01-04: Debugging

1 - Purpose

- Set up RStudio diagnostics to help find errors in your code
- Find syntax errors in your script using the Console Window
- Find logic errors in your script using the Environment Window
- Use breakpoints to pause you script

2 - Concepts

3 - RStudio Diagnostics

RStudio has real-time debugging (called *diagnostics*) that you can turn on at: ***Tools*** -> ***Global Options*** -> ***Code*** -> ***Diagnostics.***

You should check everything on the page (*Fig 1*) *except* "Provide R style diagnostics", which is not particularly helpful

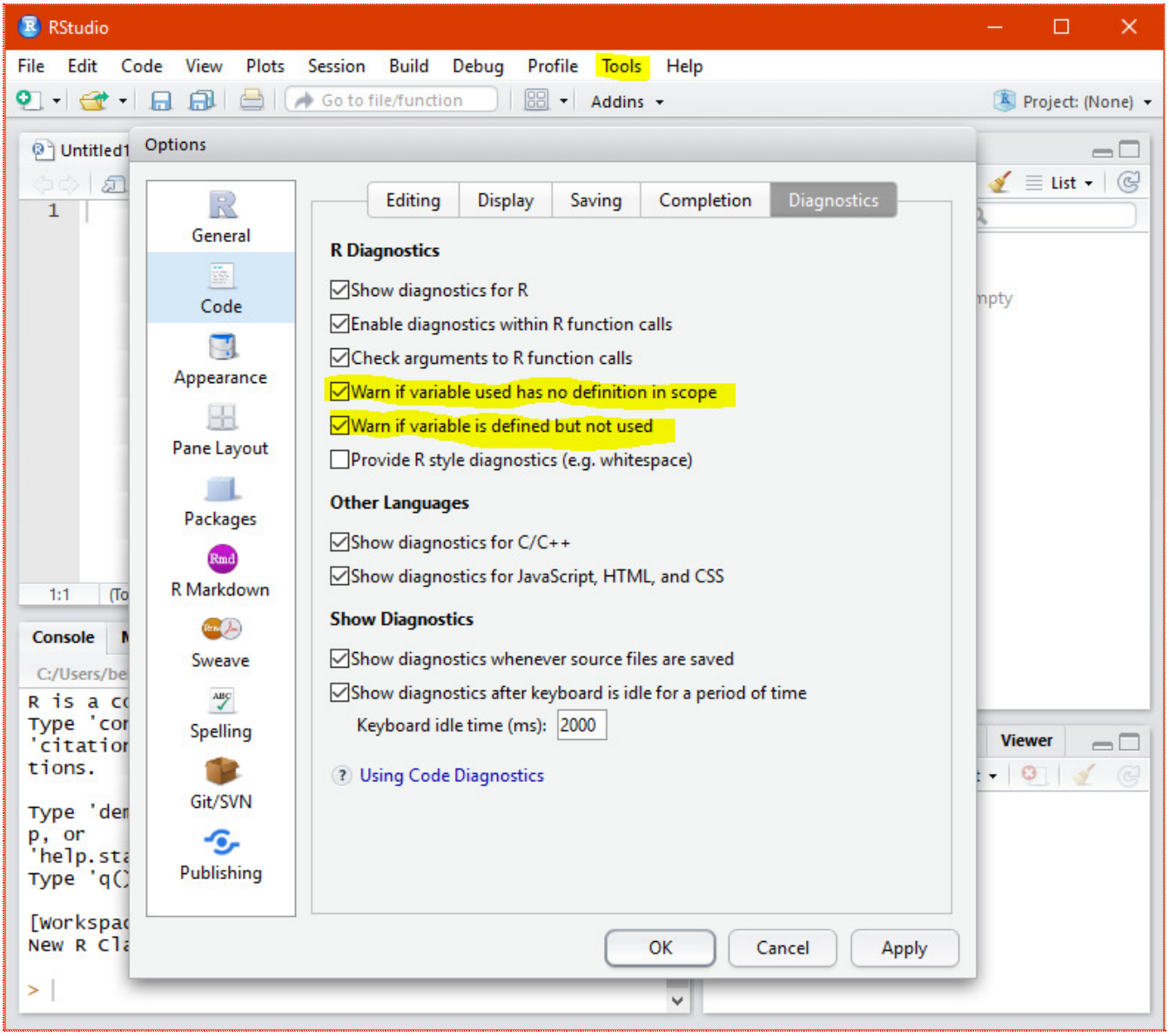

Fig 1: RStudio diagnostic settings page.

RStudio diagnostics works in the background checking your script for certain issue. RStudio diagnostics will not run properly unless:
1) You have saved your script file (i.e., your script does not still have the name ***Untitled1***).
2) You use semicolons (***;***) to end your code statements.
3) You have curly brackets (***{ }***) at the start and end of your script.
Note: Most R script you will run across do not use semicolons or put curly brackets at the beginning and end.

The two main issues that diagnostics will warn you about are:
1) You use variables that have not been declared (i.e., no variable exists with that name)
2) You are missing a matching start or end quote, bracket, or parenthesis.

3.1 - Yellow Triangle: variable used that is not declared

A yellow triangle (*Fig 2*) will be placed on any line that is using a variable that has not previously been declared:

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

  time = 10;
  velocity = distance/time;  # distance is undeclared variable
}
```

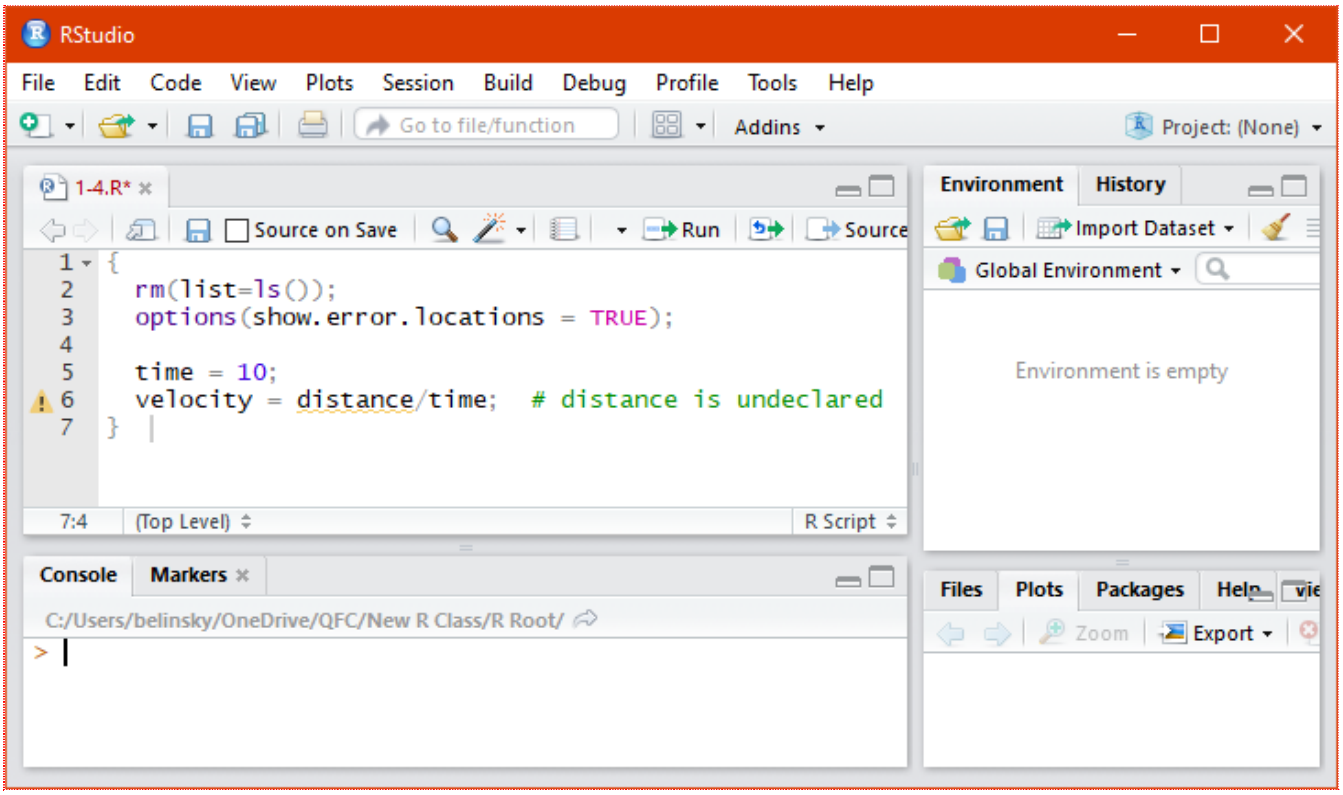

Fig 2: A yellow triangle appears where there is a undeclared variable.

The often happens because a variable name is misspelled. In the following example, velocity is initially declared with a lowercase ***v***, but later it is spelled with an uppercase ***V***. Remember, case counts and changing the case means you have changed the name.

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

  velocity = 10;
  mass = 3;
  kineticEnergy = (1/2)*mass*Velocity^2;  # velocity is "spelled" wrong
}
```

If you hover over the yellow triangle (*Fig 3*), a message appears that Velocity is not a valid variable (symbol) -- and then offers a suggestion for what you might have meant.

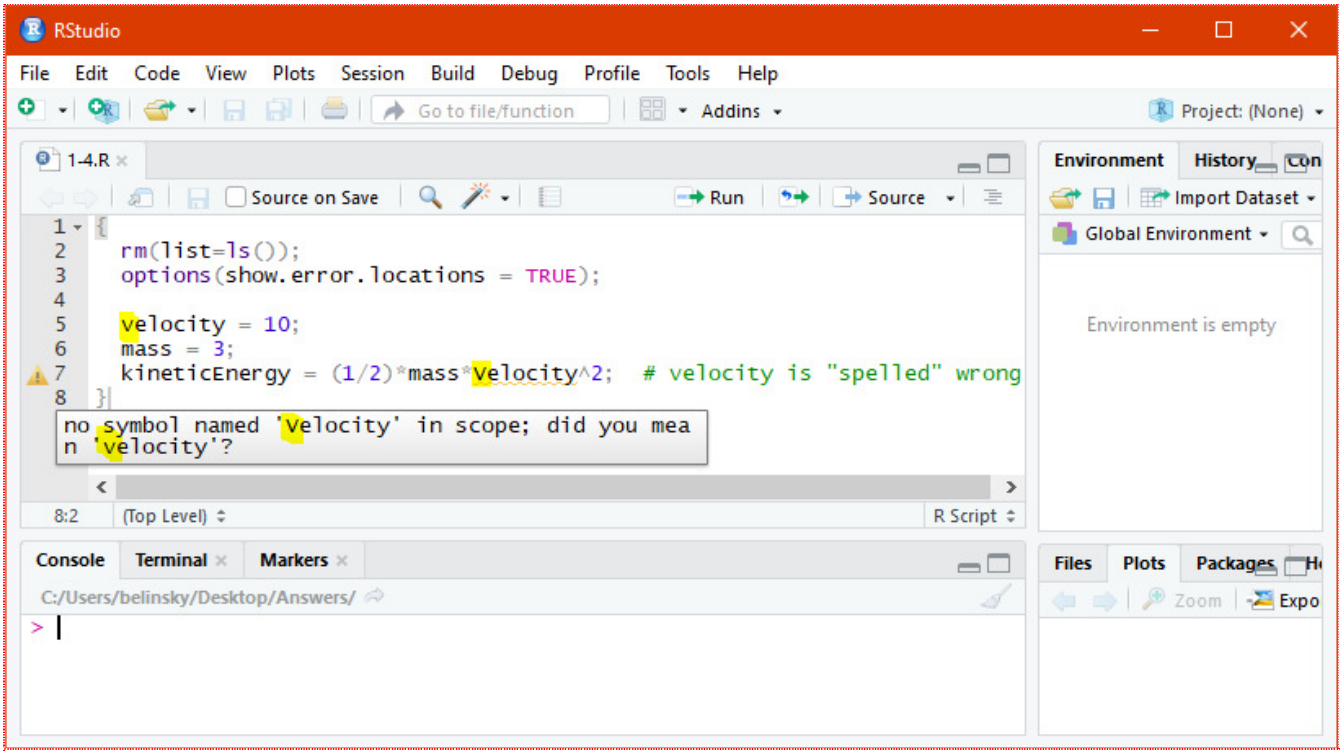

Fig 3: Hovering over the yellow triangle gives some helpful information about the error.

3.2 - Red X: Missing a matching quote, parenthesis, or bracket

A red x almost always means that there is a quote, parenthesis, or bracket missing. When this happens, you will often see multiple red x's (*Fig 4*) -- RStudio diagnostics is not quite robust enough to figure out the exact line. It is best to start with the first line that has a red x and move down until you fix the issue.

The following script has a missing end parenthesis on line 2:

```
1 {
2   rm(list=ls();  # missing end parenthesis
3   options(show.error.locations = TRUE);
4
5   distance = 10;
6   time = 5;
7   velocity = distance/time;
8 }
```

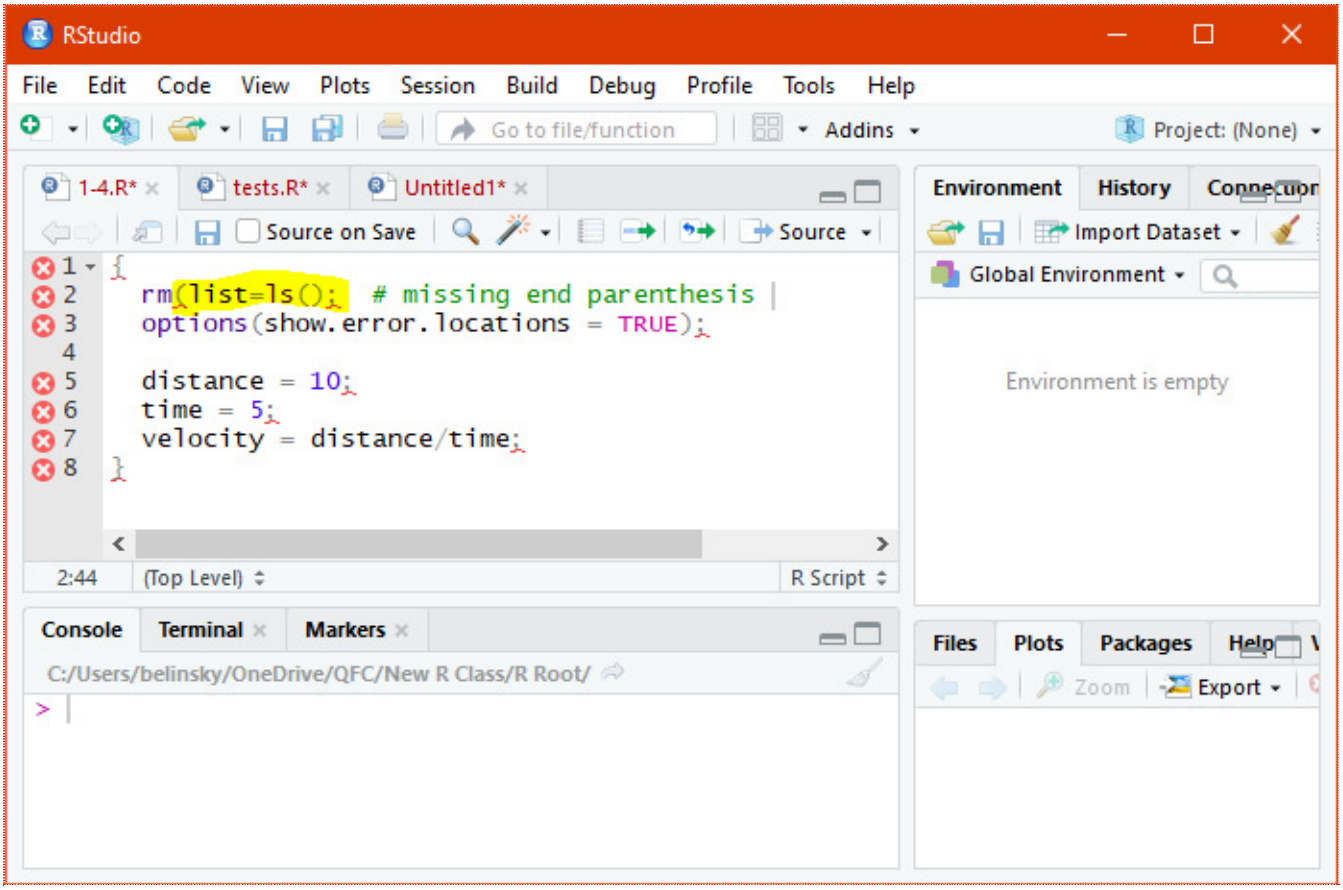

Fig 4: Missing end parenthesis on line two causing RStudio to place red x's on every line.

Here is another example with a missing parenthesis

```
1 {
2   rm(list=ls());
3    options(show.error.locations = TRUE);
4
5    velocity = 10;
6    mass = 3;
7    kineticEnergy = 1/2)*mass*velocity^2;  # missing parenthesis
8 }
```

Hovering over the red X (*Fig 5*) says that:
1) the end parenthesis was unexpected (because there was no start parenthesis)
2) the * was unexpected (this is a cascade error that gets fixed when you correct the first issue)

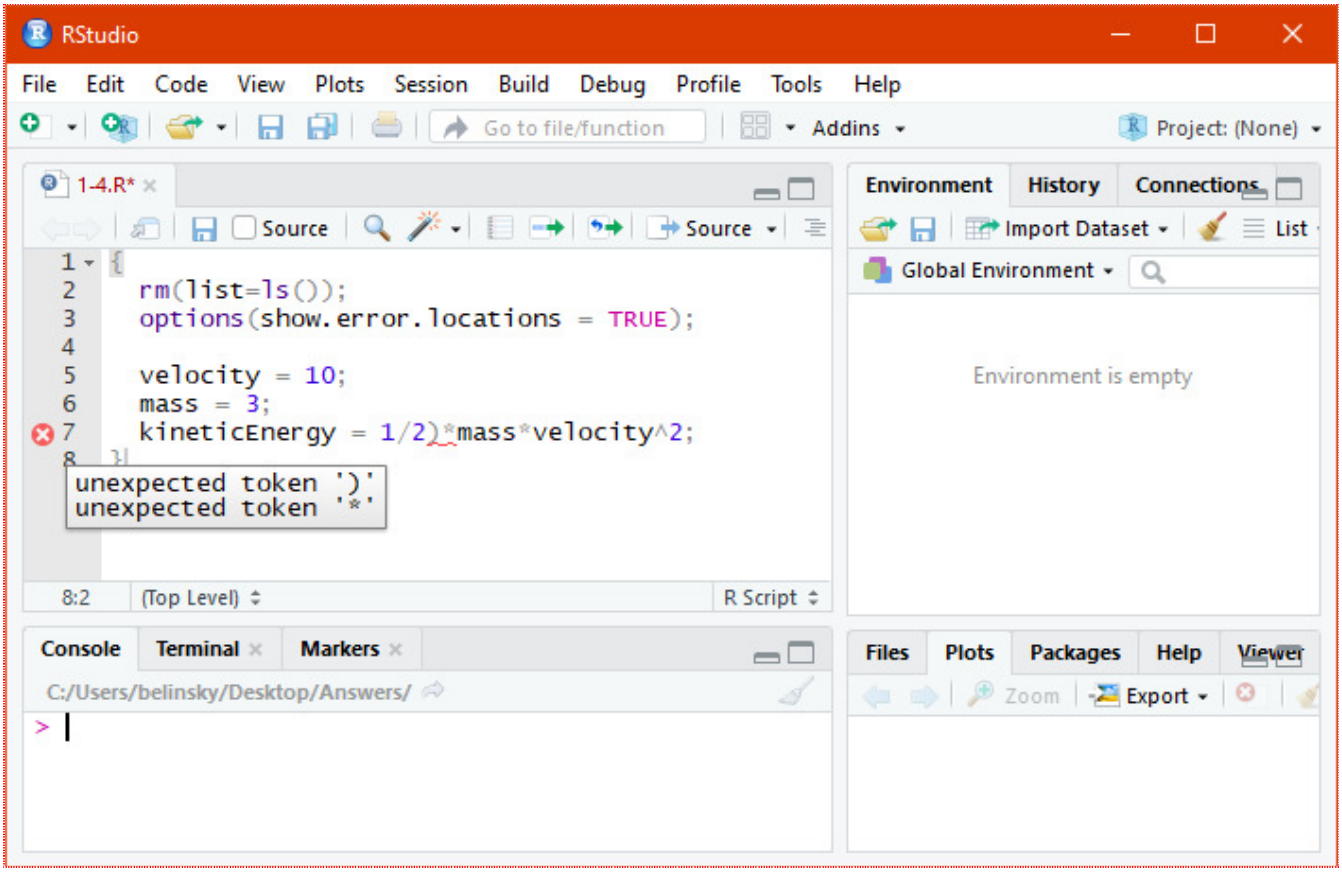

Fig 5: Hovering over the red X which indicated the missing parenthesis.

4 - Debugging Errors

In programming tiny mistakes can easily propagate especially as scripts get longer and more complicated -- and these mistakes can cause huge problems if they are not resolved. *Debugging* helps you ferret out these tiny errors making debugging a vital part of maintaining a larger script.

In RStudio, there are three levels of debugging:
1) Using the Console Window to debug *syntax errors*
2) Using the Environment Window to debug *logic errors*
3) Using *breakpoints* to examine the script in the middle of execution

4.1 - Using console to debugging syntax errors

The simplest form of debugging is to use the Console Window to check for syntax errors in your code. Syntax errors prevent the script from moving forward because the script is being told to do something it cannot do. *R will stop execution of the script at the first syntax error it finds* -- so all code beyond that point will not be executed. The mistakes indicated by the yellow triangle and the red X are examples of syntax errors.

Examples of syntax errors include:
1) Using an undeclared variable
2) Forgetting to use the multiplication operation
3) Forgetting to end a parenthesis

4.1.1 - Using an undeclared variable

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

  finalDistance = 10;
  initDistance = 50;
  finalTime = 20;
  initTime = 15;
  velocity = (finalDistance - initDistance) / (finaltime - initialTime);
  mass = 100;
  kineticEnergy = (1/2)*mass*velocity^2;
}
```

The Console Window error is: "*object 'initialTime' not found*" because ***initialTime*** has not been declared as a variable (i.e., it does not exist in your script).

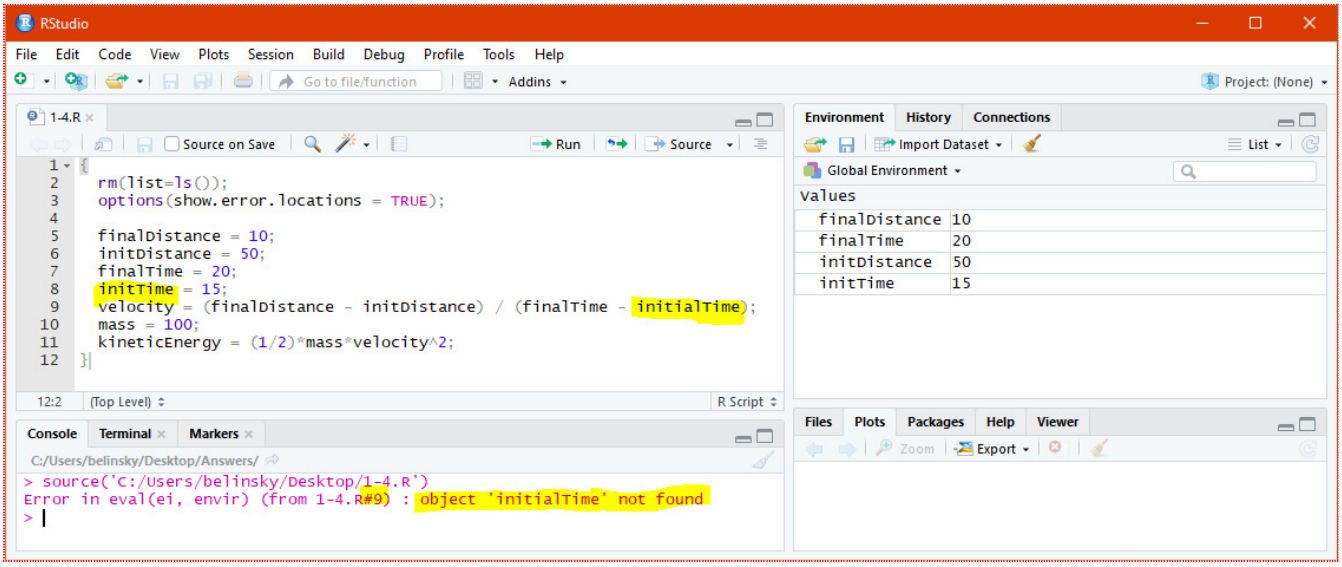

Fig 6: Object not found error because of an undeclared variable

4.1.2 - Forgetting to use the multiplication operation

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

  finalDistance = 10;
  initDistance = 50;
  finalTime = 20;
  initTime = 15;
  velocity = (finalDistance - initDistance) / (finaltime - initTime);
  mass = 100;
  kineticEnergy = (1/2)mass*velocity^2;
```

```
12 }
```

Console Window error: "*unexpected symbol*" because the script is looking for an operator after ***(1/2)*** and gets a variable, ***mass***, instead

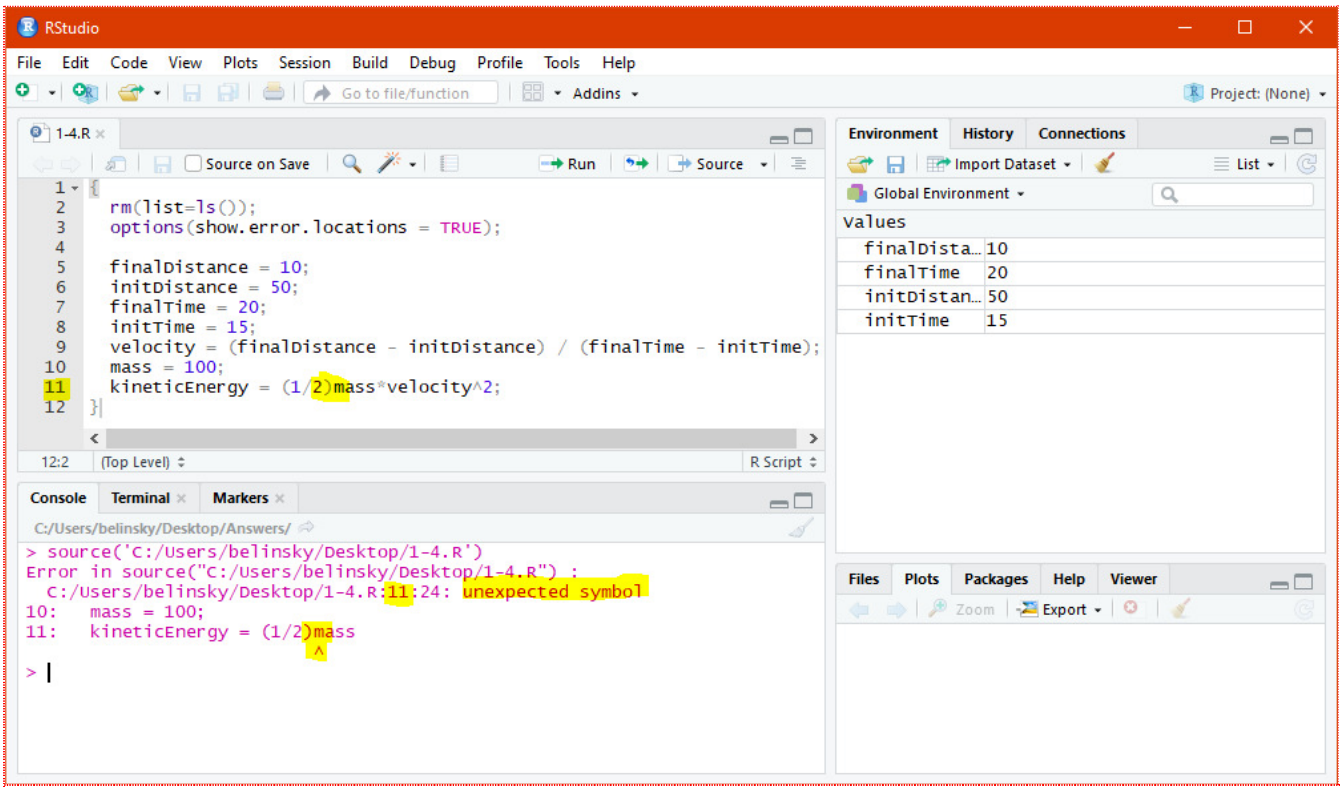

Fig 7: Unexpected symbol error because of the missing multiplication symbol

4.1.3 - Forgetting to end a parenthesis

```
1  {
2    rm(list=ls());
3    options(show.error.locations = TRUE);
4
5    finalDistance = 10;
6    initDistance = 50;
7    finalTime = 20;
8    initTime = 15;
9    velocity = (finalDistance - initDistance) / (finalTime - initTime;
10   mass = 100;
11   kineticEnergy = (1/2)*mass*velocity^2;
12 }
```

The Console Window (*Fig 8*) puts a caret (**^**) under the semicolon (***;***) and the error is "*unexpected ';'*" . R was expecting the closing parenthesis to come before the semicolon, which indicates the end of the statement.

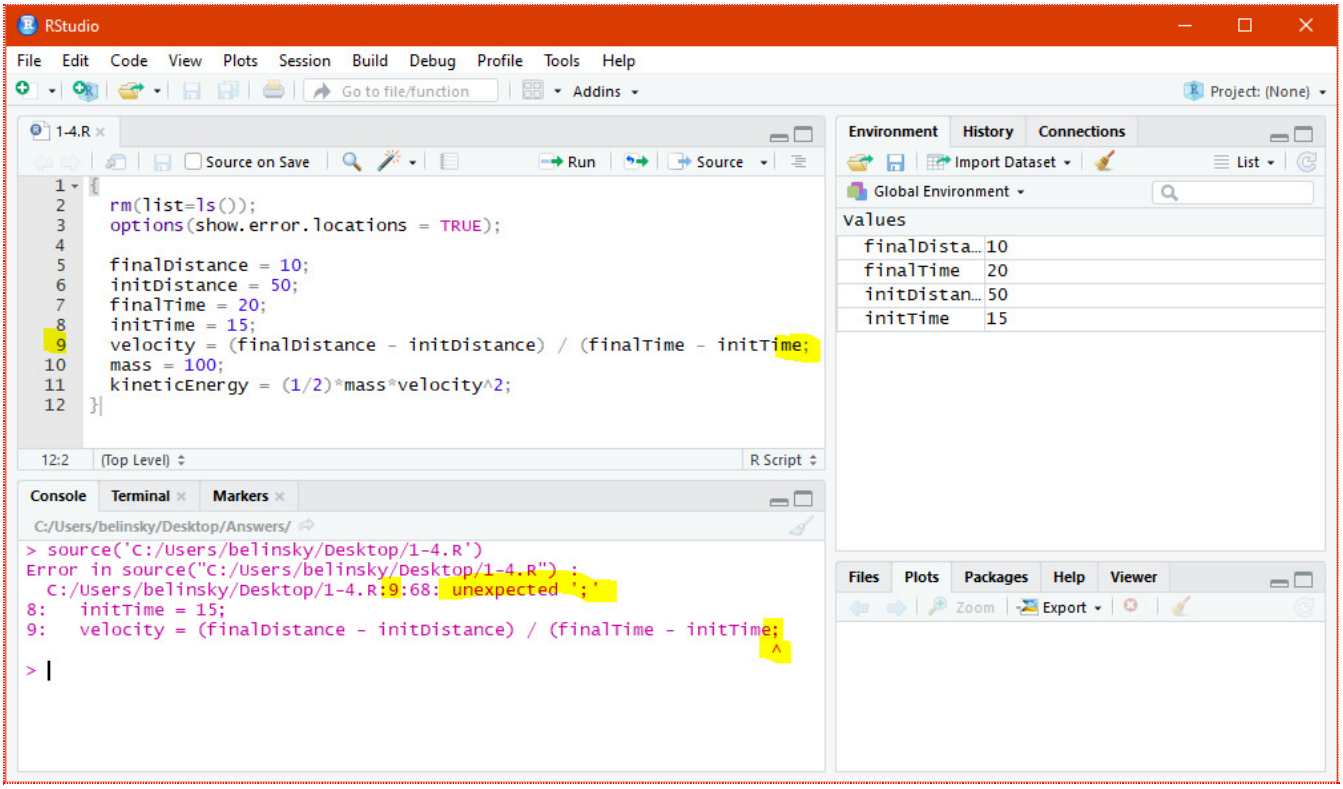

Fig 8: Unexpected ";" error because the closing parenthesis is missing

This type of error is common as there are many characters used in scripting that require matching opening and closing characters:

- square brackets: ***[ ]***
- curly brackets: ***{ }***
- parentheses: ***( )***
- double quotes: ***" "***
- single quotes: ***' '***

4.2 - The Environment Window: debugging logic errors

Logic errors are any error that results in the script running differently than the user expected. Logic errors are different, and more cryptic, than syntax errors because a script can still run with logic errors but the script will exhibit unexpected behaviors. These errors are often found by looking at the Environment Window.

We will be talking about logic error in future lesson when the coding gets more advanced. For now, the most common logic error is going to be parentheses need to be added or parenthesis placed wrong.

4.2.1 - Parentheses need to be added

```
1 {
2   rm(list=ls());
3   options(show.error.locations = TRUE);
4
5   finalDistance = 10;
6   initDistance = 50;
```

```
  finalTime = 20;
  initTime = 15;
  velocity = (finalDistance - initDistance) / finalTime - initTime;
  mass = 100;
  kineticEnergy = (1/2)*mass*velocity^2;
}
```

In the Environment Window, the values for ***velocity*** is **-17** (it should be **-8**) because you are subtracting ***initTime*** from (***finalDistance*** - ***initDistance***) / ***finalTime.***

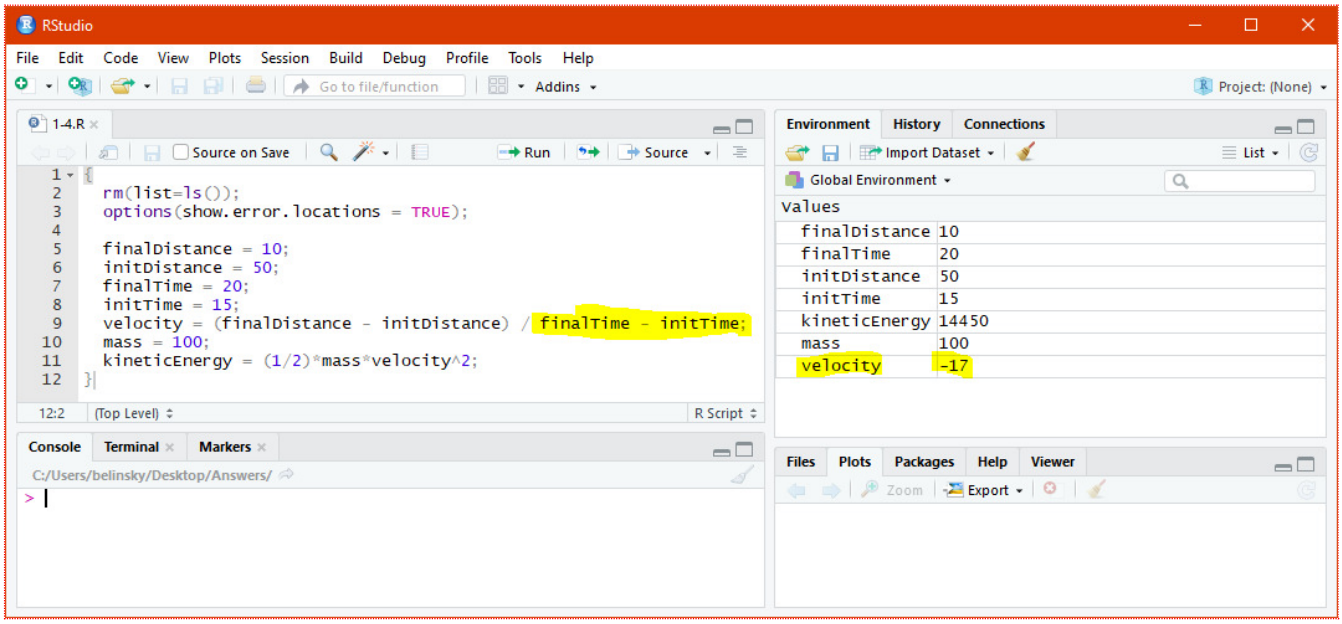

Fig 9: Logic error: parentheses are needed around ***(finalTime - initTime)***

4.2.2 - Parentheses placed wrong

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

  finalDistance = 10;
  initDistance = 50;
  finalTime = 20;
  initTime = 15;
  velocity = (finalDistance - initDistance) / (finalTime - initTime);
  mass = 100;
  kineticEnergy = (1/2)*(mass*velocity)^2;
}
```

kineticEnergy is **320000** (should be **3200**) because you are squaring ***mass*velocity*** instead of squaring just ***velocity***.

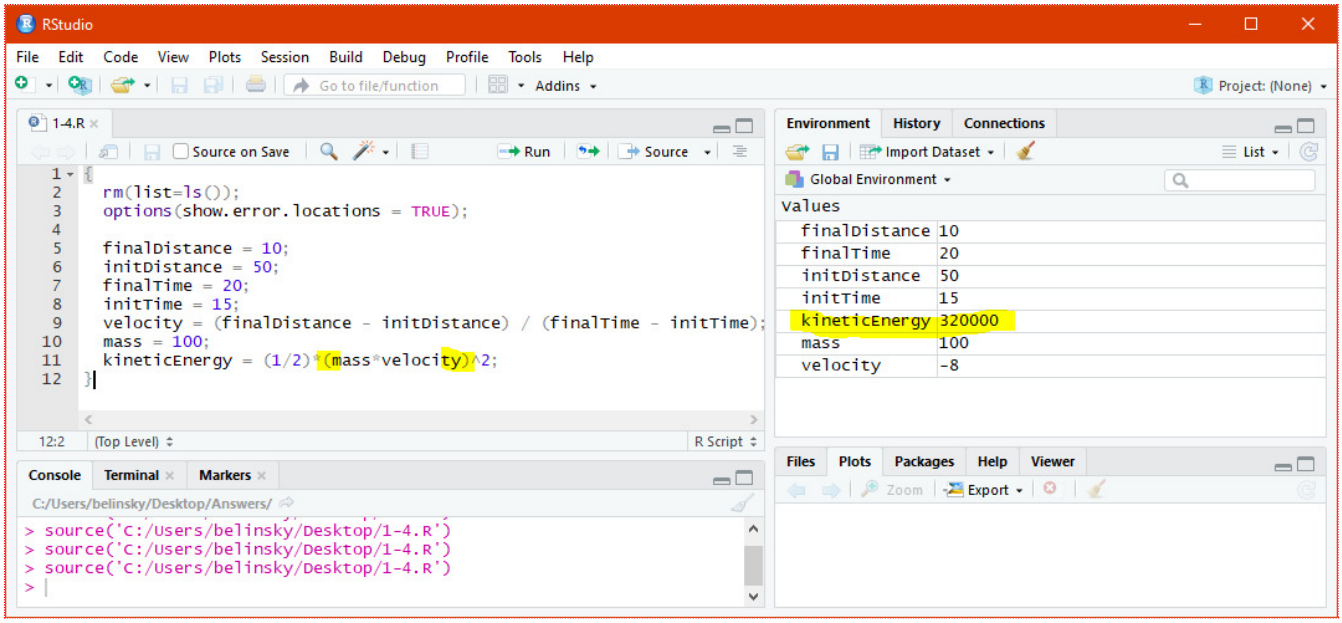

*Fig 10: Logic error: parentheses are misplaced around **mass*velocity**.*

This problem (*Fig 10*) can either be fixed by getting rid of the parenthesis:

```
11  kineticEnergy = (1/2)*mass*velocity^2;
```

Or by putting only velocity in parenthesis:

```
11  kineticEnergy = (1/2)*mass*(velocity)^2;
```

5 - Using breakpoints

So far all of the debugging in this lesson has happened after the whole script has be executed or while it is being developed. The real power of debugging is the ability to stop the script in the middle of its execution and observe all the variables at this point. RStudio gives you the ability to stop execution at any point in your script using *breakpoints*.

A *breakpoint* is an instruction that says "pause the script here". You can specify which line to stop at by clicking to the left of the line number. Click once and a red dot appears to the left of the line number. Click again and it disappears. You can put a breakpoint on any line *that has executable code* -- so you cannot put breakpoints on comment lines or blank lines.

Note: *You must save your script before using breakpoints.* Copy the script below, save it, put a breakpoint at line 10, and execute the script:

```
1  {
2    rm(list=ls());
3     options(show.error.locations = TRUE);
4
5     finalDistance = 10;
6     initDistance = 50;
7     finalTime = 20;
8     initTime = 15;
9
10    velocity = (finalDistance - initDistance) / (finalTime - initTime);
```

```
11
12   # change initDistance and recalculate velocity
13    initDistance = -10;
14    velocity2 = (finalDistance - initDistance) / (finalTime - initTime);
15 }
```

A green arrow now appears to the left of line 10 (*Fig 11*). The arrow indicates that code execution has been stopped at line10 (and line 10 *has not been executed*). Notice that all variables except ***velocity*** and ***velocity2*** are in the Environment Window. *This is because **velocity** and **velocity2** has not yet been declared.*

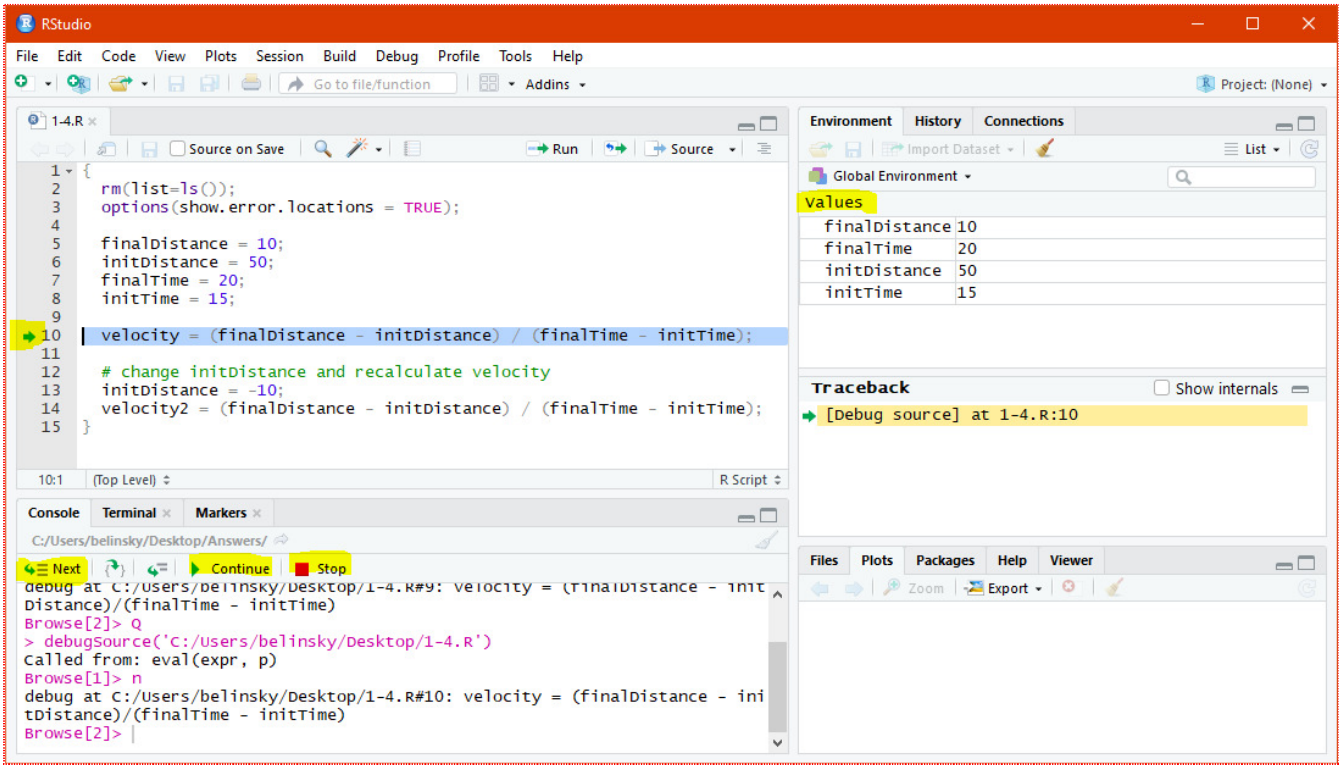

Fig 11: Executing a script that has a breakpoint.

The Console Windows has 5 new buttons at the top:

- ***Next***: executes only the current line of code and moves the breakpoint to the next line with code
- ***Continue***: runs the rest of the script
- ***Stop***: ends the script (no more lines are executed)
- The other two buttons (between ***next*** and ***continue***) are used when you are dealing with functions or looping code.

With your script stopped at line 10:

1) Click the ***next*** button.

- This will execute line 10 and move the breakpoint to the *next executable line of code*, which is line 13.
- ***velocity***, which was assigned a value in line 10, now appears with a value of ***-8*** in the Environment Window (*Fig 12*).

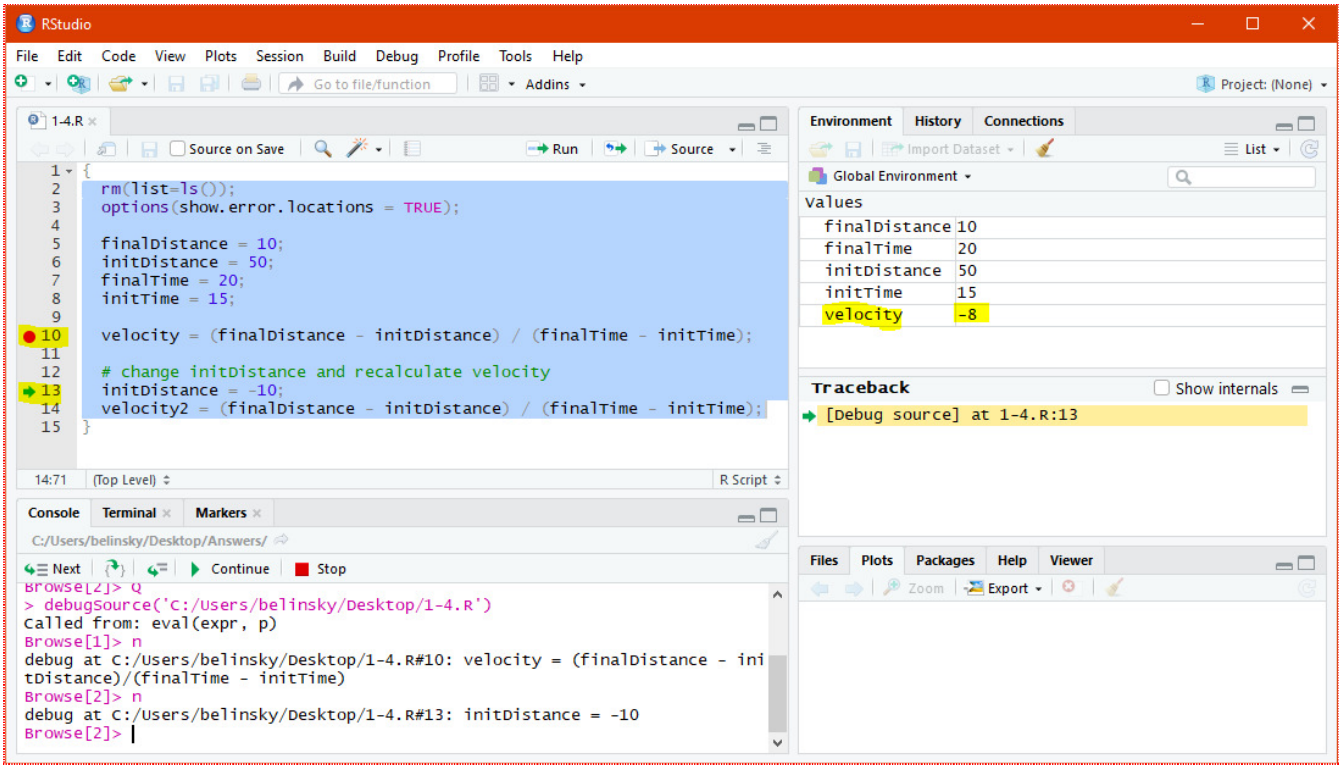

Fig 12: Executing one line using the next button

2) Click the ***next*** button again:

- This will execute line 13 and move the breakpoint to the next executable line of code, which is line 14.
- ***initDistance*** has been reassigned the value of ***-10*** (*Fig 13*) -- remember variables can change values in a script by simply assigning the variable a new value

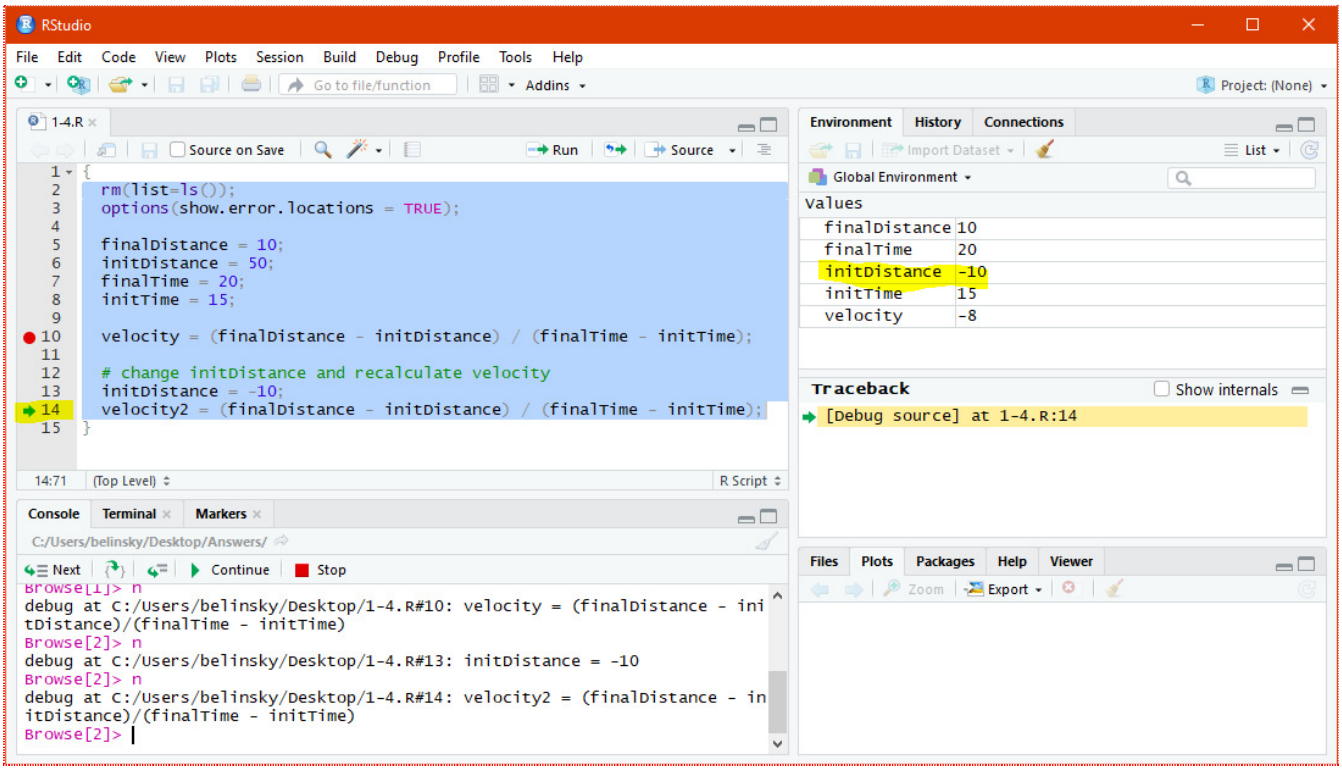

Fig 13: Variable gets a new value assigned to it

3) Click the ***next*** button a third time:

- This will execute line 14 and end the script.
- ***velocity2*** now appears and has a value in the Environment Window (*Fig 14*).

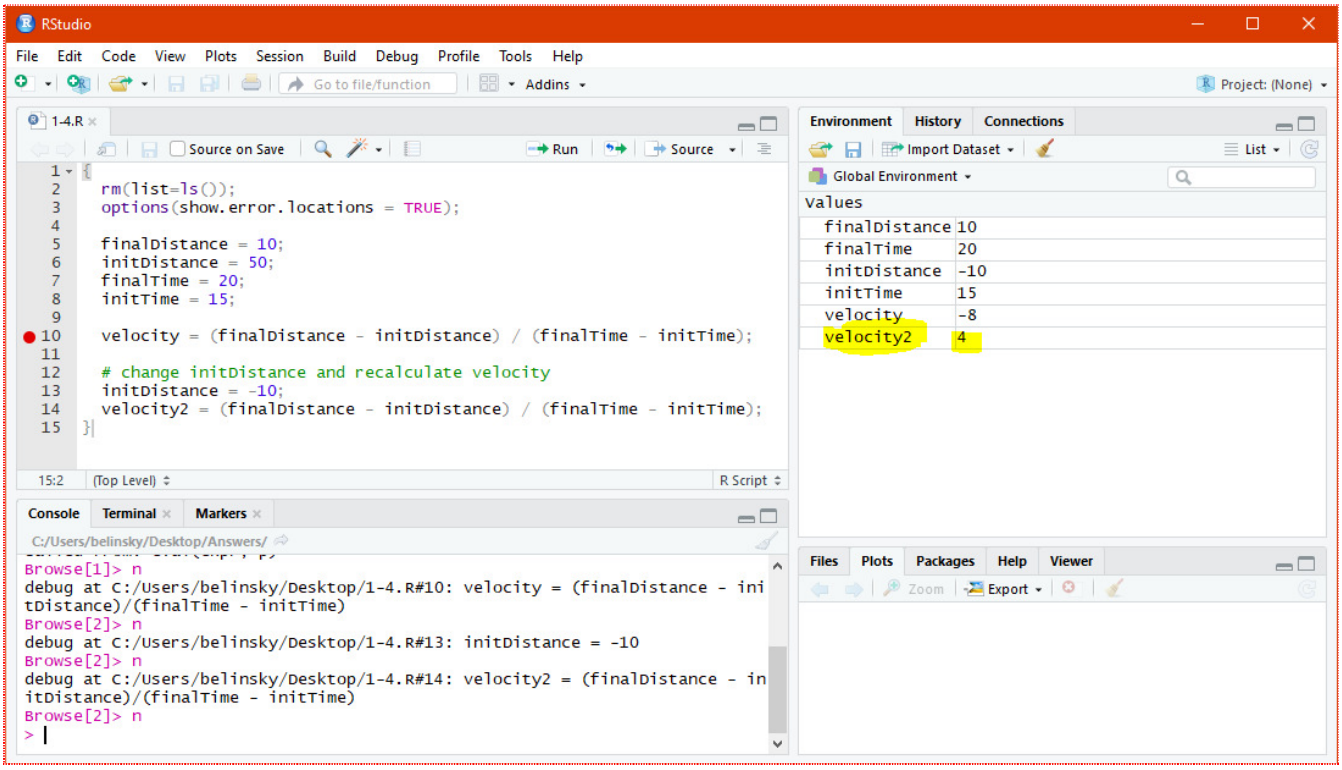

Fig 14: Debugger reached the end of the script

The use of breakpoints is vital when developing large scripts. Large scripts are never developed from top to bottom, often they are edited in the middle. If you want to debug the area of the script you are editing, you should use breakpoints so that you can see the environment as it exists at the point of the script you want to edit.

6 - Application

Find the errors (both syntax and logic) in the following code:

```
1  {
2    rm(list=ls()) # clean out the environment
3     options(show.error.locations = TRUE) # give line numbers on errors
4
5     fish1Weight = 45
6     fish2Weight = 64
7     fish3Weight = 50
8     fish4Weight = 58
9     fish5Weight = 49
10
11    # solve for the mean fish weight
12    meanWeight = fish1Weight + fish2Weight + Fish3weight + fish4Weight + fish5Weight
       /5
```

```
13
14    # solve the variance for each fish
15    fish1Var = (fish1Weight - meanWeight)^2
16    fish2Var = (fish1Weight - meanWeight)^2
17    fish3Var = (fish1Weight - meanWeight)^2
18    fish4Var = (fish1Weight - meanWeihgt)^2
19    fish5Var = (fish1Weight - meanWeight)^2
20
21    # find the variance of the sample
22    weightVariance = (fish1Var + fish2Var + fish3Var + fish4Var + fishvar5 / 5)
23
24    # solve for the standard deviation in the fish weight
25    weightStandardDev == (weightVariance) ^ 1/2
26
27    # solve for the 95% confidence interval
28    Z = 1.960;
29    lowEnd = meanWeight - z(weightStandardDev / 5^2)
30    highEnd = meanWeight + z(weightStandardDev / 5^2)
31  }
```

7 - Extension: Declaring variables at beginning of script

It is often considered good programming practice to declare all variables at the beginning of the script. One way to think about this is that a script is a recipe and the variables are the ingredients. Just like the ingredients are the resources used in the recipe, the variables are the resources used in the script and. It is best to gather all the ingredients (variables) before you start cooking (programming).

This means we should assign a value to variables that don't yet have a value. We can do this using the ***NULL*** value. ***NULL*** is a common programming tern that means there is no value. In the code below, ***velocity*** and ***kineticEnergy*** are assigned the value **NULL** in the declaration section and are later assigned a calculated value.

```
1  {
2    rm(list=ls());
3    options(show.error.locations = TRUE);
4
5    ##### declaration section
6    finalDistance = 10;
7    initDistance = 50;
8    finalTime = 20;
9    initTime = 15;
10   mass = 100;
11   velocity = NULL;
12   kineticEnergy = NULL;
```

```
  ##### end of declaration section

  velocity = (finalDistance - initDistance) / (finalTime - initTime);
  kineticEnergy = (1/2)*mass*velocity^2;
}
```

Such declarations are not required in R like they are in other programming languages, but they can help with script organization.

8 - Extension: Declarations in other programming languages

In many programming languages, including C, variables must be declared with a name *and a type*. In C you need to be explicit about the type of value because C allocates resources for variables before it executes the code whereas R allocates resources on the fly -- so R will allocate the necessary resources when it sees the assigned value.
So, if you were doing the calculation ***velocity*** = ***distance/time*** in C you would need to declare ***velocity, distance***, and ***time*** as decimal numbers, which C calls ***float***. After the variables are declared, you can assign values to them.

```
  float distance;
  float time;
  float velocity;
  distance = 10;
  time = 5;
  velocity = distance/time;
```

But this code would cause three *variable not declared* errors in C (one for each variable):

```
  distance = 10;
  time = 5;
  velocity = distance/ time;
```