A Starting Point: Some Simple R

ACTIVITY 12

We are going to do some simple things to learn about R. This will be a guided exercise – meaning that the instructor will demonstrate the steps on the screen while you follow along on your desktop.

Purpose

The purpose of this activity is to give you the opportunity to become familiar with the R interface and syntax.

Learning Objective

To be able to use R.

Required Resources

- R, R Studio
- Chapters 1 & 2 in *Introductory statistics with R* (Dalgaard)
- R Script for this activity from class companion website

Time Allocated

60 minutes in class

Tasks

This activity begins assuming R Studio is open.
If not, please start R Studio and open the sample script for the activity (if provided)

A. Concept: Storing Values or Objects and Variables

At the R console prompt, type:

```
1+1
```

It returns:

```
[1] 2
```

The [1] identifies the index of the number on that line of the answer. It is not that helpful here, but there will be more on that later.

In R, you can assign values to variables. The assignment operator is `<-` is the operator that "assigns" the results of the operation to the right to the variable to the left.

You can use any text you want to create variables, though it must not start with a number or a dot followed by a number.

IMPORTANT: R syntax is case-sensitive. *Y* is **not** the same variable as *y*. Type:

```
y <-1+1
```

Notice now that R just returns you to an empty command prompt. But the value of operation 1+1 has been stored as the variable "`y`" in R's memory. This only lasts until you clear the memory, or end the session. You will see later that there is a lot of flexibility in assigning many objects to variables, including entire data frames.

Type `y` at the command prompt to retrieve the value of y. Notice that you get the same output as before:

```
> y<- 1+1
> y
[1] 2
```

TIP: name your variables in one case (upper or lower) and try to use short but memorable names. If you want to use more than one word combine them with a . or _ character. Names like `my.variable` or `my_variable` are examples. Don't use spaces as it will only cause you trouble. Again, variables cannot begin with a number.

1. Now, you try to assign another variable the value of `y+y`. What should be the value of that variable?

```
> x<- y+y
> x
[1] 4
```

Type `ls()` at the R prompt. This shows you a list of objects in memory. Until you reassign or clear the memory, R will remember the values of these variables.

```
ls()
[1] "x" "y"
```

You can also see these two objects shown in the workspace tab of R Studio. This is a very helpful window since it shows you the current objects in memory and their values.

To clear objects in memory, type

```
rm(list=ls(all=TRUE))
```

2. Does R still know what value of `x` you assigned to the variable you created to store `y+y`?

Now let's go ahead and create a script file that you can log your program in. Choose ***File > New > Script File***. Select the file you downloaded from the class website.

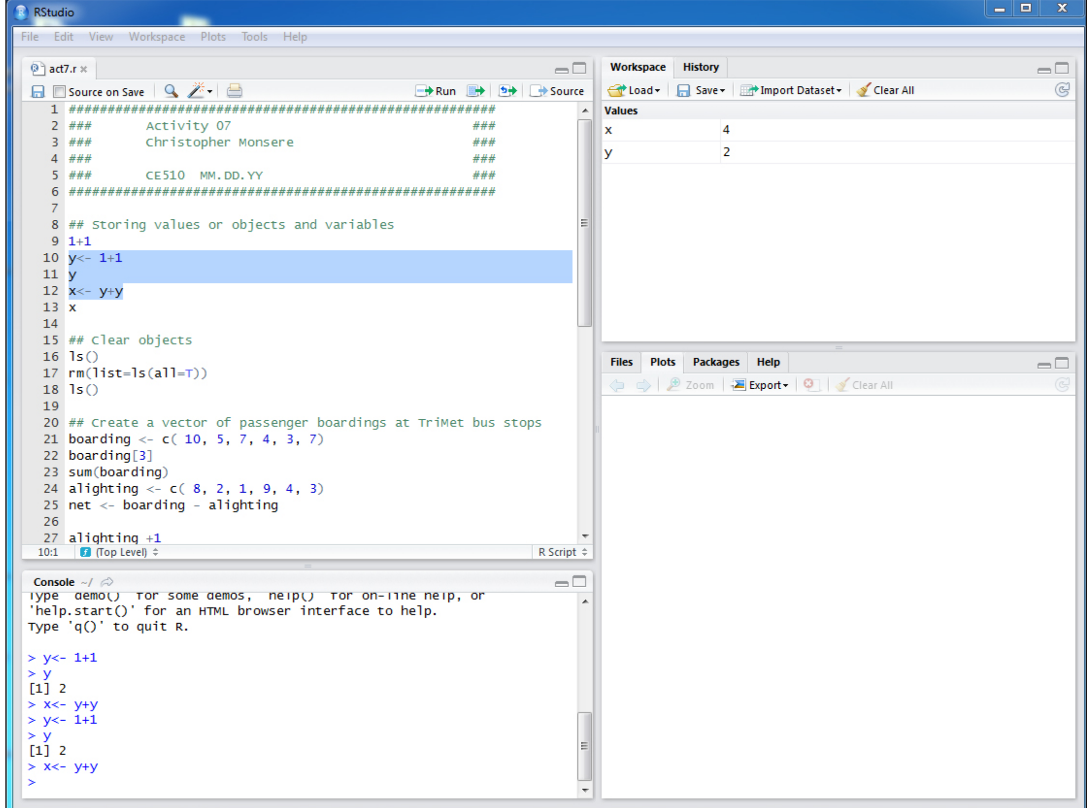


Figure 13 Screen Capture of R Studio (with R Script File open)

B. Concept: Vectors

One of the strengths of R is its use of vectors and the ease of which things can be manipulated and calculated with the use of vectors. These vectors can be numbers or text. The most common way to create a simple vector is to use the `c(,,,)` construct. Each element is separated by a comma (`,`). Let's create a vector of passenger boardings at a sequence of TriMet bus stops:

```
boarding <- c( 10, 5, 7, 4, 3, 7)
```

Notice that there are six elements in the vector. Click on that object in the R Studio workspace window. It returns to you the value of the boarding objects. We can extract the value in any one position by using the index. The index of a one row vector is the position of the element in the assignment. To get the value in the third position (a value of 7), we tell R with this syntax:

```
boarding[3]
```

R will apply many standard operations to the vector. To sum the values type:

```
sum (boarding)
```

3. Can you write the R syntax that would do this the "hard" way by adding each element in the vector?

R will apply "vectorized" arithmetic to operations combining vectors of the same length. Let's create another variable of the number of people getting off the bus at each stop:

```
alighting <- c( 8, 2, 1, 9, 4, 3)
```

Let's subtract these to show the "net" passengers at each stop:

```
net <- boarding - alighting
```

Can you see how this operation is akin to creating the formula in a cell in Excel, then copying and pasting this operation for each row?

If vectors are not the same length, the shorter vector is "recycled" and used again. A special case of this is a vector of single value — try adding 1 to every alighting value.

```
alighting +1
```

See how 1 is added to every element of *alighting*? Now try adding the vector `c(1,10)` to *alighting*

```
alighting + c(1,10)
```

To see how this works, first confirm that the function `c(1,10)` produces a 2-element vector. When added to alighting (which is 6 elements long), the `c(1,10)` is recycled twice. Notice how 1 is added to the first element, 10 to the second, 1 to the third and so on. The bottom line is that you need to be careful if vectors are not of the same length and you are performing operations on them.

4. If there were 40 people on the bus at the first stop, how many people are on the bus at the last stop? Write the R code to calculate this.

Lastly, there are lot of other helpful functions and syntax to create vectors.

One is the shorthand for from:to; R knows you want a vector back. Try typing:

```
1:10
```

Another way to do the same thing:

```
seq (from=1,to=10, by=1) or simply seq(1,10,1)
```

This short cut to creating vector lists can be really helpful.

C. Concept: Data Frames

If it isn't already obvious, working with data is the theme of this class. Vector concepts are the "building blocks" of data frames in R. Let's create another variable called *stop_id*:

```
stop_id <- c( 1200, 1030, 1456, 300, 1251, 1800)
```

Let's "glue" these vectors together. If we use the command `cbind` ('column bind') we can stick these vectors together as columns.

```
cbind ( stop_id, boarding, alighting)
```

Notice what is returned:

```
stop_id boarding alighting
[1,] 1200 10 8
[2,] 1030 5 2
[3,] 1456 7 1
[4,] 300 4 9
[5,] 1251 3 4
[6,] 1800 7 3
```

The syntax `[1,]` is a row, column notation. To refer to any value in this combined data frame, we can use `[r,c]` where **`r`** is the row index and **`c`** is the column index for the element.

Before we do that, notice that we didn't actually assign this `cbind` operation to anything; right now the only output is to the screen. R won't remember this until you tell it where to store it.

Let's assign this `cbind` operation to the variable *trimet*:

```
trimet <- cbind ( stop_id, boarding, alighting)
```

5. Can you extract the number of passengers boarding at stop 1456 from this *trimet* vector array using the row, column syntax?
6. You can also "glue" things together with `rbind`. ('row bind'). Try it and see what you get here. Do you notice the difference?

It is often extremely helpful in R to know what type or class an object is. A quick check of this will help you in a lot of debugging exercises. The command to use is `str()` ('structure').

```
str(trimet)
```

The return here is a little cryptic but this object is a numeric array of 6 rows (1:6) and 3 columns (1:3) and has attributes of NULL row names and character column names. Learning to decode these will help you in a lot of debugging exercises.

Let's turn this numeric array into a `data.frame` (a special class of R objects)

```
trimet <- as.data.frame (trimet)
```

It is not necessarily good practice to overwrite a variable (the original *trimet* has been overwritten by the data frame *trimet*) but as long as you keep track of things it can be okay in development of code. One thing that you will want to do is at the end of your code, to clear things from memory and run it to make sure everything works as planned. Now check the structure:

```
str(trimet)
```

What does it say now? Notice how R was smart enough to take the variable names and assign them as the column names. In a data frame, you have two options to select data and both can be helpful.

The first is by `dataframe$columnname`. Try

```
trimet$boarding
```

The second is by the `[r,c]` syntax. Try

```
trimet[,2]
```

Read this as [all rows, just column #2]. You can also specify it like this,

```
trimet[,c(1,3)]
```

Which reads, [all rows, column #1, #3]. Or, use the negative sign to mean "not this column or row" as in

```
trimet[,-2]
```

Let's add the variable net to the data frame. This is very easy to do using the `dataframe$columnname` syntax.

```
trimet$net <- trimet$boarding - trimet$alighting
```

Subsetting data frames is also easy and like many things in R, there is more than one way to do things, each helpful in different contexts. This flexibility can be part of the reason that new users get frustrated with R! Think of subsetting data frames as very similar to the SQL exercises that you were doing.

Let's say we only want records from column `stop_id` with `stop_id >= 1200`

```
trimet[stop_id >= 1200, ]
```

The `[r,c]` notation and the comma make all the difference. Try

```
trimet$stop_id[stop_id >= 1200]
```

What are you getting here? Just the elements of `stop_id` that are greater than or equal to 1200.

I think by far the easiest syntax to remember is the `subset` function:

```
subset (trimet, stop_id >= 1200)
```

Notice this returns the whole data frame subsetted. You can combine logic in the subsetting criteria such as

```
subset (trimet, stop_id >= 1200 & net >=0)
```

It is helpful to assign the subsetted dataframe to another variable name so that you can keep the original data intact:

```
trimet_stop1200 <- subset (trimet, stop_id >= 1200)
```

Notice now that if you will need to refer the subsetted data frame as:

```
trimet_stop1200$net
```

R Studio is also advantageous because it has a built in viewer that allows you to see the data frame. Click on *trimet* in the workspace window and see the view of the dataframe.

D. R Concept: Packages

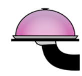
The base R install will do much of what you need. However, sometimes you need to add functionality. These packages are all on the CRAN. Downloading and installing them is an easy, menu-driven process. Be sure you set up the **RLibs** directory.

Let's get the package ('ISwR') for the Dalgaard textbook.

In the R Console, choose ***Packages > Install Packages***. Select an appropriate "mirror" site (choose your favorite University or country on the list), then find the package you are interested in. It is advisable to choose a "close" server.

Packages can contain functions and or data and are contributed by open source users.

Deliverable



Annotated code – submit your R script to the Dropbox with the answers to the six questions. In your R code use comments to indicate your answers but make sure the R code that you modify to answer the questions can be run by the instructor to verify that it works. For example:

```
#Question 3 - The "hard" way would be to add each element of the vector
individually using the [] syntax. My code to do this would be

boarding[1]+boarding[2]+boarding[3]+boarding[4]+boarding[5]+boarding[6]
```

Assessment

Activity 12   Grading Rubric

	Excellent (10)	Good (8)	Poor (6)	0
Script	Organized, complete, accurate and executes.	Missing minor parts, but executes and is otherwise organized and accurate.	Missing significant portions of the activity, unorganized, inaccurate, but executes.	Code does not execute
Annotation	Annotations are complete and describe what the code is accomplishing.	Some annotations are incomplete or do not describe what the code is accomplishing.	Annotations were not included.	Code does not execute
Discussion	Insightful discussion or commentary relating to the question at hand demonstrating student understanding of the task.	Discussion or commentary was partially incomplete.	Minimal to no discussion or commentary.	Code does not execute