02-10: Functions 2

1 - Purpose

- create a separate file to hold functions
- create a counter function
- use functions saved in a separate file

1.1 - Student-Instructor meeting

This is the last lesson in Unit 2, which means it is time to schedule a 20-30 minute meeting with the instructor. Please email the instructor and provide some times that you are available.

2 - Questions about the material...

If you have any questions about the material in this lesson or your Class Project *feel free to email them to the instructor here*.

3 - barplot(), boxplot(), hist(), plot() are all functions...

Two lessons ago we created our own reusable code, or function, called ***pythogoras()***. In the last lesson, we used built-in functions (***hist()***, ***barplot()***, etc..) to create various types of plots. Unlike ***pythagoras()***, where the function exists inside your script file, all of the plotting functions exist outside the script file -- and the plotting functions we use are automatically loaded in R and can be called from any script. We can do the same with ***pythagoras(),*** or any function that we create. In other words, we can create functions that can be called by a different script.

4 - Creating an external file for functions

Functions are like tools in that they are called upon to perform a specific task. It is often useful to create a script that only contains functions -- sort of a toolbox script that performs commonly used tasks that can be called from other scripts.

We are going to create a toolbox script, called ***toolbox.r***, and start by adding ***pythagoras()*** to it. To do this:
1) In RStudio, create a new script file
2) Copy and paste the ***pythagoras()*** function (below) to the new script file
3) Save the file as ***toolbox.r*** inside the ***script*** folder.

```
pythagoras = function(a,b)
{
   c = (a^2 + b^2)^(1/2);
   return(c);
}
```

If we click ***Source*** on ***toolbox.r*** the script does nothing except list the ***pythagoras()*** function in the Environment Window. This is because ***pythagoras()*** does not do anything until it is called upon. However, R now knows that ***pythagoras()*** is a defined function.

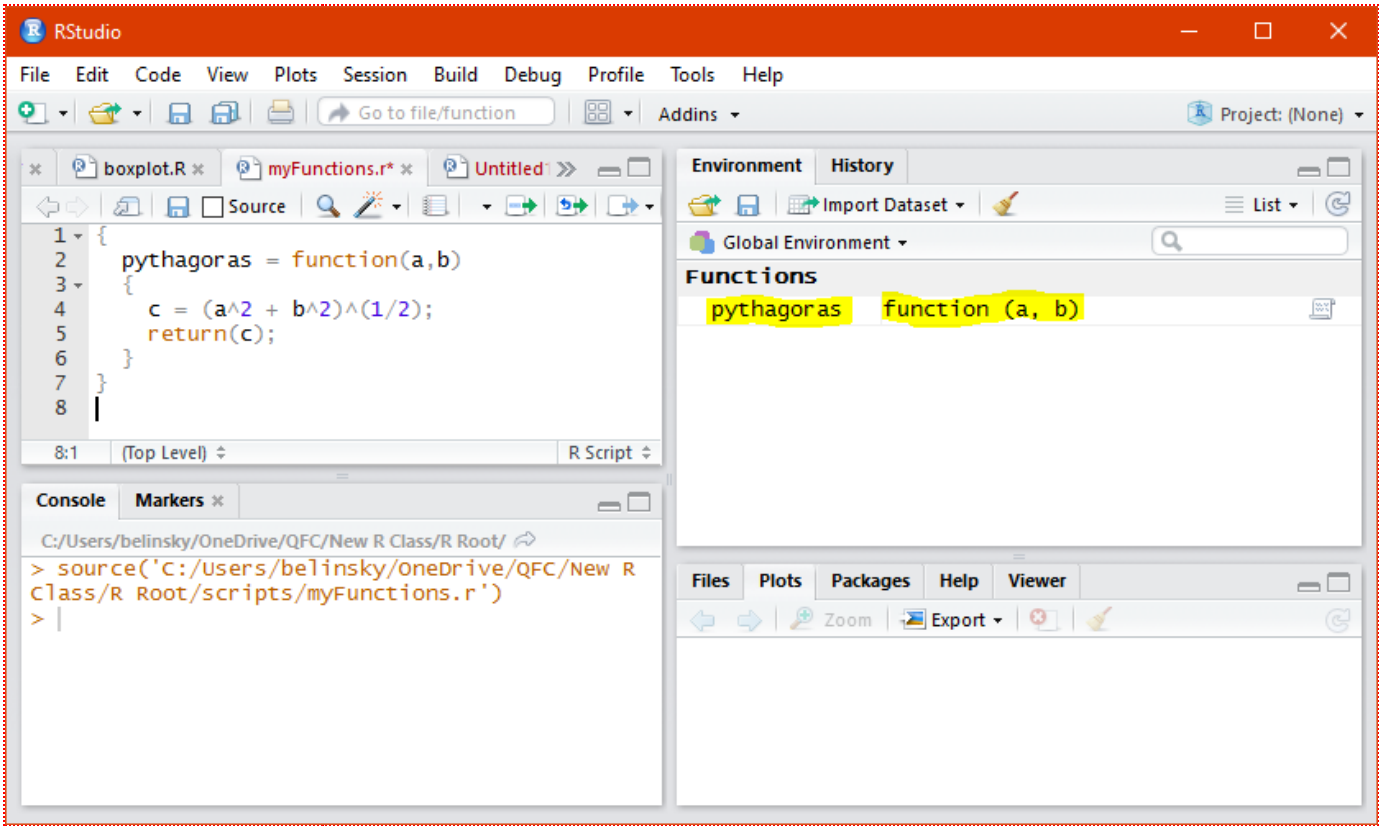

*Fig 1: Adding **pythagoras()** to a functions script*

Note: We do not include ***rm(list=ls())*** in a functions script. If we did, it would remove all the variables created in the script that called the functions script.

5 - Using the functions

We want any script file to be able to use the ***pythagoras()*** function that is included in ***toolbox.r***. To do this, we just need to put code inside the script file that reads ***toolbox.r***. We can do this using the ***source()*** function and the parameter is the location and name of the script file you want to include.

This code tells R to include everything in the ***toolbox.r*** file (which is in the ***scripts*** folder) in your current script file. Functionally, this is equivalent to running all the code in toolbox.r by manually clicking the Source button when that script is in focus.

```
1 source(x="scripts/toolbox.r");  # load script with pythagoras() function
```

Since everything inside ***toolbox.r*** is now part of this new script, we can call ***pythagoras()*** (line 4) *as if pythagoras() existed in our new script*. The following script will execute ***pythagoras()*** located in ***toolbox.r*** and get a return value, which is assigned to ***hypoteneuse***:

```
1 {
2   rm(list=ls()); options(show.error.locations = TRUE);
3   source(x="scripts/toolbox.r");
```

```
4
5   hypoteneuse = pythagoras(8,12);
6 }
```

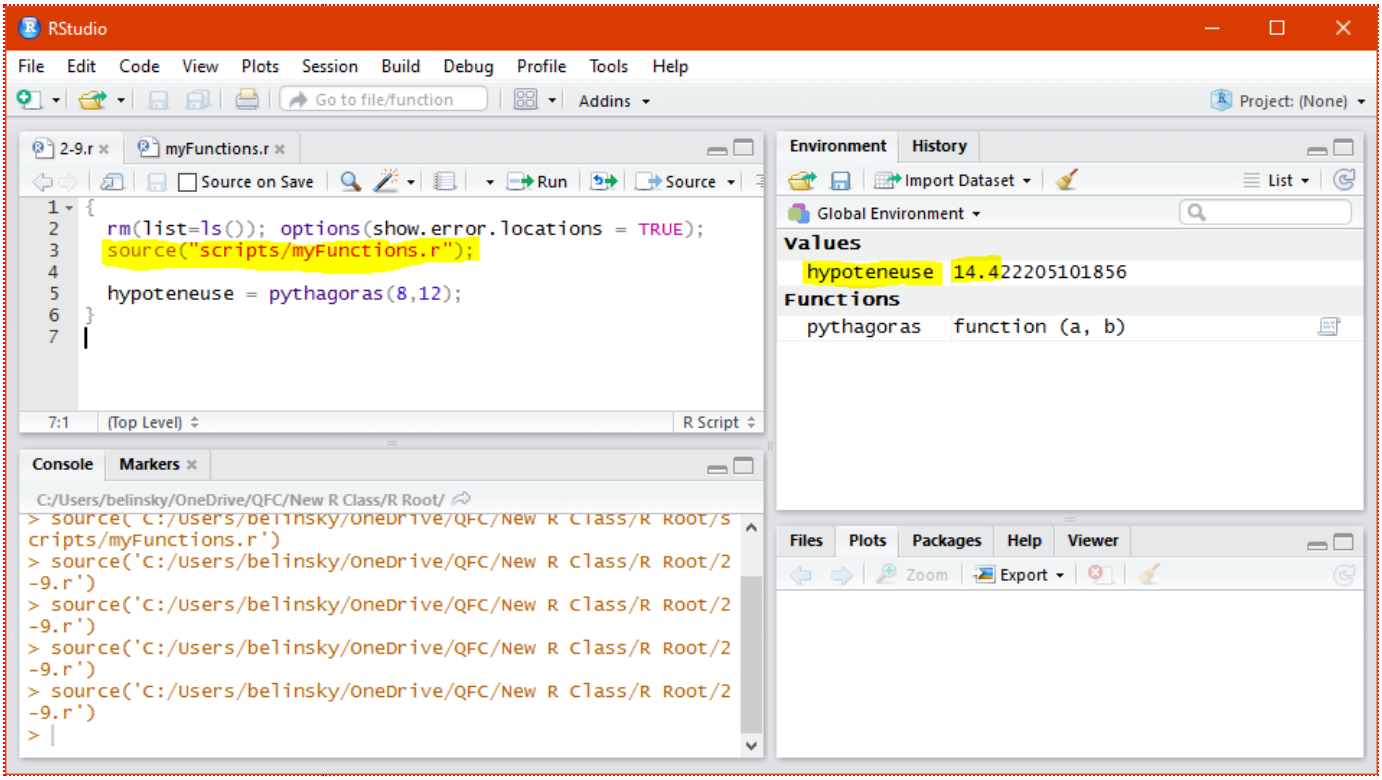

*Fig 2: Calling **pythagoras()** from a separate script*

We will be adding more functions to the ***toolbox.r*** script file in this lesson and in later lessons.

5.1 - Common source() errors

A really easy mistake is to name the source file wrong, have the wrong folder name, or the wrong folder path. In all of the cases RStudio will give you a *"No such file or directory"* error.

The following code erroneously gives the folder name as "script" instead of "scripts":

```
1 {
2   rm(list=ls()); options(show.error.locations = TRUE);
3   source("script/toolbox.r");  # error here
4
5   hypoteneuse = pythagoras(8,12);
6 }
```

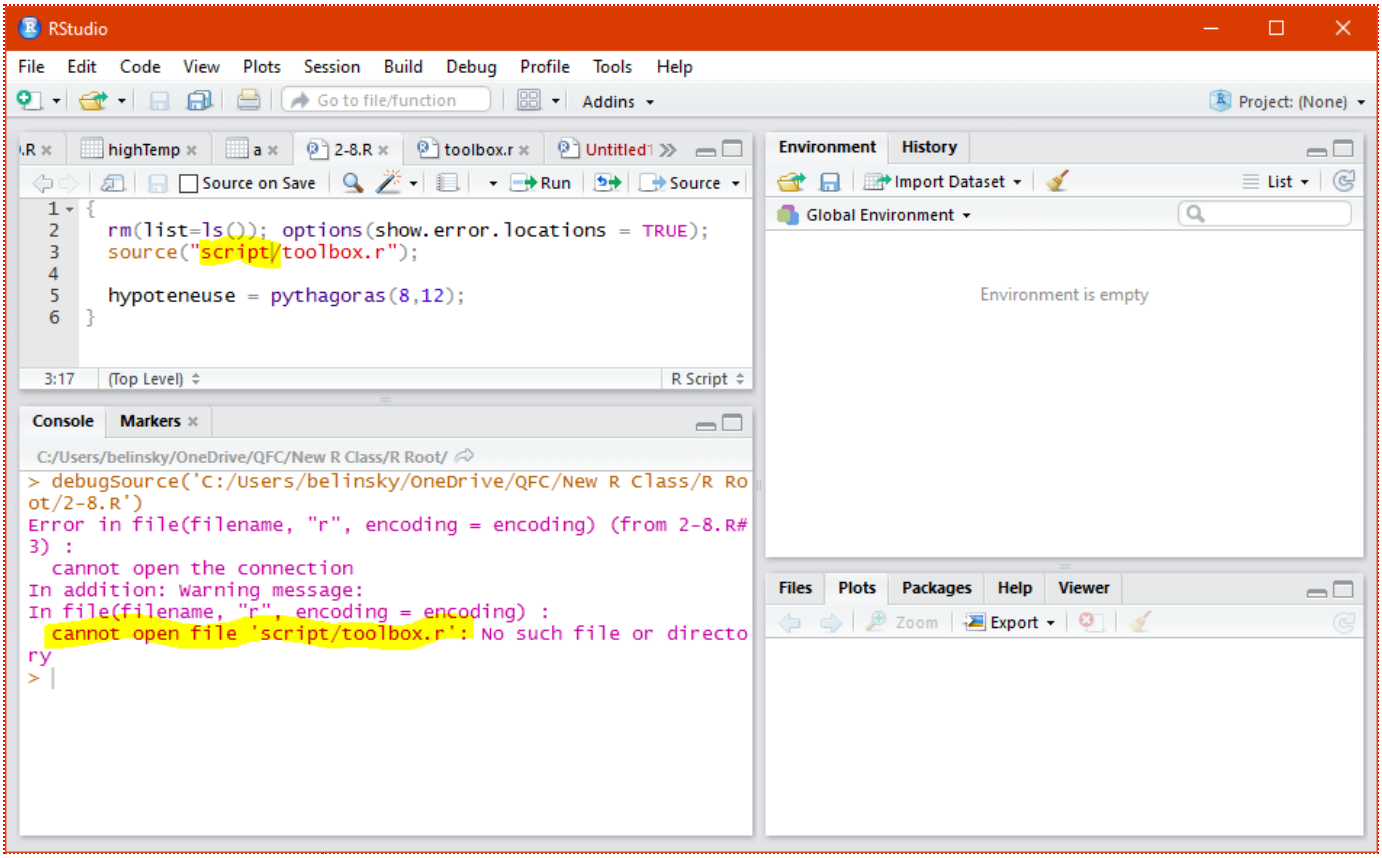

Fig 3: Error calling the source file

6 - Adding to our toolbox: the counter function

Now we will add a counter function to ***toolbox.r***. ***counter()*** will count up all the values in a numeric vector that meet a user-specified condition.

So counter will be able to do things like:
a) count all temperature values greater than 40 in the ***highTempData*** vector
b) count all non-zero precipitation values in the ***precipData*** vector

And, just like ***pythagoras()*** returns the third side of the right-triangle, ***counter()*** will return the count to the caller.

6.1 - Adding counter() to toolbox.r

In our first version of this function, ***counter()*** will count the number of values *in a vector* that are greater than *some comparison value*.

This means there are two parameters for ***counter()*** that are *assigned values by the caller*:
1) ***vector***: the vector to be searched through
2) ***compareVal***: the value used for comparison when determining whether each of the elements meets the condition.

So, in ***toolbox.r***, we will create a function called ***counter()*** that takes two values from the caller

```
1 counter = function(vector, compareVal)
2 {
3
```

```
4 }
```

Remember that ***vector*** and ***compareVal*** are parameters whose *values get assigned by the caller.*

6.2 - Going through each value in the vector

The first thing we want the function to do is to go through all the values in the vector that the caller supplied.

To do that we need to:
1. find the number of values in the vector using ***length()***
2. create a ***for()*** that iterates through each value in the vector

```
1 counter = function(vector, compareVal)
2 {
3   vecLength = length(vector);  # get the length of the vector
4   for(val in 1:vecLength)      # go through each value in vector
5   {
6     # counting script will go here
7   }
8 }
```

We get the length of the vector in line 3 by using the ***length()*** function:

```
4 vecLength = length(vector);
```

and we use the length to give the number of iterations -- the sequence ***1:vecLength***. Note: ***val*** will change each iteration through the ***for()*** going from **1** to ***vecLength.***

```
5 for(val in 1:vecLength)
```

6.3 - Checking each value

The ***for()*** will iterate through each vector value, using ***val*** as the index. Now we need to check each vector value against ***compareVal*** supplied by the caller.

Inside the ***for()***, we check each indexed value to see if it is greater than the value given by the caller

```
1 counter = function(vector, compareVal)
2 {
3   vecLength = length(vector);  # get the length of the vector
4   for(val in 1:vecLength)      # go through each value in vector
5   {
6     # check if the vector value is greater than the compareVal
7     if(vector[val] > compareVal)
```

```
    if(vector[val] > compareVal)
    {
      # add one to the count
    }
  }
}
```

6.4 - Creating a count value

The function still does not do anything. We need a way to capture how many times the indexed vector value was greater than the one supplied by the caller.

So we need a *state variable* that will hold the count -- we will call it ***countVal*** and initially assign ***countVal*** the value of **0**.

```
countVal = 0;
```

If no values in the vector are greater than the ***compareVal***, ***counter()*** will return ***0*** to the caller

countVal is increased by one each time the following conditional statement is ***TRUE***

```
if(vector[val] > compareVal)
```

Increasing ***countVal*** by one is the same as saying *"assign the value **(countVal** + 1) to **countVal**"*. In R this is:

```
countVal = countVal + 1;   # intermediate states for countVal
```

Putting all the code together:

```
counter = function(vector, compareVal)
{
  vecLength = length(vector);  # get the length of the vector
  countVal = 0;                # initialize the count to 0
  for(val in 1:vecLength)    # go through each value in vector
  {
    # check if the vector value is greater than the compareVal
    if(vector[val] > compareVal)
    {
      countVal = countVal + 1; # add one to the count
    }
  }
}
```

6.5 - Returning the count to the user

The last step is to return the final state of ***countVal*** to the caller using the ***return()*** function.

```
counter = function(vector, compareVal)
{
  vecLength = length(vector);  # get the length of the vector
  countVal = 0;                # initialize the count to 0
  for(val in 1:vecLength)      # go through each value in vector
  {
    if(vector[val] > compareVal)
    {
      countVal = countVal + 1; # add one to the count
    }
  }
  return(countVal);  # return the count value to the caller
}
```

Now your ***toolbox.r*** file should look like this:

```
pythagoras = function(a,b)
{
   c = (a^2 + b^2)^(1/2);
   return(c);
}
counter = function(vector, compareVal)
{
  vecLength = length(vector);  # get the length of the vector
  countVal = 0;                # initialize the count to 0
  for(val in 1:vecLength)      # go through each value in vector
  {
    if(vector[val] > compareVal)
    {
      countVal = countVal + 1; # add one to the count
    }
  }
  return(countVal);  # return the count value to the caller
}
```

7 - Calling the function

Now we have the ***counter()*** function in ***toolbox.r*** and we have included ***toolbox.r*** in our external script, we can use ***counter()*** from our new script.

The following script makes 3 calls to ***counter()*** and assigns the return value to a variable (***count1***, ***count2***, and ***count3***).

```
{
  rm(list=ls()); options(show.error.locations = TRUE);
  source("scripts/toolbox.r");   # load functions from toolbox.r file

  weatherData = read.csv("data/LansingWeather3.csv");
  highTempData = weatherData[ ,"highTemp"];
  lowTempData = weatherData[ ,"lowTemp"];

  count1 = counter(vector = highTempData, compareVal = 50);
  count2 = counter(vector = highTempData, compareVal = 60);
  count3 = counter(lowTempData, 40);
}
```

In the Environment Window, you can see that:

- the number of ***highTempData*** values greater than 50 is ***9*** (***count1***)
- the number of ***highTempData*** values greater than 60 is ***2*** (***count2***)
- the number of ***lowTempData*** values greater than 40 is ***10*** (***count3***)

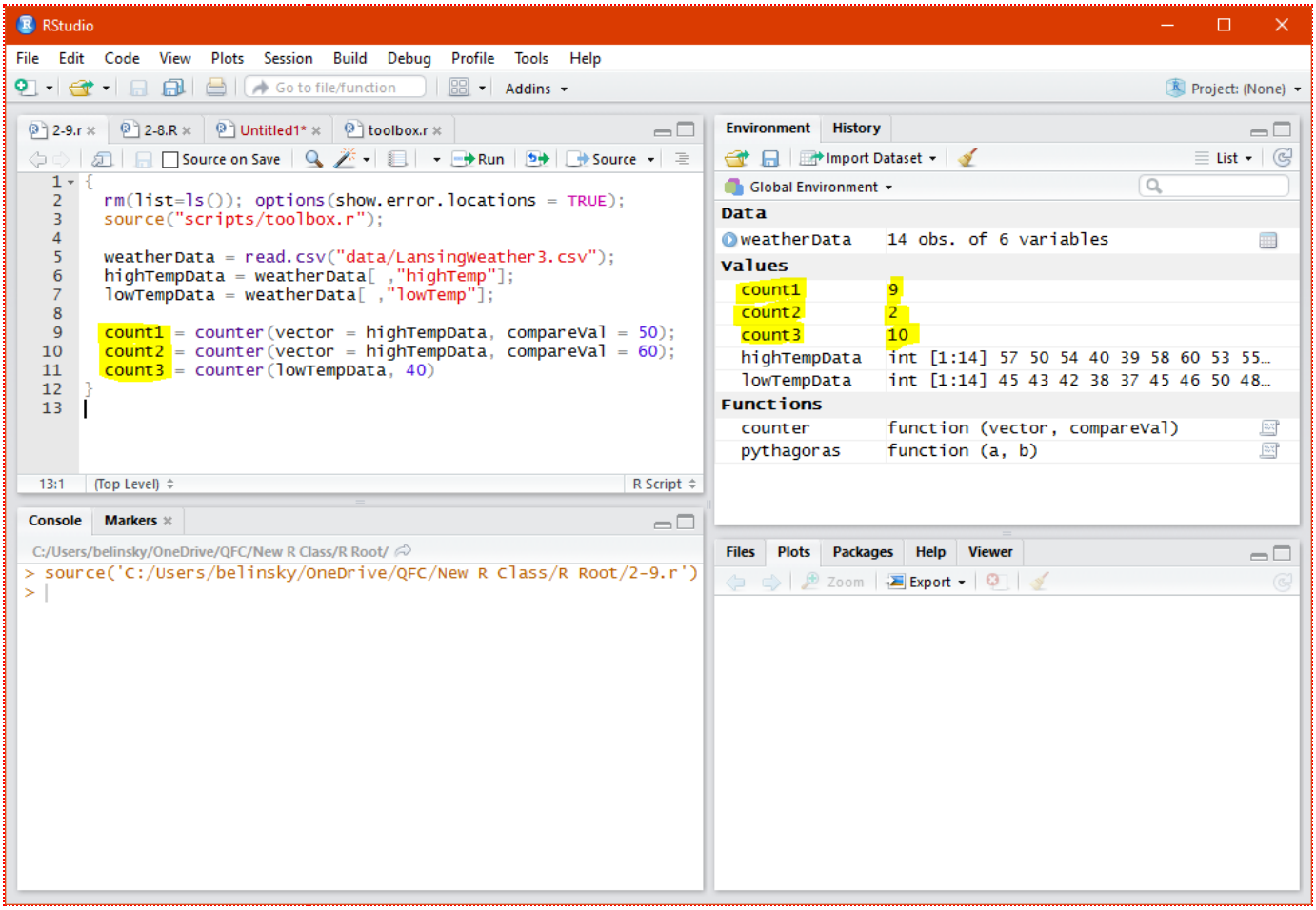

*Fig 4: Calling **counter()** multiple times from an outside script*

Note: you can be explicit and use the parameter names in the function call (e.g., ***count1, count2***) or just put the values in the call (e.g., ***count3***). If you just put in the values, make sure you put them in the right order.

Trap: Putting parameters values in the wrong order

8 - Allowing for more conditional operators

As of right now, ***counter()*** only uses the greater than (**>**) conditional operator when comparing the vector values to another value. We want to make the ***counter()*** more robust by allowing the user to choose the conditional operator.

For the next example we will ***give counter()*** the ability to compare vector values with a comparison value using one of three conditional operators: greater than (>),less than (<), or equal to (==).

To do this, *we need a third parameter in the arguments of **counter()***, which we will call ***conditionalOp*** (for *condition operator*) and we will set the default value for ***conditionalOp*** to greater than (>). So, if the caller does not assign a value to ***conditionalOp*** then the value will be ">".

```
1 counter = function(vector, compareVal, conditionalOp=">")
```

The code will allow for three values to be assigned to the parameter ***conditionOp***: ">", "<", and "==" and the script needs to check for each of these cases. This is an ***if-else-if*** structure.

There are three ways in which the count value will increase by one:

1. if ***conditionalOp*** is ">" and the vector's *value is greater than* ***compareVal***
2. else if ***conditionalOp*** is "<" and the vector's *value is less than* ***compareVal***
3. else if ***conditionalOp*** is "==" and the vector's *value is equal to* ***compareVal***

Putting the code together (in the **toolbox.r** file):

```
counter = function(vector, compareVal, conditionalOp=">")
{
  vecLength = length(vector);  # get the length of the vector
  countVal = 0;                # initialize the count to 0
  for(val in 1:vecLength)      # go through each value in vector
  {
    if(conditionalOp == ">" && vector[val] > compareVal)
    {
      countVal = countVal + 1; # add one to the count
    }
    else if(conditionalOp == "<" && vector[val] < compareVal)
    {
      countVal = countVal + 1; # add one to the count
    }
    else if(conditionalOp == "==" && vector[val] == compareVal)
    {
      countVal = countVal + 1; # add one to the count
    }
  }
  return(countVal);    # return the count value to the caller
}
```

Extension: Checking for invalid values

9 - Calling the updated function

Make sure that ***toolbox.r*** *is saved with the updated* ***counter()!*** Then open an external script, and copy,paste, and execute the following code:

```
{
  rm(list=ls()); options(show.error.locations = TRUE);
  source("scripts/toolbox.r");

  weatherData = read.csv("data/LansingWeather3.csv");
  highTempData = weatherData[ ,"highTemp"];
  lowTempData = weatherData[ ,"lowTemp"];

  count1 = counter(vector = lowTempData, compareVal = 40, conditionalOp = "<");
  count2 = counter(vector = highTempData, compareVal = 54, conditionalOp = "==");
```

```
11    count3 = counter(vector = lowTempData, compareVal = 40);
12    count4 = counter(vector = highTempData, compareVal = 60);
13    count5 = counter(lowTempData, 38, "==");
14    count6 = counter(highTempData, 50, ">");
15  }
```

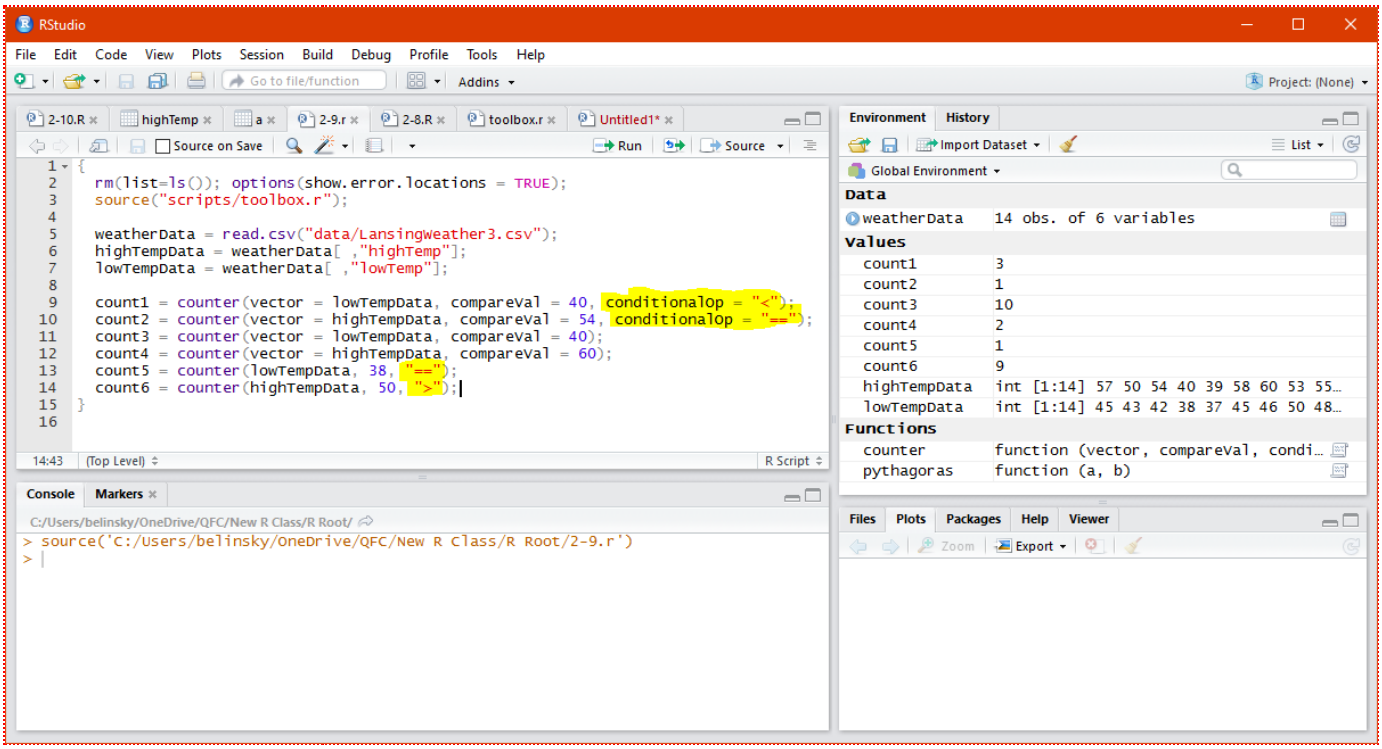

Fig 5: Comparing vectors using different conditional operators

- ***count1*** and ***count2*** explicitly assign values to the three variables (***vector***, ***compareVal***, and ***conditionalOp***)
- ***count3*** and ***count4*** use the default value for ***conditionalOp*** (which is ">")
- ***count5*** and ***count6*** pass in the values without the variable name (so, they need to be in order)

10 - Application

If you have any questions regarding this application or your Class Project, feel free to email them to the instructor here. You can attach the whole Root Folder as a zipped file.

1) Create a function that checks to see if at least one value in a vector is greater than, equal, or less than a given value
So, like ***counter()***, this function will take in three arguments:

- vector with data
- conditional operator
- comparison value

But, the function will only return ***TRUE*** or ***FALSE***

- ***TRUE*** if any value in the vector meets the condition
- ***FALSE*** if no values in the vector meet the condition

Add the function to ***toolbox.r*** and call the function from an outside script

hint: set the parameter default to FALSE

2) extend the counter function to include three more conditions:
- greater than or equal to (>=)
- less than or equal to (<=)
- not equal to (!=)

3) Challenge: Add an error condition to ***counter()*** if the caller gives an invalid conditional operator value. Note: this cannot be done by just putting an ***else*** at the end of the ***if-else-if*** structure.

*Save you script file as **app2-10.r** in the **scripts** folder of your RStudio Project for the class.*

11 - Extension: Checking for invalid values

One situation that can occur when checking for conditional operators is that the caller provided an invalid operator.

Let's say we have the situation:
1. the valid operators are: **<, >, ==**
2. the caller gave the operator: **@**

First, we want to create a conditional statement that checks to see if the operator is invalid. There are two ways to do this. The first is to check to see if the operator is valid and invert the conditional statement using the (**!**) operator:

```
if(  !(conditionalOp == ">" || conditionalOp == "<" || conditionalOp != "==")  )
{
  cat("Sorry, ", conditionalOp, " is not a valid operator");
}
```

The conditional statement first checks the inner parenthesis. The ***or*** operator (**||**) means inner parenthesis will be ***TRUE*** if ***conditionalOp*** is any of the three operators given. The ***not*** operator (**!**) inverts that condition to **FALSE**. This means the code attached to the ***if()*** will be executed only if ***conditionalOp*** is ***not*** any of the three.

There is a second way we can write the conditional statement using the ***and*** operator (**&&**):

```
if(conditionalOp != ">" && conditionalOp != "<" && conditionalOp != "==")
{
  cat("Sorry, ", conditionalOp, " is not a valid operator");
}
```

The ***&&*** operator says that all three conditions must be ***TRUE*** in order for the conditional statement to be ***TRUE***. The three conditions all involve the ***not equal to*** operator. So, the conditional statement is ***TRUE*** if ***conditionalOp is not equal*** to all the three operators given.

In mathematical parlance, the two conditional statements above are *contrapositives*. In other words, two things were reversed and the result stays the same.

We will often first check for invalid values in an ***if-else*** structure. This is because there is no point in making any of the other check if the ***conditionOp*** is not valid.

```
counter = function(vector, compareVal, conditionalOp=">")
{
  vecLength = length(vector);  # get the length of the vector
  countVal = 0;                # initialize the count to 0
  for(val in 1:vecLength)      # go through each value in vector
  {
    if(conditionalOp != ">" && conditionalOp != "<" && conditionalOp != "==")
    {
      cat("Sorry, ", conditionalOp, " is not a valid operator");
    }
    else if(conditionalOp == ">" && vector[val] > compareVal)
    {
      countVal = countVal + 1; # add one to the count
    }
    else if(conditionalOp == "<" && vector[val] < compareVal)
    {
      countVal = countVal + 1; # add one to the count
    }
    else if(conditionalOp == "==" && vector[val] == compareVal)
    {
      countVal = countVal + 1; # add one to the count
    }
  }
  return(countVal);    # return the count value to the caller
}
```

12 - Trap: Putting parameters values in the wrong order

```
{
  rm(list=ls()); options(show.error.locations = TRUE);
  source("scripts/toolbox.r");   # load functions from toolbox.r file

  weatherData = read.csv("data/LansingWeather3.csv");
  highTempData = weatherData[ ,"highTemp"];
  lowTempData = weatherData[ ,"lowTemp"];

  count1 = counter(vector = highTempData, compareVal = 50);
  count2 = counter(vector = highTempData, compareVal = 60);
  count3 = counter(40, lowTempData);  # problem here!
}
```

Unfortunately, executing this script causes no errors. ***counter()*** happily tries to compare the one-value vector (40) to the value of ***lowTempData***. And the answer it gets for ***count3*** is ***0***. This is because ***counter()*** can only use one comparison value -- so it uses the first value in the ***lowTempData***, which is ***45***. **40** is not greater than **45**, so the answer is **0**. If the first value in ***lowTempData*** is less than **40**, then ***count3*** will be **1**.

Either way, this is not what you want. So, be careful when you don't use names to keep the parameter values ordered correctly -- or be safe and use parameter names.