02-10: Functions 2

1 - Purpose

- create a separate file to hold function
- create a counter function
- use functions from separate file

1.1 - Student-Instructor meeting

This is the last lesson in Unit 1, which means it is time to schedule a 20-30 minute meeting with the instructor. Please email the instructor at belinsky@msu.edu and provide some times that you are available.

2 - Concepts

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 the plots. Unlike **pythagoras()**, which is a function that exists inside your script file, all of the plotting functions exist outside the script file -- and the plotting functions 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 used by an external script.

4 - Creating an external file for functions

Functions are like tools in that they 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()** (below) function to the new script file
3) Save the file as   **toolbox.r** inside the **script** folder.

```
1   pythagoras = function(a,b)
2   {
3      c = (a^2 + b^2)^(1/2);
4      return(c);
5   }
```

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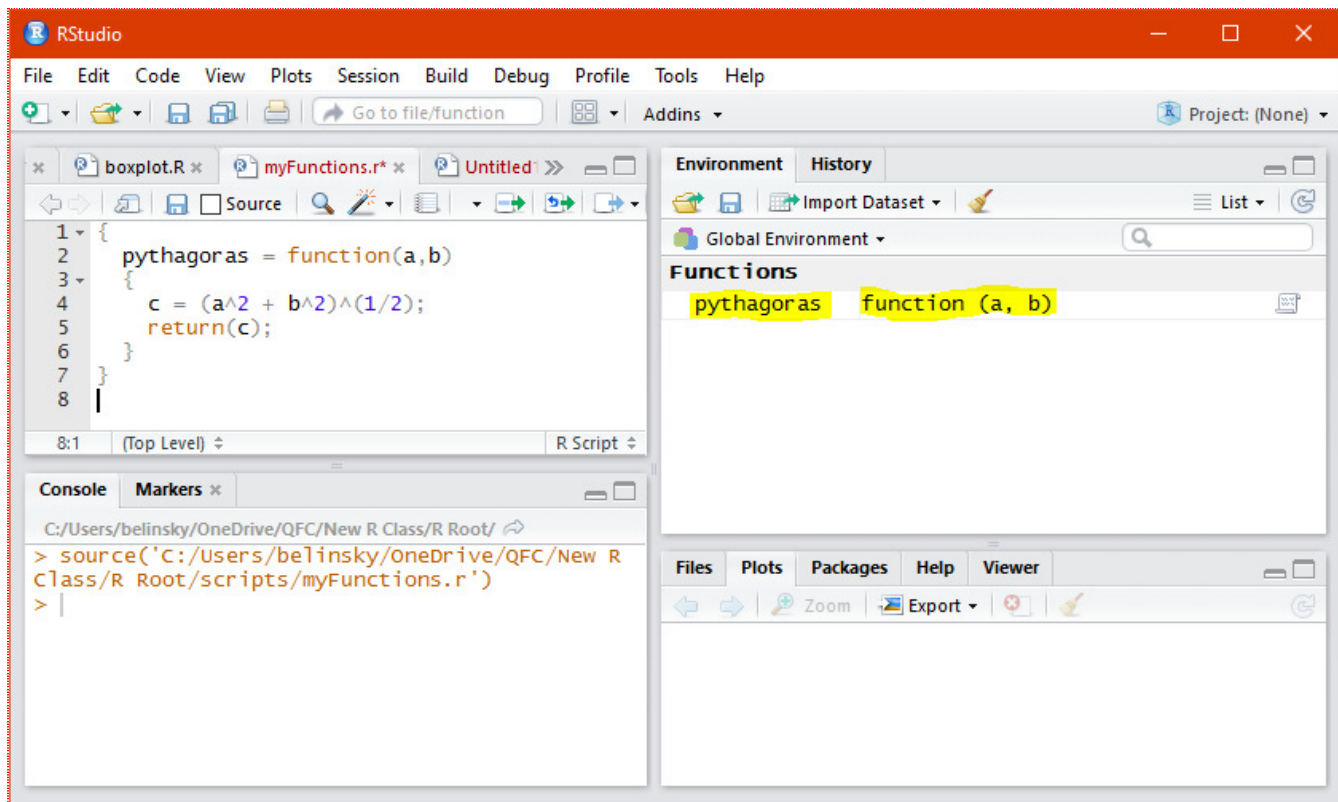
*Fig 1: Adding **pythagoras()** to a functions script*

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**. This is done using the **source()** function and the parameter is the location and name of the script file you want to include.

This code tell R to include everything in the **toolbox.r** file (which is in the **scripts** folder) in your current script file.

```
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 5) *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  }
```
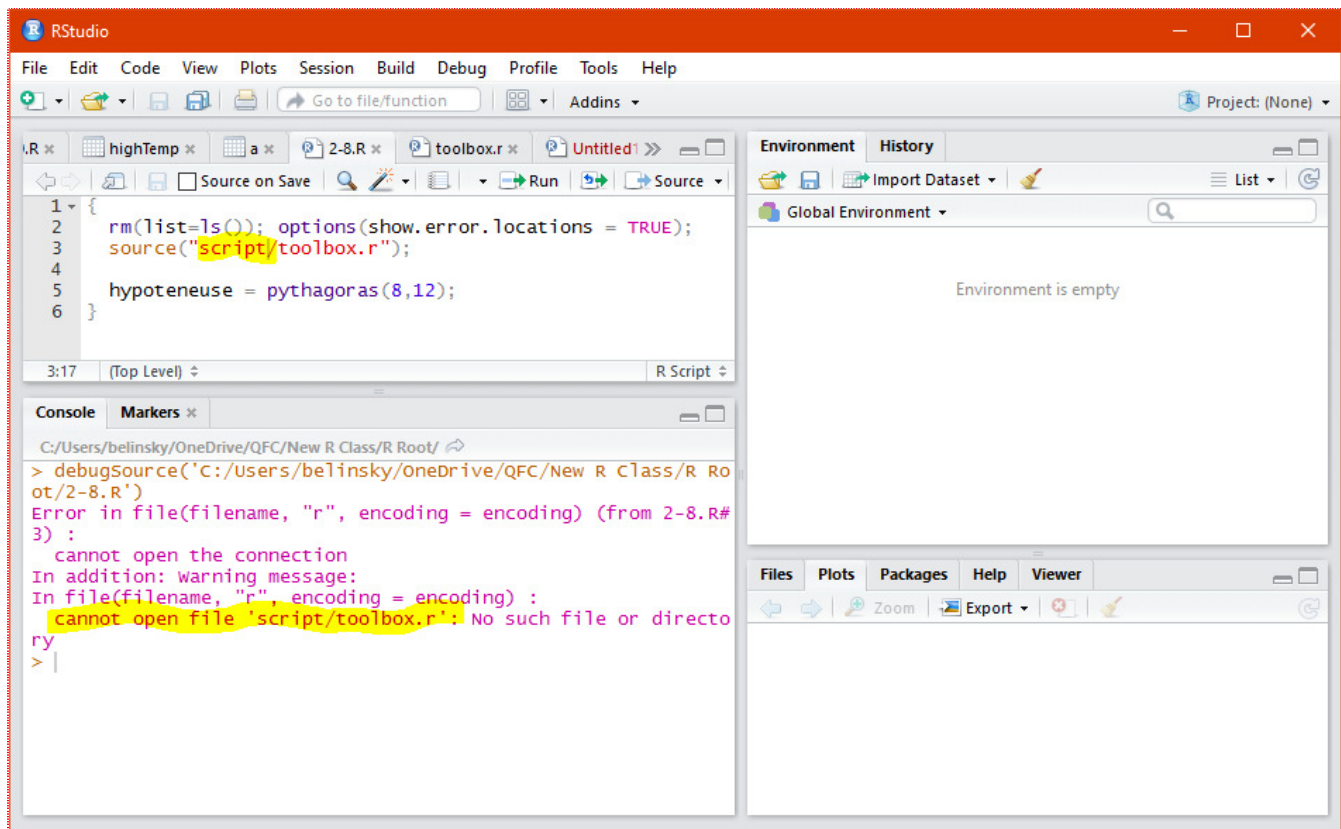
Fig 3: Error calling the source file

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 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

Initially, **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 looking through the vector

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

```
1  {
2      counter = function(vector, compareval)
3      {
4
```

```
5    }
6  }
```

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  {
2     counter = function(vector, compareVal)
3     {
4       vecLength = length(vector);  # get the length of the vector
5       for(val in 1:vecLength)      # go through each value in vector
6       {
7         # counting script will go here
8       }
9     }
10 }
```

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

```
 8        if(vector[val] > compareVal)
 9          {
10            # add one to the count
11          }
12        }
13    }
14 }
```

6.4 - Creating a count value

The function is sill not doing 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 assigned **countVal** the value of 0.

```
1    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**

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

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

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

Putting all the code together:

```
 1 {
 2   counter = function(vector, compareVal)
 3   {
 4     vecLength = length(vector);  # get the length of the vector
 5     countVal = 0;               # initialize the count to 0
 6     for(val in 1:vecLength)     # go through each value in vector
 7       {
 8         # check if the vector value is greater than the compareVal
 9         if(vector[val] > compareVal)
10         {
11           countVal = countVal + 1; # add one to the count
12         }
13       }
14   }
15 }
```

6.5 - Returning the count to the user

The caller is only interested in the answer so the last step is to return the final state of **countVal** to the caller using the **return()** function.

```
 1  {
 2    counter = function(vector, compareVal)
 3    {
 4      vecLength = length(vector);   # get the length of the vector
 5      countVal = 0;                 # initialize the count to 0
 6      for(val in 1:vecLength)       # go through each value in vector
 7      {
 8        if(vector[val] > compareVal)
 9        {
10          countVal = countVal + 1; # add one to the count
11        }
12      }
13      return(countVal);   # return the count value to the caller
14    }
15  }
```

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

```
 1  {
 2    pythagoras = function(a,b)
 3    {
 4        c = (a^2 + b^2)^(1/2);
 5        return(c);
 6    }
 7    counter = function(vector, compareVal)
 8    {
 9      vecLength = length(vector);   # get the length of the vector
10      countVal = 0;                 # initialize the count to 0
11      for(val in 1:vecLength)       # go through each value in vector
12      {
13        if(vector[val] > compareVal)
14        {
15          countVal = countVal + 1; # add one to the count
16        }
17      }
18      return(countVal);   # return the count value to the caller
19    }
20  }
```

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**).

```
 1  {
 2    rm(list=ls()); options(show.error.locations = TRUE);
 3    source("scripts/toolbox.r");
 4
 5    weatherData = read.csv("data/LansingWeather3.csv");
 6    highTempData = weatherData[ ,"highTemp"];
 7    lowTempData = weatherData[ ,"lowTemp"];
 8
 9    count1 = counter(vector = highTempData, compareVal = 50);
10    count2 = counter(vector = highTempData, compareVal = 60);
11    count3 = counter(lowTempData, 40);
12  }
```

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 a different 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.

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** 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:

```
1  {
2    counter = function(vector, compareVal, conditionalOp=">")
3    {
4      vecLength = length(vector);  # get the length of the vector
5      countVal = 0;               # initialize the count to 0
6      for(val in 1:vecLength)     # go through each value in vector
7      {
8        if(conditionalOp == ">" && vector[val] > compareVal)
9        {
10          countVal = countVal + 1; # add one to the count
11        }
12        else if(conditionalOp == "<" && vector[val] < compareVal)
13        {
14          countVal = countVal + 1; # add one to the count
15        }
16        else if(conditionalOp == "==" && vector[val] == compareVal)
```

```
17        {
18           countVal = countVal + 1; # add one to the count
19        }
20     }
21     return(countVal);      # return the count value to the caller
22   }
23 }
```

9 - Calling the updated function

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

```
1 {
2   rm(list=ls()); options(show.error.locations = TRUE);
3   source("scripts/toolbox.r");
4
5   weatherData = read.csv("data/LansingWeather3.csv");
6   highTempData = weatherData[ ,"highTemp"];
7   lowTempData = weatherData[ ,"lowTemp"];
8
9   count1 = counter(vector = lowTempData, compareVal = 40, conditionalOp = "<");
10   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 }
```
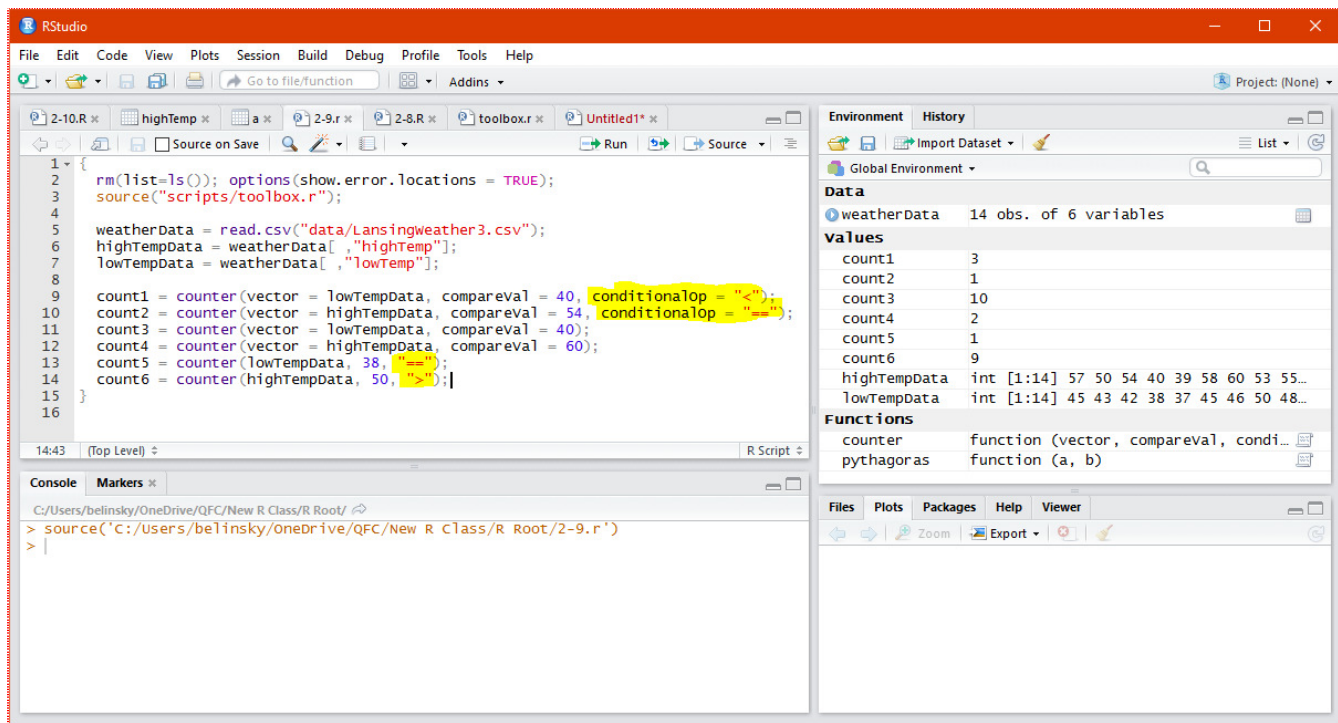
Fig 5: Comparing the vector using multiple conditional operators

- **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

1) Create a function that checks to see if any 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) 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. The reason will be explained in

the answer but think about the condition that would cause the *else* to be executed.