01-02: Variables

1 - Purpose

- Discuss what a programming variable is and how it compares to an algebraic variable.
- Discuss naming conventions for variables.
- Assign and reassign values to variables
- Introduce in-line comments.

2 - Concepts

3 - Different types of variables

Variables is a widely used term found in many different fields. Unfortunately, the definition of variable is not consistent throughout the fields. For this reason I will always precede the term ***variable*** with the *type of variable* (e.g., *algebraic variable*, *experimental variable*, *statistical variable*) with the exception of programming variable. *If you see the term variable without anything preceding it, then it is assumed to be a programming variable.*

3.1 - Algebraic variables

In Physics, properties are denoted algebraically by some symbol and the symbol represents some generic value that changes based on the situation. For example, velocity can be expressed algebraically as ***v=d/t*** where ***v***, ***d***, and ***t*** are symbols representing the physical properties velocity, distance, and time.

In this case, ***v, d***, and ***t*** are all algebraic variables that have the following properties:
1) **Name**: ***v, d, t*** (name is synonymous with symbol)
2) **Type**: numeric (***v***, ***d***, and ***t*** are all numbers)
3) **Value**: given by situation -- the value can be known or unknown

3.2 - Programming variables

In programming, **variables** work in a similar way. Programming variables are named storage locations in memory that hold information. This information could be a *runner's distance, time, and velocity* or it could be the *number of fish* caught in a day or the *time of day* the fish were caught.

Like algebraic variable, programming variables have ***name***, ***type***, and ***value***.

3.2.1 - Programming variable name

The name corresponds to the symbol in physics -- it is *how the variable* is represented and referred to in the script. The name is used to reference the storage location that holds the information. For our initial example, we will use the symbols (***v,d,t***) as the programming variables' names. Later in this lesson I will talk about variable naming conventions.

3.2.2 - Programming variable value

The value is the information that is stored in the location pointed to by the name. So, when ***v = 10*** (we will not worry about units), that mean there is a storage location in memory named ***v*** and in that location there is the number ***10***. *Trap: Constants in programming.*

3.2.3 - Programming variable type

The type describes what kind of value is being stored in memory which, in turn, tells the script what kind of operations can be performed on the variable. Distance, velocity, and time all have *numeric values* and mathematical operations can be performed on them. **Numeric values** are one type of variable but there are multiple types of values that can be stored. Some other types include **Boolean** (a TRUE/FALSE statement), **strings** (a text value like fish species), or **categorical** (a limited set of text values like the four seasons).

It would not make sense to perform most mathematical operations on ***strings*** or ***categorical*** variables. We will talk more about non-numeric variable types in future lessons, for now we will stick with numeric variables.

4 - Using variables in a script

We are going to create a simple script that calculates velocity given a distance and time.
So there are three variables in this script: ***d***, ***v***, and ***t***

To make this script this you need to:
1) open a new script file
2) add code to the script
3) run the code.

4.1 - Open a new script

In RStudio, click on ***File -> New File -> R Script*** (*Fig 1*)

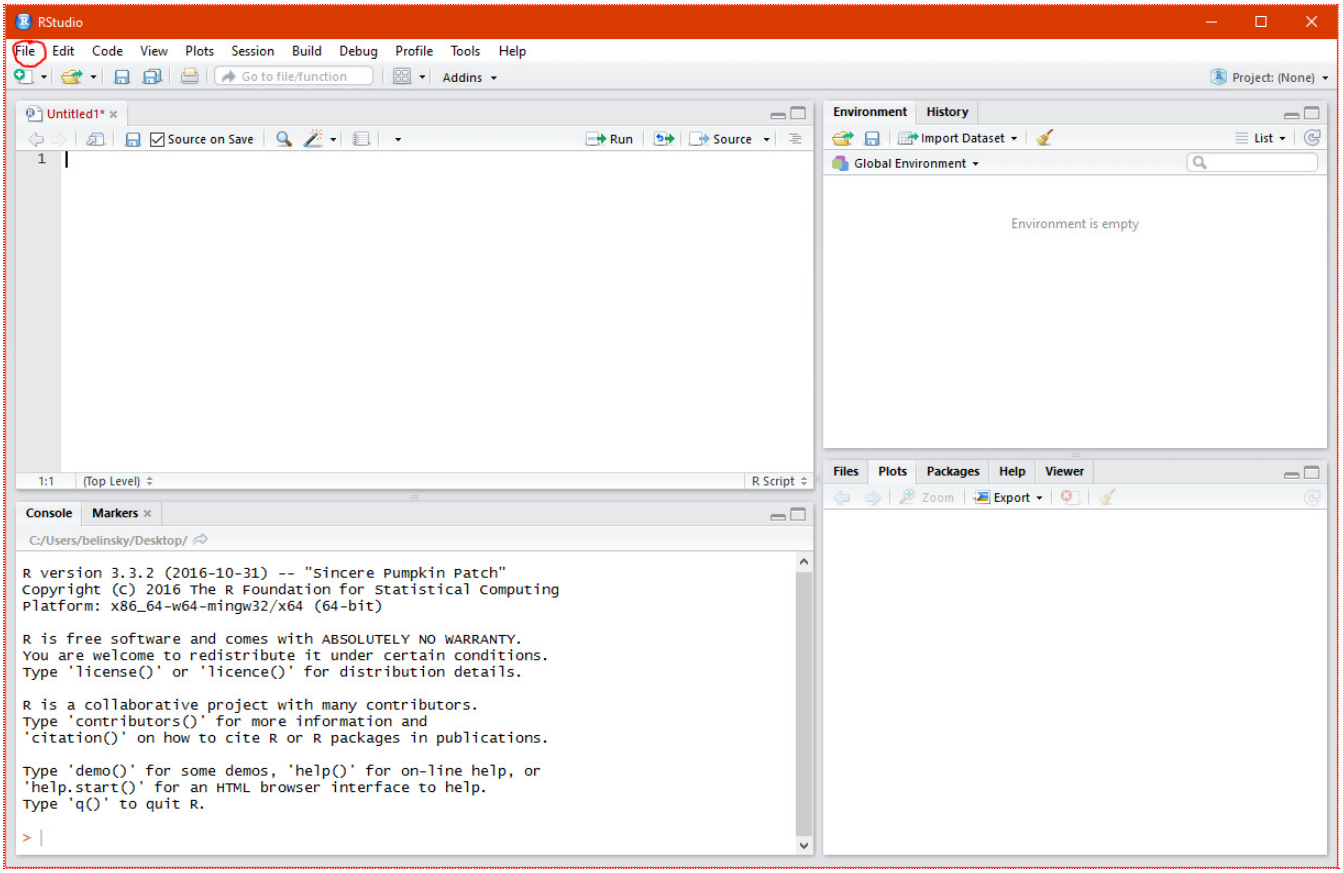

Fig 1: Opening a new R file in RScript

4.2 - Add code to the script

Copy and paste the following lines into the code window:

```
1  {
2    # the next two lines should be at the top of all your scripts
3    rm(list=ls());
4    options(show.error.locations = TRUE);
5
6    # create three variables: d, t, and v
7    # give d and t values and use them to calculate v
8    d = 100;
9    t = 20;
10   v = d/t;
11 }
```

Lines **2**, **6**, and **7** are comment lines, they are there to make the script easier to understand. You can take them out and the script will run exactly the same. Go ahead, try it -- but put the comments back in.

Lines **3** and **4** are helpful code that should always go at the beginning of all your scripts. They are explained further in *Extension: The First Two Lines*

Lines **3,4,8,9**, and **10** all have semicolons (***;***) at the end. The semicolon designates the end of a programming statement just like the period designates the end of a sentence. The semicolon is optional

and often not used in R but I highly recommend the use of semicolons in all your scripts.

Line **1** and **11** respectively have a start (***{***) and end (***}***) curly bracket. They designate the start and end of a codeblock, which is similar to a paragraph in script. Like semicolons, these curly brackets are useful but not required. *Extension: Why semicolons and curly brackets?*

Lines **8**, **9**, and **10** are where the real action occurs and each line contains a *variable assignment*:

- Line **8** assigns the value ***100*** to the variable named ***d***
- Line **9** assigns the value ***20*** to the variable named ***t***
- Line **10** assigns the value of the calculation ***d/t*** to the variable named ***v***.

Trap: Assignment vs Equality Operations
Extension: Alternate Assignment Operation

4.3 - Execute your code

Click ***Source*** (*Fig 2*). Your RStudio window should look like this:

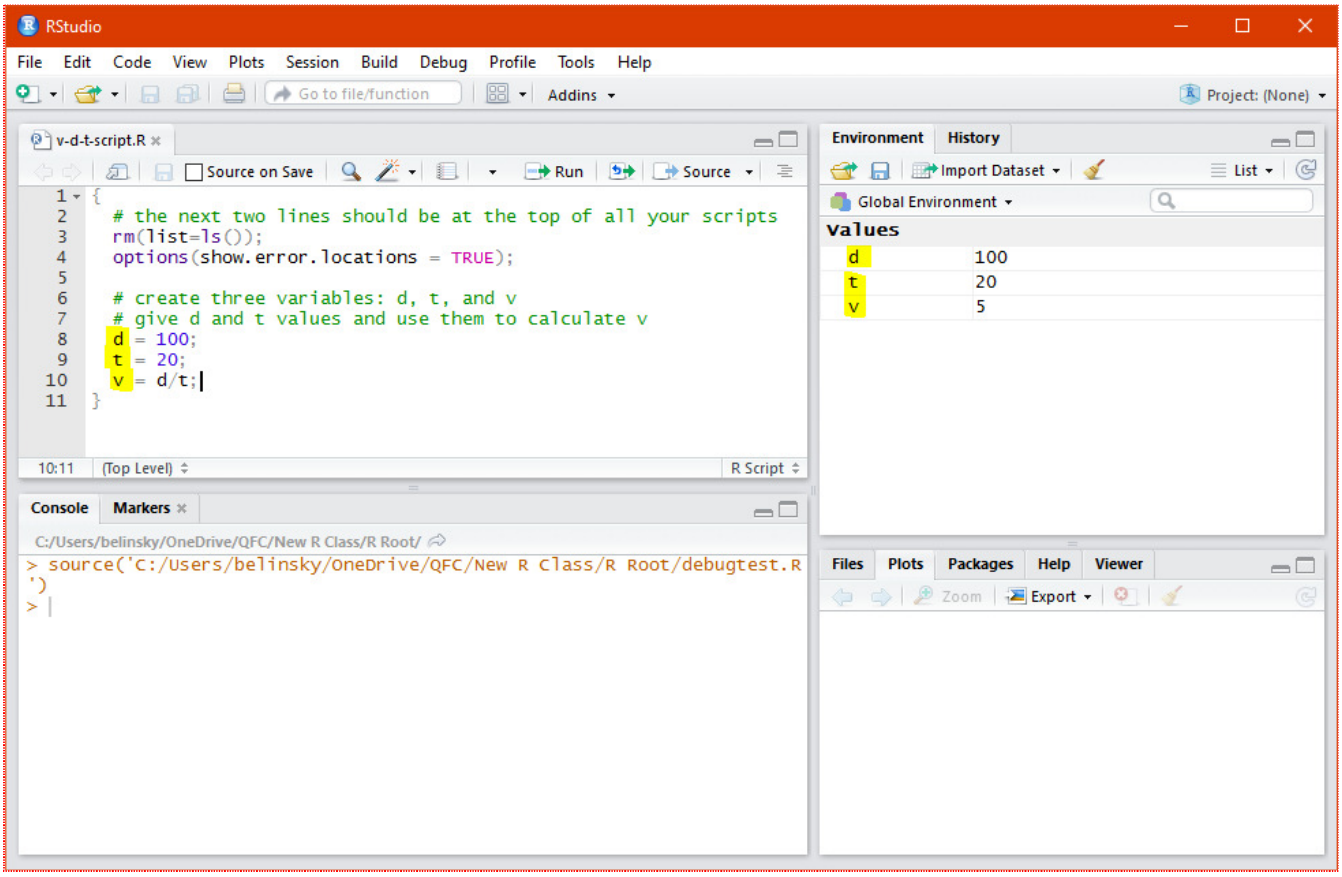

Fig 2: Executing (sourcing) the velocity-distance-time script

Note: The **Environment Window** displays the values for the variables ***v***, ***d***, and ***t***.

Try changing the values for ***d*** and ***t*** (lines 7 and 8) and click ***Source*** again to see how the values in the **Environment Window** change.

5 - Re-assigning values to variables

Variables get their namesake from the fact that their values can change.

In the code below...
1) ***d*** is assigned the value ***100*** in line 8
2) ***d*** to used to calculate ***v*** in line 10
3) ***d*** is assigned a *new value of* ***400*** in line 11

Run the code below to see what happens

```
1  {
2    # the next two lines should be at the top of all your scripts
3    rm(list=ls());
4    options(show.error.locations = TRUE);
5
6    # create three variables: d, t, and v
7    # give d and t values and use them to calculate v
8    d = 100; # assign d the value 100
9    t = 20;
10   v = d/t; # use d to calculate v
11   d = 400; # re-assigns d to 400
12 }
```

Notice in the **Environment Window** that the value of ***v*** remains ***5*** even though ***d*** changes.

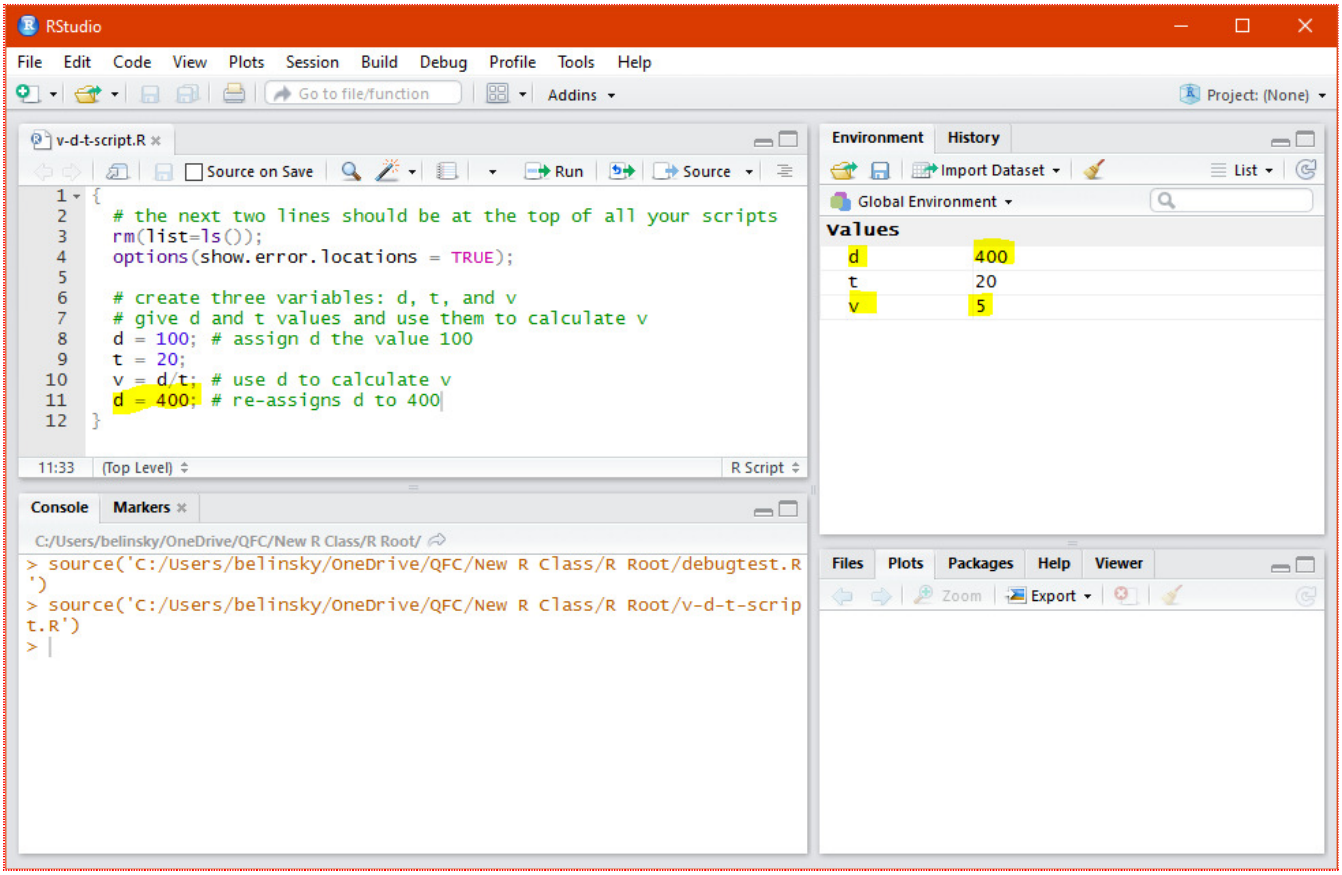

*Fig 3: Changing **d** in the script does not retroactively change **v***

The value of ***v*** remains ***5*** because line 10 assigns the value of ***d/t*** to ***v.*** But, as of line 10, ***d*** is still equal to **100** -- ***v*** does not get updated because one of the variables used to calculate it, ***d***, changes after the calculation is made.

v will remain ***5*** until it is assigned another value.

Trap: Assigning nonexistent values

What happens if you add the line ***v=d/t*** at the end of the script? Try the following code and click ***Source***

```
1  {
2    # the next two lines should be at the top of all your scripts
3    rm(list=ls());
4    options(show.error.locations = TRUE);
5
6    # create three variables: d, t, and v
7    # give d and t values and use them to calculate v
8    d = 100; # assign d the value 100
9    t = 20;
10   v = d/t; # use d to calculate v
11   d = 400; # re-assigns d to 400
12   v = d/t; # re-assign v using the new value of d
13 }
```

Notice that now the value of ***v*** has changed. This is because line 12 *assigns a new value* to ***v*** calculated using the value of ***d*** from line 11.

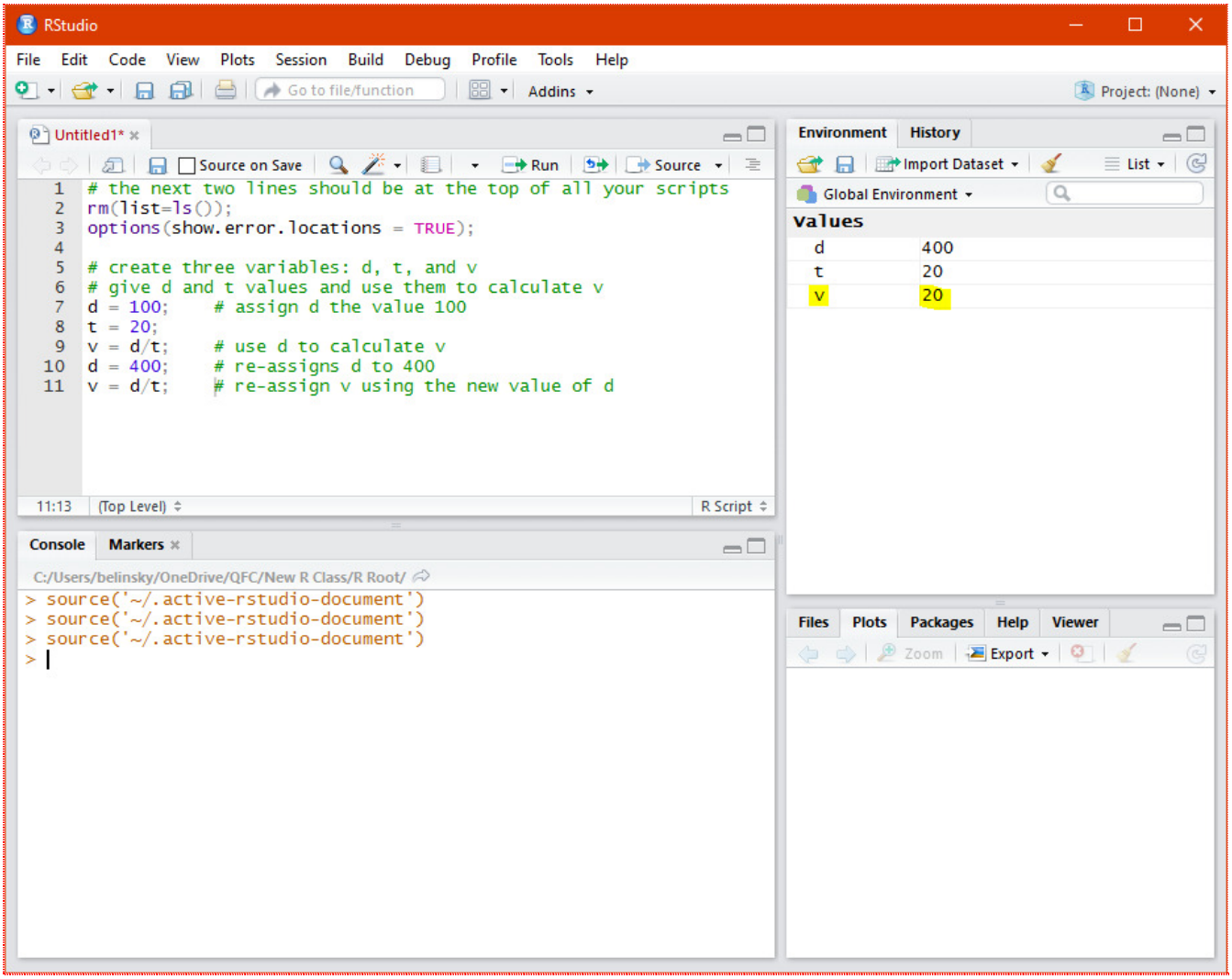

*Fig 4: Assign a new value to **v** after assigning the new value to **d***

6 - In-line and whole-line comments

The code in the previous script has example of *whole-line comments* (lines 2, 6, and 7) and *in-line comments* (lines 8, 10, 11, and 12). R ignores everything after the (**#**) but still executes everything on the line before the (**#**). In-line comments are a nice way to give a quick description of the code on the line, whereas whole-line comments are better for more robust descriptions.

7 - Naming variables

The problem with variable names like ***v***, ***d***, and ***t*** is that they are not very descriptive. It is good programming practice to give names that are descriptive so that people reading your code can more easily understand what is going on. So, the first step would be to spell the names out, for example: ***velocity***, ***distance***, and ***time***.

However, it is most likely that a script solving for velocity will be calculating multiple velocities. Perhaps the script is calculating the velocity of both a runner and a car -- the variable names should reflect this.

7.1 - Naming Conventions

The are two common programming conventions for variables names:
1) Capitalize the first letter of every word except the first: ***runnerVelocity***, ***runnerDistance***, ***runnerTime***
2) Put an underscore (_) between each word: ***runner_velocity, runner_distance, runner_time***

Trap: Case Counts

For this class I will use the first convention. For your class project, you need to choose one of these two conventions.

7.2 - Naming Rules

There are also a few rules for naming a variable:
1) It must start with a letter
2) It can only contains letters, numbers, the underscore (_), or dash (**-**)
   - note: dots (**.**) are also accepted in R but dots are not accepted in most programming languages
3) There can be no spaces in the name
4) There are system reserved words you cannot use as variable names (e.g., if, else, while, TRUE, FALSE, function, next...)

8 - Application

In RStudio, write a script that calculate the number of fish caught per day in:
1) the north fishing port, where there were ***1000 fish*** caught over an ***eight day*** period.
2) the south fishing port, there were ***500 fish*** caught over an ***ten day*** period.

To do this:
A) Create variables to represent the
1) number of fish caught at both ports
2) number of days to catch the fish at both ports
3) calculated number of fish caught per day at both ports

So, there should be a total of ***6 variables***.

B) Make sure you are using proper naming convention for the variables

C) Make sure all the variables appear in the **Environment Window** with correct values.

9 - Trap: Assignment vs. Equality Operations

The equal sign (**=**) in programming plays a different role in programming than in Algebra.

In algebra, the equal sign is an *equality operator* saying that *the two sides are equivalent to each-other.*
- So, in algebra, ***v = d/t*** says that ***v*** *is equivalent to* ***d*** divided by ***t***
- In this case, ***v*** will change if ***d*** or ***t*** changes

In programming, the equals sign is an *assignment operator* and it says that the *variable on the left side will be assigned the value calculated on the right side*.

- So, in programming, ***v = d/t*** says that ***v*** will be *assigned* the calculation of ***d*** divided by ***t***
- In this case ***v*** will not change if ***d*** or ***t*** changes. The value of ***v***, once assigned, is independent of the variables used in the calculations.

9.1 - Treating the equal sign as an equality operation

A very common error in programming is to treat the assignment operator (***=***) as an equality operator. The following statements make sense in algebra as *equality statements* but will cause errors in R.

```
1  d/2 = 100; # d/2 is not a valid variable -- you cannot "assign" a value to "d/2"
2             #                                (d = 100*2 is valid)
3  20 = t;    # '20' cannot be assigned the value of 't' (t = 20 is valid)
4  d/t = v;   # d/t is not a valid variable (v = d/t is valid)
```

Put the above lines of code in your script individually. You will notice an error message gets displayed in the **Console Window**. The error messages are often not intuitive but they do give you a line number.

In the following example (*Fig 5*), line 9 contains an error and the lines below it never get executed

```
1  {
2    # the next two lines should be at the top of all your scripts
3    rm(list=ls());
4    options(show.error.locations = TRUE);
5
6    # create three variables: d, t, and v
7    # give d and t values and use them to calculate v
8    d = 100;
9    20 = t;   # error!
10   v = d/t;
11 }
```

Notice that the execution of the script stops as soon as an error occurs -- so, in this case, line 10, which calculates ***v***, never gets executed.

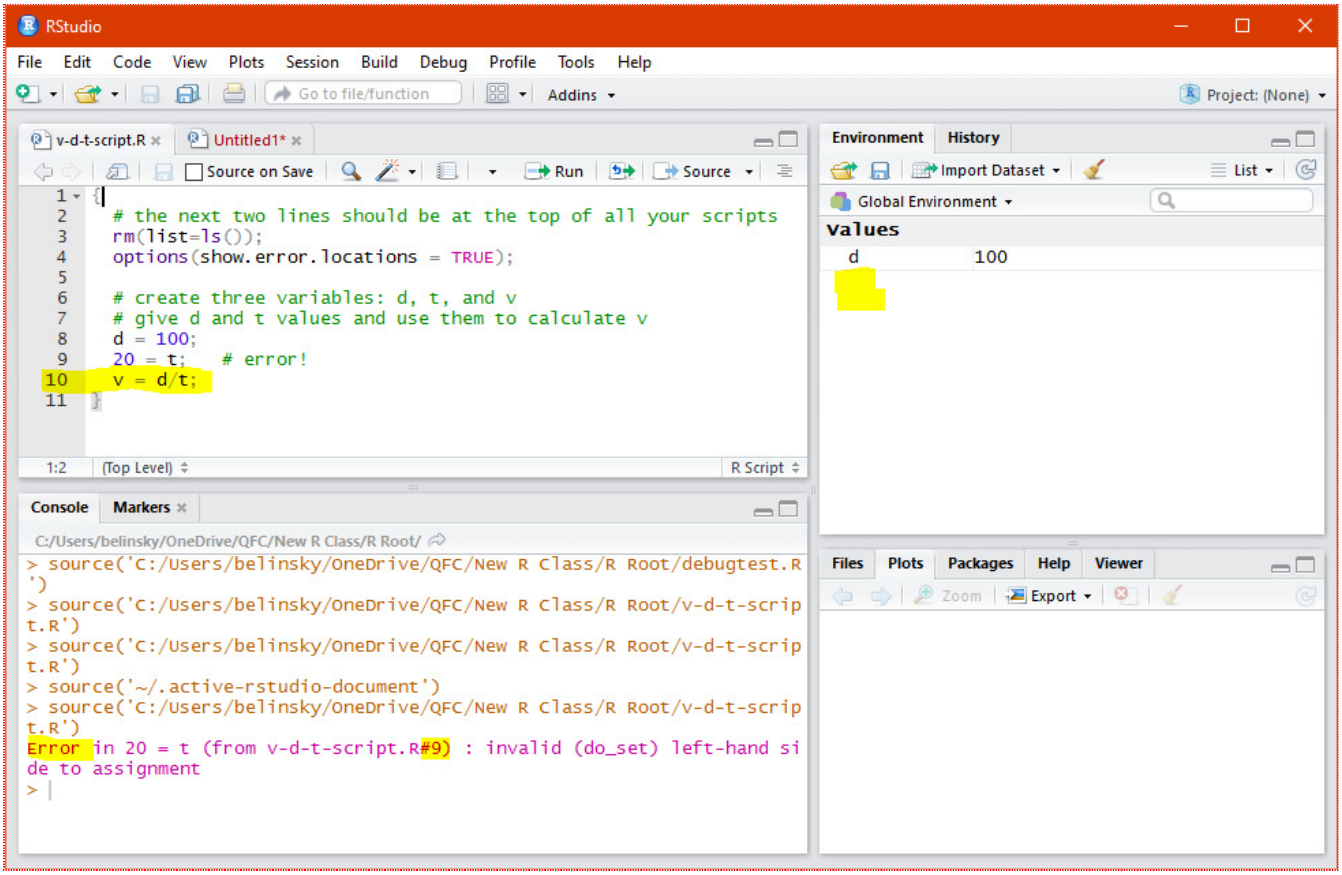

Fig 5: Assignment error in the R Script

10 - Trap: Constants in programming

The term *variable* suggests that the value can be changed and this is usually true. However, mathematical constants like ***π*** do not change, but they are still referred to as variables in programming. In other words, there can be a storage location referred to as ***pi*** that contains the numeric value ***3.1415***. These variables are often, and confusingly, referred to as *constant variables*.

11 - Trap: Assigning nonexistent variables

The first time you assign a value to a variable, a storage container is created in memory and a value is put in it. We call this a declaration (more about declaration in lesson 4). After a variable has been declared, it can be used in calculations and re-assigned values. Before a variable is declared, R knows nothing about the variable. Here is an example of using a variable before it is declared. *Notice the only change in this script was moving the line **v=d/t** before the declaration of **d** and **t**.*

```
{
  # the next two lines should be at the top of all your scripts
  rm(list=ls());
  options(show.error.locations = TRUE);

  # create three variables: d, t, and v
  # give d and t values and use them to calculate v
  v = d/t;  # error
```

```
  d = 100;
  t = 20;
}
```

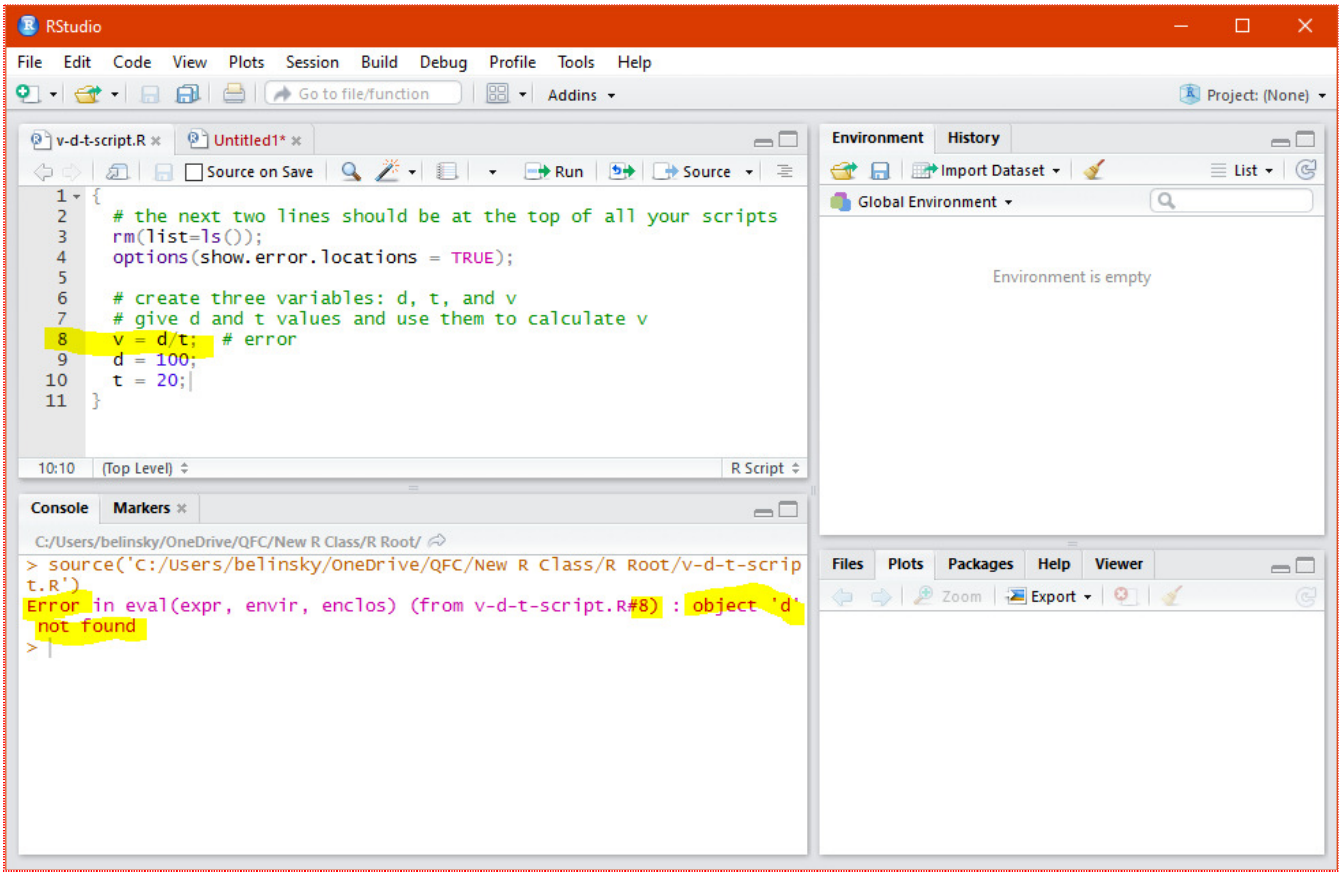

*Fig 6: Did not assign a value to the variable named **d** -- error*

There is an error in the console window (Fig) saying "object 'd' not found". The reason for this error is that, on line 8, the script is asked to assign the calculation d/t to the variable ***v*** but ***d*** *and* ***t*** *do not exist yet.*

Try the following code and see how the error changes. *Notice in this code **v=d/t** is after the declaration of **d** but before the declaration of **t**.*

```
{
  # the next two lines should, for now, be at the top of all your scripts
  rm(list=ls());
  options(show.error.locations = TRUE);

  # create three variables: d, t, and v
  # give d and t values and use them to calculate v
  d = 100;
  v = d/t;  # error
  t = 20;
}
```

12 - Trap: Case counts in variable names

In R, as in most scripting language, uppercase and lowercase letters are always seen as different. So, ***runnersTime*** and ***runnerstime*** are seen by R as two different variables. If the case is not correct then you will receive an "Object not found" error just like you would if you spelled the variable name wrong.

13 - Extension: the first two lines of code

The following two lines should, for now, be put at the beginning of any script you create.

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

- Line 1 cleans out the **Environment Window** each time the script is executed. Essentially this provides you with an clean slate each time you run a script and keep yours **Environment Window** from getting too cluttered.
- Line 2 gives the line number when an error statement appears in the **Console Window**. This is an incredibly useful feature for fixing you code -- especially as your script gets longer (*Fig 7*).

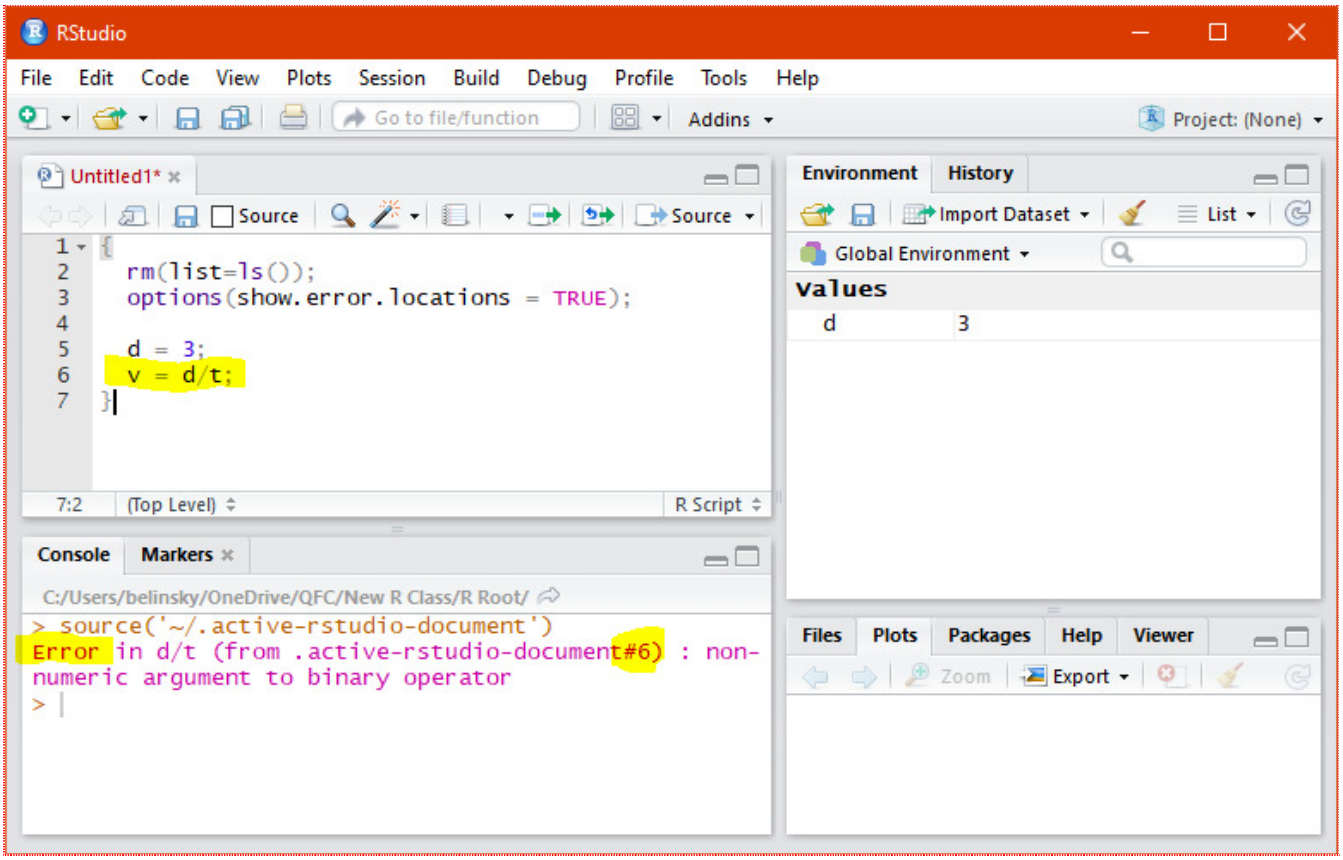

Fig 7: Line number of the error is given in the Console Window.

14 - Extension: alternate assignment operator

In R there are two assignment operators, the equal sign (**=**) and the arrow (**<-**). The two operators are (almost) completely synonymous.
In the vast majority of code, (**=**) can be replaced with (**<-**) with no change in functionality:

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

  d <- 100;
  t <- 20;
  v <- d/t;
}
```

You will see a lot of R programs written using the arrow. In this class, I will use the equal sign because that is the standards for most other programming languages (including JavaScript and C).

15 - Extension: Why semicolons and curly brackets?

You will almost never see curly bracket (***{ }***) put at the start and the end of an R script and you will rarely see semicolons put at the end of a statement -- your code will run just fine if you do not do either of these things. However, the two little practices make fixing (what we call debugging) your script easier.

RStudio has tools to help you fix mistakes in your script and putting the curly brackets at the beginning and end of your scripts and adding semicolons to the end of statements helps RStudio help you fix those mistakes. So the practice of adding semicolon and initial/final curly brackets are tricks to help find mistakes in your script. In some ways, they are making up for the deficiencies in the debugging tools within R and RStudio.

Also, both of these tricks are required when programming in C. So, I would suggest you use these tricks -- even if your fellow R programmers are not.