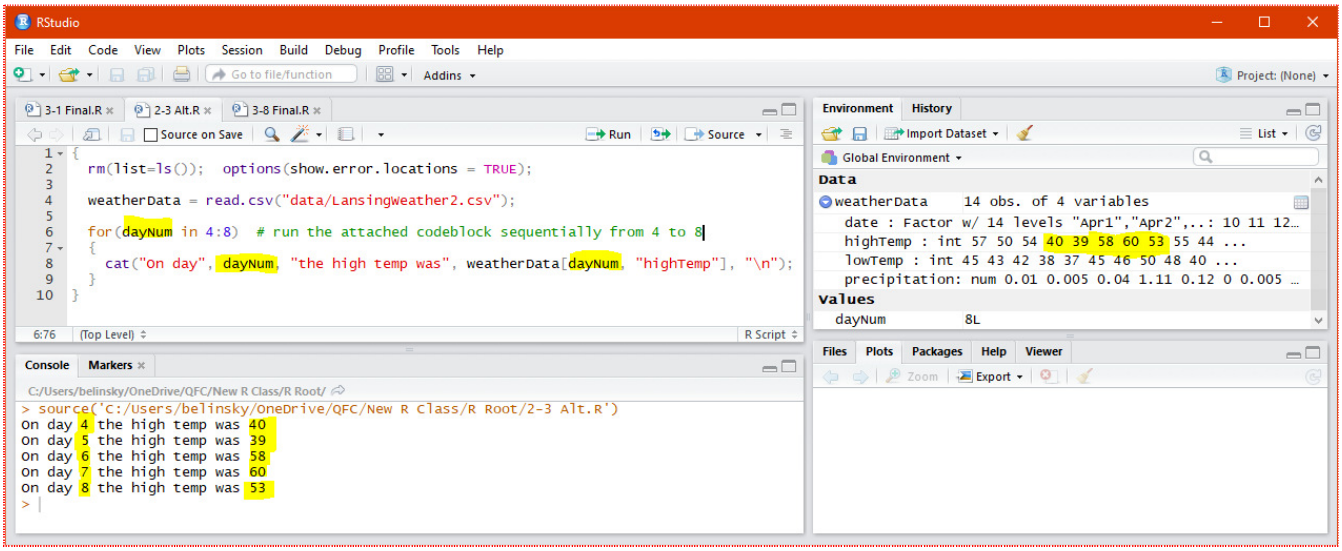

*Fig 5: The **for()** changes the value of the sequence variable each time it executes the codeblock*

5 - More complex sequences

The sequences we have use so far either go up by 1

```
4:8 # goes through sequence 4,5,6,7,8
```

or down by 1

```
13:8 # goes through sequence 13,12,11,10,9,8
```

We can create evenly spaced sequences that increment by numbers other than **1** using the ***seq()*** function.

seq() takes three parameters:

- ***from***: the start value
- ***to***: the end value
- ***by***: the increment value

```
seq(from=1, to=10, by=2);   # start at 1, go to 10, increment by 2
```

We can also go down by using a negative number as the increment:

```
seq(from=13, to=5, by=-3);  # start at 13, go down to 5, increment by -3
```

Let's use the sequences above to look at our weather data:

```
{
  rm(list=ls());  options(show.error.locations = TRUE);

  weatherData = read.csv("data/LansingWeather2.csv");

  cat("First sequence: 1 to 10 by 2\n");
```

```
  for(dayNum in seq(from=1, to=10, by=2))  # go from 1 to 10 by 2
  {
    cat("On day", dayNum, "the high temp was",
         weatherData[dayNum, "highTemp"], "\n");
  }

  cat("\nSecond sequence: 13 to 5 by -3\n");
  for(dayNum in seq(from=13, to=5, by=-3))  # go down from 13 to 4 by -3
  {
    cat("On day", dayNum, "the high temp was",
         weatherData[dayNum, "highTemp"], "\n");
  }
}
```

The ***to*** value in the sequence is the boundary -- the sequence will not go beyond that value but the sequence does not have to use that value. In both examples above the ***to*** value was not used because it was not part of the sequence.

So, ***seq(13, 5, -3)*** executes the loop using the values ***13,10,7.*** The next value in the sequence would be **4** but **4** is not within the sequence.

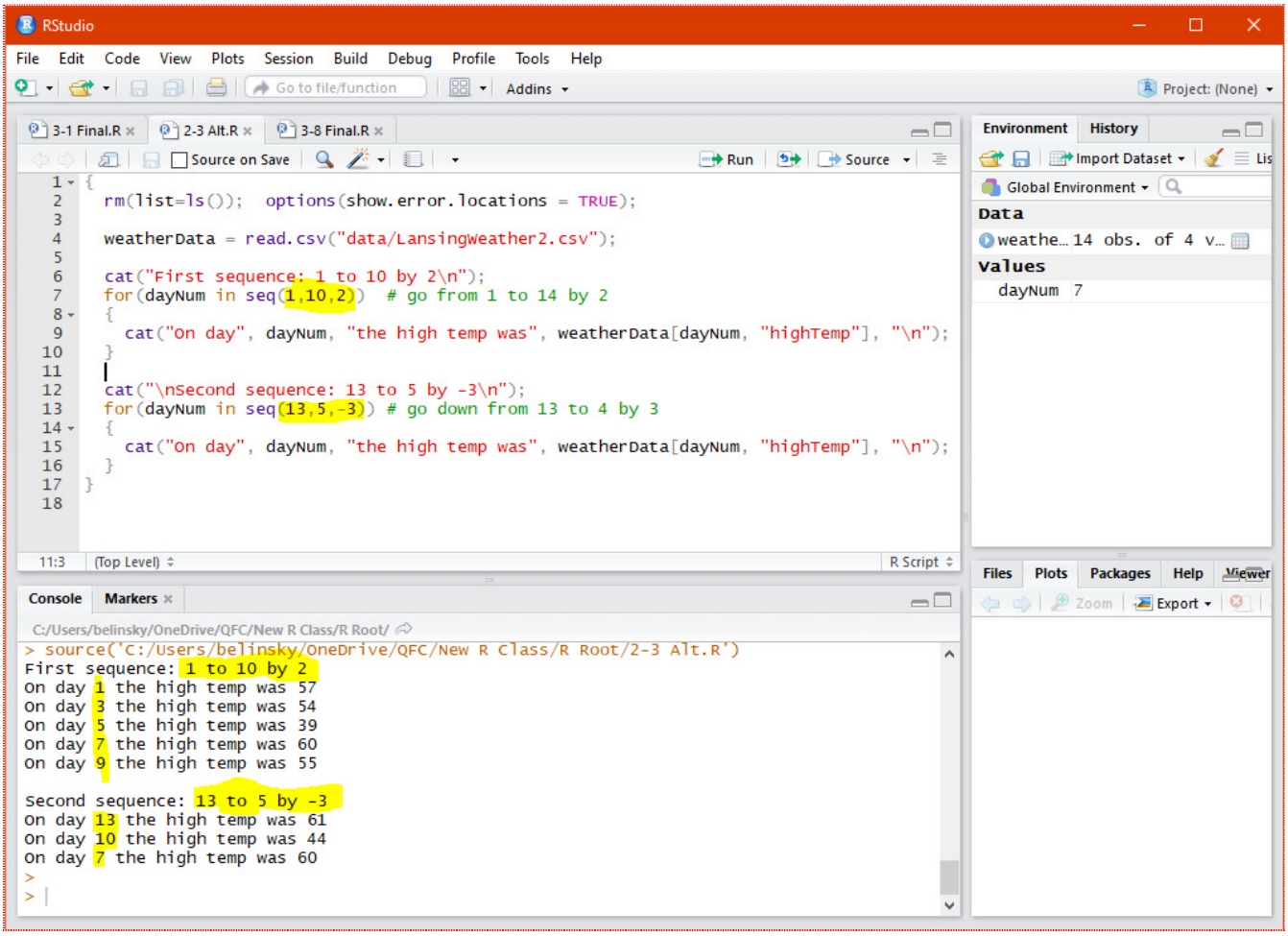

*Fig 6: Using the **seq()** function to create a sequence with increment values other than **1***

6 - Creating a manual sequence

All the sequences that we created above are really vectors -- we can see that if we save the sequence to a variable.

```
{
  rm(list=ls());  options(show.error.locations = TRUE);

  seq1 = 1:5;
  seq2 = 13:8;
  seq3 = seq(from=13, to=4, by=-3);
  seq4 = seq(from=1, to=10, by=2);
}
```

In the Environment Window we can see that the four sequence variables (***seq1, seq2, seq3, seq4***) are vectors that have the values of all the numbers in the sequence.

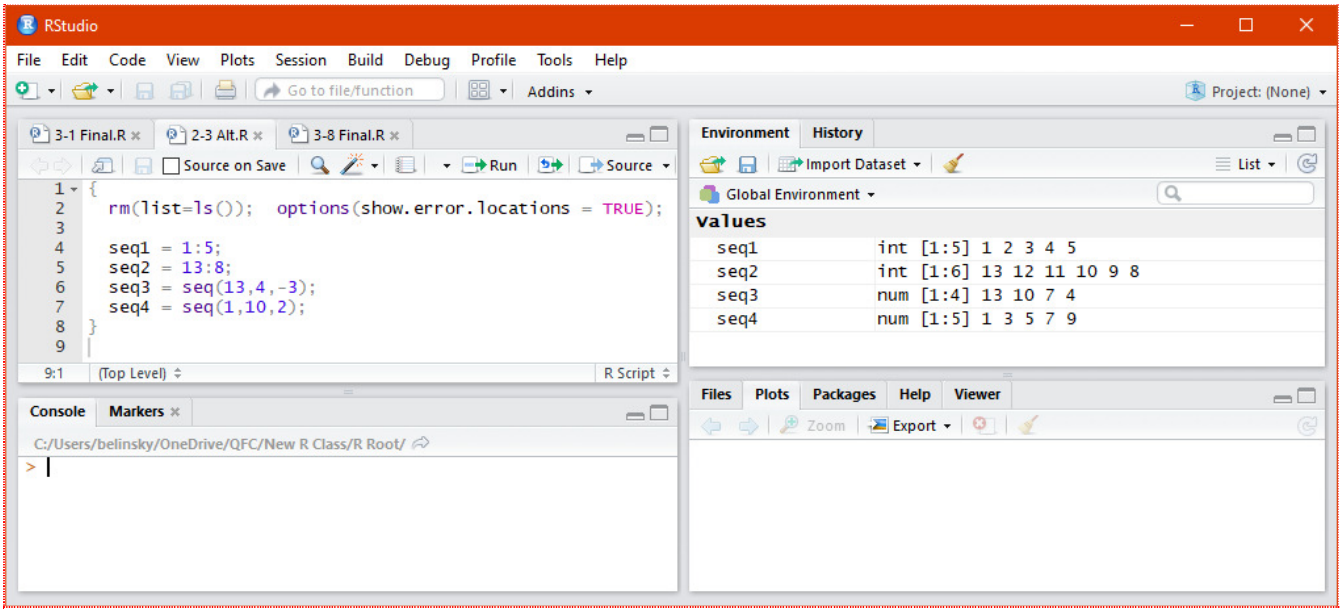

Fig 7: Saving sequences to a vector variable.

We can use any of these variables as the sequence in a ***for().***

```
{
  rm(list=ls());  options(show.error.locations = TRUE);

  seq1 = 1:5;
  seq2 = 13:8;
  seq3 = seq(from=13, to=4, by=-3);
  seq4 = seq(from=1, to=10, by=2);

  weatherData = read.csv("data/LansingWeather2.csv");

```

```
  for(dayNum in seq4)  # seq4 is a vector that holds five values: 1,3,5,7,9
  {
    cat("On day", dayNum, "the high temp was",
         weatherData[dayNum, "highTemp"], "\n");
  }
}
```

seq4 is a vector variable that holds the values 1,3,5,7,9 and the ***for()*** goes through that sequence assigning ***dayNum*** each of the 5 values

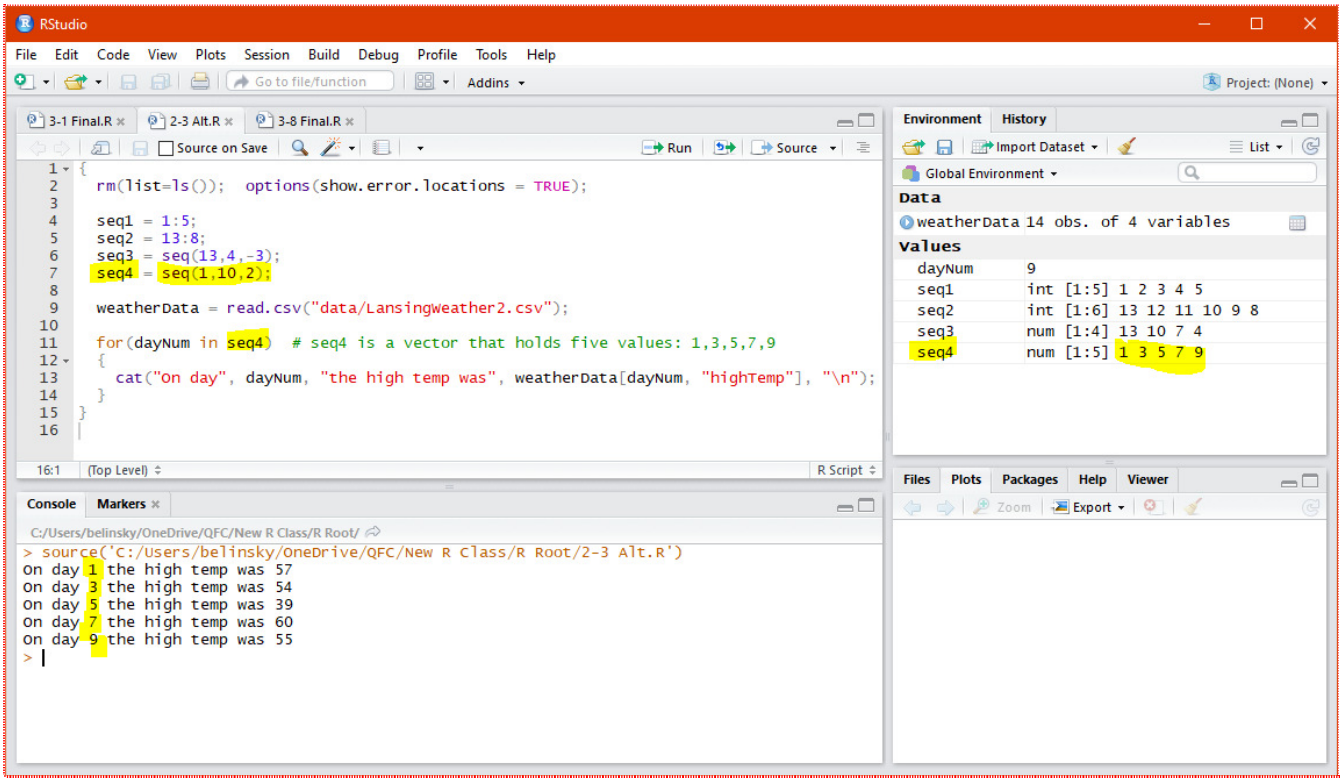

*Fig 8: Using vector variable as the sequence in a **for()***

7 - Manual Sequences

Remember that:

```
  for(i in 1:6)
```

executes the codeblock attached to the ***for()*** for all values in the sequence vector **1:6** (which are 1,2,3,4,5, and 6)

Similarly, the line:

```
  for(dayNum in seq4)
```

tells R to execute the codeblock attached to the ***for()*** for all values in the vector ***seq4.***

seq4 was assigned the values 1,3,5,7,9 using ***seq()***

```
  seq4 = seq(from=1, to=10, by=2);
```

But ***seq4*** is really just *a vector* with values 1,3,5,7, and 9. We can also manually create vectors to use as a sequence in a ***for().***

To create a vector we use the ***c()*** function.

```
{
  rm(list=ls());  options(show.error.locations = TRUE);

  manualSeq = c(7,3,9,1);  # creates a vector with four values: 7,3,9,1
}
```

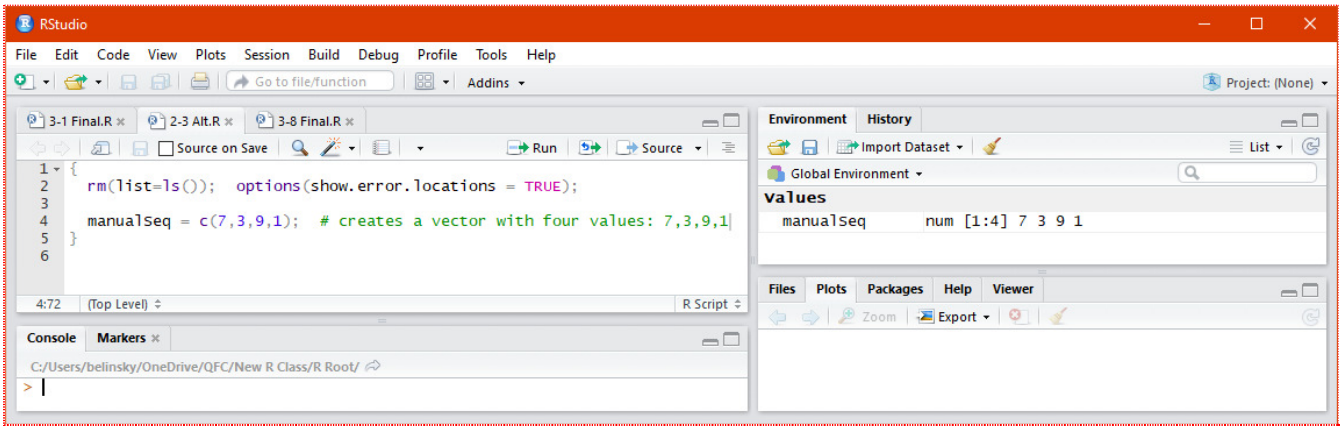

Fig 9: Creating a manual sequence and saving it as a vector.

And we can use ***manualSeq*** in the ***for()***

```
{
  rm(list=ls());  options(show.error.locations = TRUE);

  manualSeq = c(7,3,9,1);  # creates a vector with four values: 7,3,9,1

  weatherData = read.csv("data/LansingWeather2.csv");

  for(dayNum in manualSeq)  # creates a vector with four values: 7,3,9,1
  {
    cat("On day", dayNum, "the high temp was",
        weatherData[dayNum, "highTemp"], "\n");
  }
}
```

In the script above, the ***for()*** executes the codeblock attached to it **4** times, which is the length of the vector ***manualSeq***. Each time the codeblock executes, ***dayNum*** is iteratively assigned the next value from the vector ***manualSeq***.

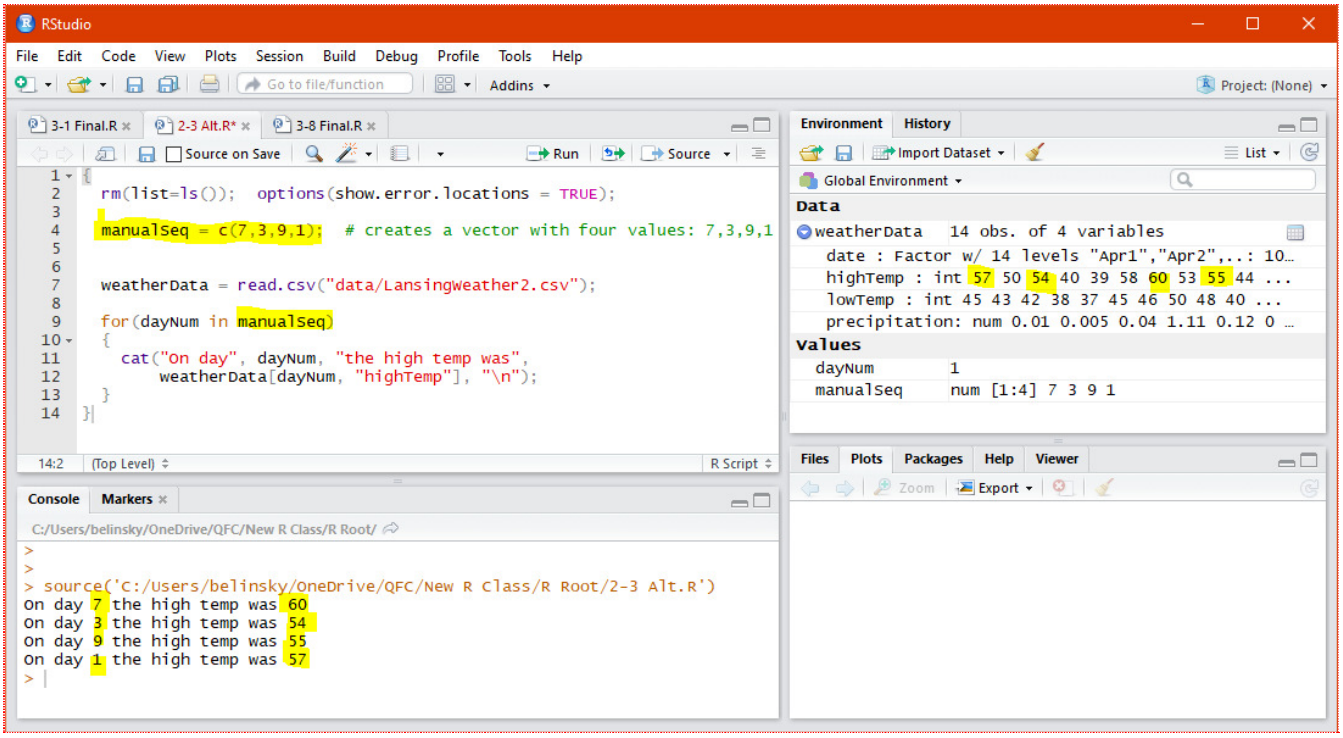

*Fig 10: Using a manual sequence to iterate through a **for()***

Inside the codeblock, ***dayNum*** is used to both specify the day and to index the ***highTemp*** vector from ***weatherData***

8 - Application

A) Create two vectors:
1) one that has all integer values from 45 to 167 in ascending order.
2) one that has all integer values from 25 down to -7 in descending order.

B) Use the root and power functions on values in a sequence vector:
1) Create a sequence that goes from 17 to 5 and increments down by 3 -- save the sequence to a vector variable.
2) Create a ***for()*** that uses the vector variables created in part A.
3) In the codeblock attached to the ***for()***, output the cubed power (3rd power) and the cubed root (1/3rd power) of each value in the vector.

C) Create a ***for()*** that outputs to the Console Window the date and precipitation for the 13th, 3rd, 5th, 8th, and 7th days from ***LansingWeather2.csv*** (make sure it is in the order 13, 3, 5, 8, 7)

Note: you will need to add the parameter ***stringAsFactor*** = **FALSE** to ***read.csv()*** to make part C work. Otherwise, you will oputput the factor number instead of the date. This is demonstrated in the Application Answers folder and there is more about this in *Trap: Strings and factors in a data frame*

D) Challenge: Go through the all the precipitation values using a ***for()*** and output to the Console Window the days that have more than 1 inch of rain. Hint: use an ***if()*** inside the ***for()***.

9 - Trap: Strings and factors in data frames

When we save the data from ***LansingWeather2.csv*** to a data frame, the **date** column gets converted to a *factor variable* (or a categorical variable).

```
{
  rm(list=ls()); options(show.error.locations = TRUE);

  weatherData = read.csv("data/LansingWeather2.csv");
}
```

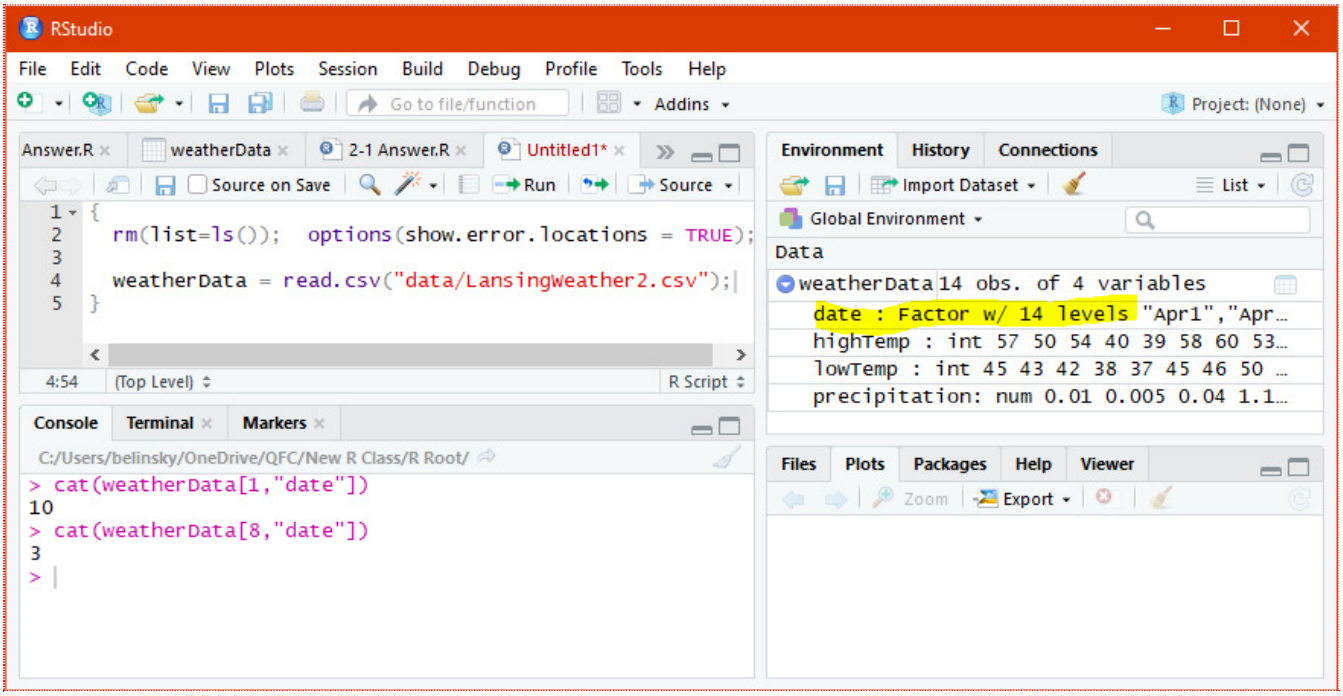

*Fig 11: The date column in **weatherData** by default is treated as a factor, or categorical variable.*

Any time there is a non-numeric value in a column of a data frame, R will categorize the column as factors. In this case, the **date** column has **14** different value, because it has **14** different dates. Each unique value gets its own factor or category so there are 14 levels (1 through 14). The "levels" are determined alphabetically so **Apr1** is **1**, **Apr2** is **2**,... and **Mar31**, alphabetically last, is **14**. In *Fig 11*, it shows that the first value, **Mar27**, is **10** and the **8th** value, **Apr3**, is **3**.

For the purposes of this class, this behavior just causes headaches and there is no real need for it. Many vectors are converted into factors unnecessarily all of the function we use in this class will convert a vector to a factor if needed -- we will see this in unit 3.

By default, it is probably best to treat vectors with non-numeric values as strings. We can do this by adding a parameters to ***read.csv()*** called ***stringAsFactor.***

```
{
  rm(list=ls()); options(show.error.locations = TRUE);

  weatherData = read.csv("data/LansingWeather2.csv",
                         stringAsFactors = FALSE);
}
```

Notice that now when we output values from the ***date*** vector, the values are given instead of the categorical level.

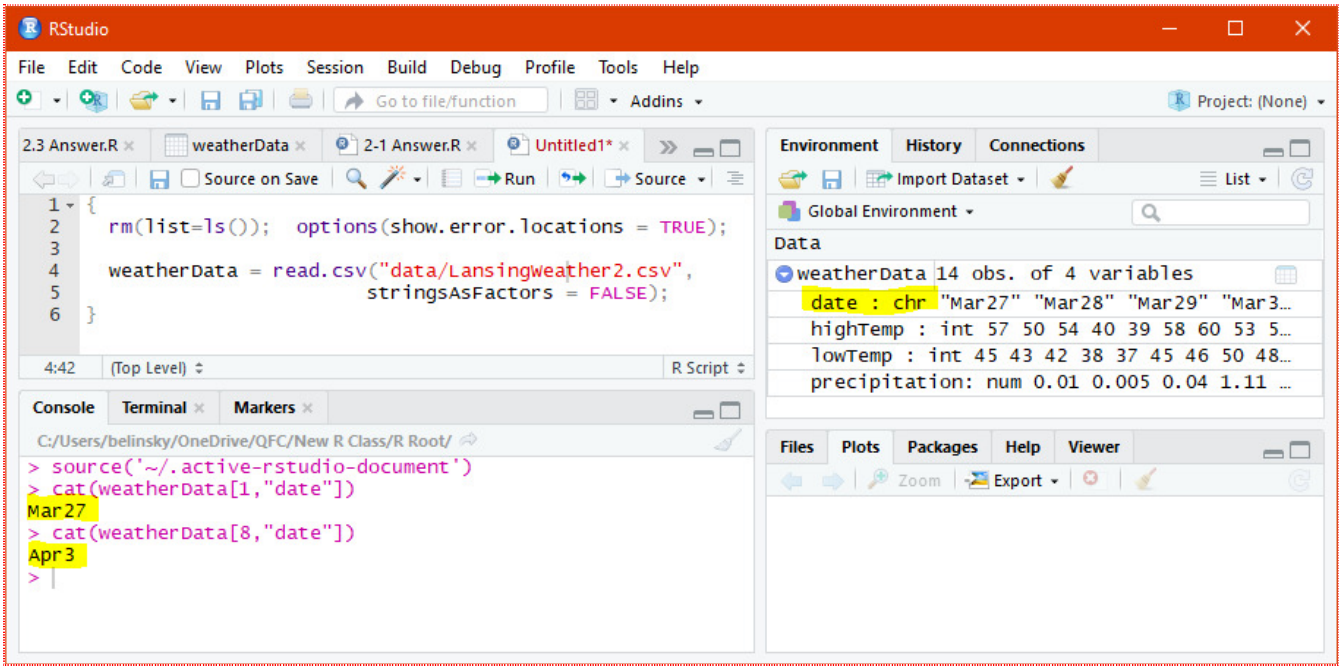

*Fig 12: Treating the **date** column as a string instead of a factor.*