Prep for Lab 01 - Familiarizing Ourselves with R

PSC-012Y

Jonathan J. Park

09/26/2024

Contents

Welcome to our first Virtual Tutorial for PSC-012Y! Throughout this session, you can see the titles of things that we'll be discussing today.

Introduction to R, RStudio, and .RMD

What is R and RStudio?

So, what is R?

Well, it's a coding language! Aside from all of the technical aspects of it, R is just a tool like Microsoft Word, Excel, or PowerPoint that may seem a bit more intimidating than those others but–once you're familiarized with it, it's not too bad… probably.

R is the language that we will write in but R itself is quite ugly and not user-friendly.

The RStudio Interface and Navigation

Because of that, folks decided to get together and make RStudio which is much more user-friendly and is what we're making this tutorial in! We'll spend some time here now in the virtual tutorial exploring the interface of RStudio and highlighting some important features that could prove helpful to us as we embark on this learning adventure.

What is an .RMD file?

.RMD's are just file formats like Word documents being .docx or music files being .MP3. The .RMD file is what will enable us to create reports of our work that seamlessly integrate the text we write, the code we write, and the visualizations we create. The benefit of this is that all of our work is in one place and we can easily share it with others.

It also means that if we ever get new data, we can just plug it in and reuse this script to generate updated results and visualizations.

Code "Chunks"

When we're in the .RMD file environment, we can create things called "code chunks". These will be places where we write our R code which is separate from the text we're writing. We can create a code chunk below by using three back-ticks and then r and then three more back-ticks.

Let's go ahead and make an R chunk now:

You can see that the chunk is a different color than the rest. This is a good indication that it only accepts R code so regular text won't work inside of those areas. This is important to remember as it can be easy to accidentally put R code in your text without meaning to!

When you make an R chunk, you can add arguments after the letter r to tell the .RMD file whether you want certain things to show or not. Let's go over some of the functions that may be useful to you throughout the quarter:

- `echo = TRUE` will show the code that you wrote in the chunk
- `eval = TRUE` will run the code in the chunk
- `include = TRUE` will show the output of the code in the chunk
- `message = TRUE` will show any messages that the code in the chunk produces
- `warning = TRUE` will show any warnings that the code in the chunk produces
- `error = TRUE` will show any errors that the code in the chunk produces

Let's try testing this out:

```
words do not belong in r chunks
```

```
## Error: <text>:1:7: unexpected symbol
## 1: words do
##           ^
```

You can see that the code chunk above produced an error because it was not R code. Thankfully, the argument `error = TRUE` forced the code to run anyway so we could see the error message.

Getting `?help`!

If you ever see an error message, it may be useful to go look for `help` relating to code that isn't working. You can type: `?function_name` to get help on a specific function.

We'll test that out in the coming sections; so, just keep it in your back pocket for now.

Functions and Packages

Functions in R are like functions in math. They take in inputs, do something to those inputs, and then return output.

Typically, functions will be "called" using the name of the function, a set of parentheses, and then you'll put the arguments it needs inside of those parentheses. Let's do one together:

```
message("Hello World! This is PSC-012Y!")
```

```
## Hello World! This is PSC-012Y!
```

Functions can also have multiple arguments. Let's try out the `rep()` function and use the help documentation to learn about how it works and what it does:

```
?rep
```

```
## starting httpd help server ... done
```

```
rep(x = 1, times = 5)
```

```
## [1] 1 1 1 1 1
```

```
# If we know the order of arguments, we can skip the variable names:
  rep(1, 5)
```

```
## [1] 1 1 1 1 1
```

Cool, so it looks like the `rep()` function repeats a set–or vector–of numbers to us. It accepts arguments which are:

- The things you wanted repeated, `x`
- The number of times to repeat that thing, `times`

When you give both of those to `rep()` then you'll get what you want.

Packages in R

When someone makes a bunch of functions, they may turn it into a statistical *package* which is just a collection of functions which may have the same theme or purposes. For example, the package `ggplot2` is a package with many functions relating to the visualization of data.

By installing a package and "loading" it in, we gain access to all of the functions within that package.

Installation and Loading Packages

We can find packages on the internet through the `CRAN` repository or we can search them individually in the `RStudio` tab for *packages* but for the purposes of this tutorial, we'll just do `ggplot2`. Let's go ahead and make that happen.

```r
install.packages(ggplot2)
```

Hm, what went wrong? Let's read the error message that popped up and see what's wrong.

Ah! Since we're telling R to install a name, we have to use quotation marks to tell R that the thing we want installed is a name rather than a variable called ggplot2. Let's give it a try:

```r
install.packages('ggplot2')
```

Finding yourself in R

There's a popular saying that R–and all other computer programming languages–is dumb. That's not to say that R can't do amazing things; but, rather that R can only do amazing things that you tell it to do. Tell R to solve a math problem and it'll get it wrong if you don't write the equation correctly for example.

R likes to know where it is in your computer. This becomes important when we're looking for a file to read in (e.g., a dataset). If you have your dataset saved onto your desktop, you can't just tell R to read in your dataset because it might think it's in your Downloads folder.

So, to avoid those kinds of headaches, we can ask R where it thinks it is using the `getwd()` which means, "get the working directory".

Let's see where our computers think they are:

```r
getwd()
```

```
## [1] "C:/Users/MyComputer/Dropbox/Jonathan Park/2. UC Davis/2024-2025/Courses/PSC-12Y/Week 1"
```

We can also change where R is looking by adjusting the "working directory" using the command: `setwd()` which is just us telling R where to look.

If I want R to look at my Desktop–for example–I might say:

```
setwd("~/Desktop/")
```

But these instructions will depend on your own personal computer. For example, on PC it might be:

```
setwd("C:/Users/MyComputer/Desktop/")
```

But if your computer has an account it could be:

```
setwd("C:/Users/imJPark/Desktop")
```

Where `imJPark` is just your computer account username. This will all be computer-specific and can be a difficult thing to solve. If you run into issues with this, let me know!

```r
setwd()
```

Coding in R

Variables: Objects and Assignments

We can create variables in R chunks. Variables in R have to follow certain rules:

- Variables can't start with a number: `1VAR`
- Variables can't have spaces: `VAR 1`
- Avoid using single-letters for variables because they may coincide with arguments: `T` or `F`

There are some best practices when creating variables like using _'s or .'s when you would use spaces. For example:

- Instead of `1VAR` we could use `VAR_1` or `VAR.1` to identify our first variable.

When we've chosen the name we want for our variable we have to give it a value. This is known as "assigning" a value to the variable. We can do that using the = operator. Let's test this out by creating a variable called `VAR.1` and saying it's equal to 10

```
VAR.1 = 10
```

Great! We've stored a variable into our "environment" which has the value of 10 stored inside of it.

Now, as of now, you just have to trust me that it exists there. Luckily, there is a function that will let you view the variable we just created called `print()`

`print()`ing output

Throughout the quarter, you'll be asked to show the output of your code-based work. The `print()` function will be your best friend in these cases. Let's go ahead and print out the variable we just made:

```
print(VAR.1)
```

```
## [1] 10
```

Data Structures in `R`

Data in `R` have certain characteristics. Some data can do things that other data cannot do depending on their "structure" We'll talk a *lot* more about this next week when we talk about types of data both in-lecture and virturally but let's go over some basics here by introducing the concept of scalars, vectors, and matrices.

Scalars

Scalars are just single values in `R`. Just like the variables we covered in lecture, scalars can take on certain characteristics like being a number, a "character", or a "logical" value. We'll get to the concept of `logical values` next week but for now, we can think about numbers and characters as types of data.

We already created a scalar when we created our variable `VAR.1` above but let's create a new variable called `VAR.2` which is a "character":

```
VAR.2 = "Hello World!"
print(VAR.2)
```

```
## [1] "Hello World!"
```

In this case, we need to use quotation marks so that `R` doesn't think that "Hello" and "World" are two completely different variables. By surrounding them in quotation marks, we are basically saying that both "Hello" and "World" are a single thing and we are saving that into `VAR.2`.

We can even turn numbers into characters:

```
VAR.3 = "3"
print(VAR.3)
```

```
## [1] "3"
```

Notice that the thing we printed out has quotes around it? That's because `R` is treating this like a character–text–rather than as a number.

Sometimes, data aren't only scalar though. For example, we may have a set of scalars that are all from the same variables. For example, if we measured the IQ of 5 people. We would want a variable called IQ and it should contain each person's IQ score.

The thing we need from that is called a `vector` which we'll move onto next.

Vectors

As stated above, vectors are sets of scalars that are–in some way–related or similar to one another. It's easiest to think of these as variables which contain multiple numbers or measurements.

Let's run with the example of IQ scores. Let's create a variable called `IQ` and store 5-IQ scores into it using a new function called "concatenate" or `c()`

```
?c
IQ = c(100, 95, 98, 131, 101)
print(IQ)
```

```
## [1] 100  95  98 131 101
```

But what if we have multiple variables that we are studying? After all, we often try to visualize multiple things at once.

Perhaps, we also have a variable on Educational Attainment and another variable for Household Income. Let's make those two variables too:

```
EdAttain = c("High School", "Some College", "College", "College", "College")
HouseInc = c(35, 83, 47, 120, 98)
```

But how could we combine these different `vectors` and place them together? Well, we can use matrices and dataframes!

Matrices and Dataframes

Matrices and dataframes can be groups of vectors but there are important differences between what a matrix is and what a dataframe is. Let's be clear and explicit and work out some examples.

For the course, the primary way of telling the difference between a matrix and a dataframe is that:

- Matrices are all one "type" of variable while dataframes can have multiple types of variables at the same time.

What does that mean? Let's make one of each and compare. But before we do that, we need to learn about "binding".

There are a ton of ways of "binding" different "vectors" together but the most common you'll find are:

- `rbind()`
- `cbind()`

Which stand for row- and column-binding, respectively. That that means is that these functions will take your data and stack them on top of each other (`rbind()`) or next to each other (`cbind()`).

We'll need to use `cbind()` for our "matrix":

```
?matrix
matrix(data = cbind(IQ, EdAttain, HouseInc))
```

```
##       [,1]
##  [1,] "100"
##  [2,] "95"
##  [3,] "98"
##  [4,] "131"
##  [5,] "101"
##  [6,] "High School"
##  [7,] "Some College"
##  [8,] "College"
##  [9,] "College"
```

```
## [10,] "College"
## [11,] "35"
## [12,] "83"
## [13,] "47"
## [14,] "120"
## [15,] "98"
```

```
# Correction
matrix(data = cbind(IQ, EdAttain, HouseInc), ncol = 3)
```

```
##       [,1]  [,2]           [,3]
## [1,] "100" "High School"  "35"
## [2,] "95"  "Some College" "83"
## [3,] "98"  "College"      "47"
## [4,] "131" "College"      "120"
## [5,] "101" "College"      "98"
```

Notice that the matrix we just made has quotation marks around every column. This is because matrices *must* have the same "type" of variable inside of them.

We talked before about "characters" and numbers. In this case, `R` tries to turn all 3 columns into the one thing that they could all be.

Since text cannot be a number, `R` tries to turn the numbers into characters–or text–which all three columns can be. This can be annoying to deal with when our data are different kinds of information.

Let's see what the dataframe looks like when we create it using the `data.frame()` function:

```
data.frame(IQ, EdAttain, HouseInc)
```

```
##    IQ     EdAttain HouseInc
## 1 100  High School       35
## 2  95 Some College       83
## 3  98      College       47
## 4 131      College      120
## 5 101      College       98
```

Cool, notice that the numbers don't have quotation marks around them? That tells us that `R` is correctly keeping numbers as numbers and text as characters.

Mathematical Operations

Once we have our data, we may be interested in doing math on that data to calculate things that will be useful to visualize.

We can do math in `R` and it's pretty easy to do so considering it's one of the primary purposes of the language. Let's try doing some mathematical operations using the following:

- Scalars with Scalars
- Scalars with Vectors
- Scalars with Matrices

Scalars

This is what you're already probably familiar with when you think about math. This is when you take single variables or numbers and do math with them.

`R` has a lot of built-in calculating functions for common operations you'll do. Let's demonstrate some!

```
new.VAR.1 = 5
new.VAR.2 = 16

new.VAR.1 + new.VAR.2
```

```
## [1] 21
```

```
new.VAR.2 - new.VAR.1
```

```
## [1] 11
```

```
new.VAR.1 / new.VAR.2
```

```
## [1] 0.3125
```

```
new.VAR.2 * new.VAR.1
```

```
## [1] 80
```

```
new.VAR.1 ^ 2
```

```
## [1] 25
```

We can also do operations:

```
sqrt(new.VAR.2)
```

```
## [1] 4
```

```
exp(new.VAR.1)
```

```
## [1] 148.4132
```

```
pi
```

```
## [1] 3.141593
```

and order of operations is respected:

```
1 + 2 * 3
```

```
## [1] 7
```

```
# Versus:
  (1 + 2) * 3
```

```
## [1] 9
```

Scalars and Vectors

```
V1 = 3
V2 = c(1, 2, 3)

V1 + V2
```

```
## [1] 4 5 6
```

```
V1 * V2
```

```
## [1] 3 6 9
```

```
# But some things depend on which comes first:
V1 / V2
```

```
## [1] 3.0 1.5 1.0
```

```
V2 / V1
```

```
## [1] 0.3333333 0.6666667 1.0000000
```

```
V1^V2
```

```
## [1]  3  9 27
```

```
V2^V1
```

```
## [1]  1  8 27
```

Scalars and Matrices

```
V1 = 3
V2 = matrix(c(1, 2,
              3, 4),
            nrow = 2, ncol = 2)
V1 + V2
```

```
##      [,1] [,2]
## [1,]    4    6
## [2,]    5    7
```

```
V1 - V2
```

```
##      [,1] [,2]
## [1,]    2    0
## [2,]    1   -1
```

```
V1 * V2
```

```
##      [,1] [,2]
## [1,]    3    9
## [2,]    6   12
```

```
V1 / V2
```

```
##      [,1] [,2]
## [1,]  3.0 1.00
## [2,]  1.5 0.75
```

```
V1 ^ V2
```

```
##      [,1] [,2]
## [1,]    3   27
## [2,]    9   81
```

```
V2 ^ V1
```

```
##      [,1] [,2]
## [1,]    1   27
## [2,]    8   64
```

Reading in Data

We won't always be manually typing our data by hand. Sometimes, we'll be given our data and need to "read" it into R

That involves using multiple things we've learned so far: - Assignment - Working Directories

Let's try reading in a dataset. You've been provided on Canvas, a dataset called, "OL1.dat" which stands for the dataset for Online Lesson 1

```
ol1.data = read.table(file = "C:/Users/MyComputer/Desktop/OL1.dat", sep = ",")
```

Describing Data

It seems like our attempt at reading in the data was successful but there are several ways for us to make sure. Let's start by "describing" our data.

There are fancy and more sophisticated ways to do this but we'll just to a simple check by using the function `head()` which shows us the first 6 rows of our data.

```
head(ol1.data)
```

```
##    IQ    Income DroppedOut
## 1  88  94124.27      FALSE
## 2 112  91796.83      FALSE
## 3 102  92735.04       TRUE
## 4  85  73180.28       TRUE
## 5  95  68914.90       TRUE
## 6 118 107366.14       TRUE
```

It looks like our dataset contains IQ scores and Income data for a group of participants as well as information of whether they dropped out of the study.

In the next coding tutorial, we'll go over how to describe our data in more detail with numerical information when appropriate.

Subsetting Data

Subsetting data is something that is incredibly common and–should you ever work with real data–something you will need to know how to do.

In R, subsetting data can be super intuitive!

We can subset data by specifying the `rows` and/or `columns` of our dataset we want to keep. For example, if I wanted to only see rows 30 to 35 of my dataset I can just do this:

```
ol1.data[30:35,]
```

```
##     IQ    Income DroppedOut
## 30 133 112731.55      FALSE
## 31 124  74743.81      FALSE
## 32 123 124861.18       TRUE
## 33 102  66881.54       TRUE
## 34 111  52850.87      FALSE
## 35  91  76858.87       TRUE
```

When we use the `[]` brackets after our dataset, we're telling R that we want to "index" a part of that data. The first part of the brackets is for the rows and the second part is for the columns so:

```
ol1.data[30:35, 1:2]
```

```
##      IQ    Income
## 30 133 112731.55
## 31 124  74743.81
## 32 123 124861.18
## 33 102  66881.54
## 34 111  52850.87
## 35  91  76858.87
```

prints out the 30th to the 35th person on the first two "columns" or variables in the dataset; in this case, IQ and Income.

We can also filter our data based on if people dropped out of the study by telling `R` which column to use for filtering.

```
?subset
new.ol1 = subset(x = ol1.data, DroppedOut == TRUE)
```

Exporting Data

Finally, after we make changes or adjust our data, we can export it out.

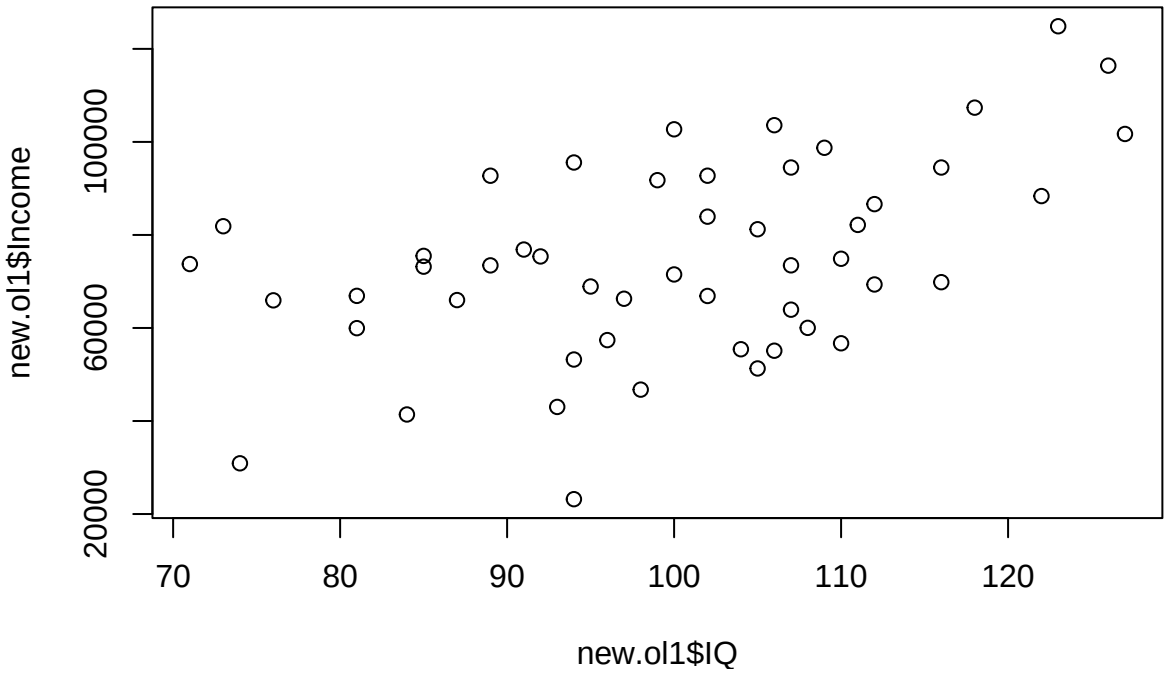
```
write.table(x = new.ol1, file = "C:/Users/MyComputer/Desktop/OL1_DroppedOut.dat", sep = "
```

Your First Plot!

```
plot(IQ, Income)
```

Weird.. that doesn't work. What's the error message?

```
plot(new.ol1$IQ, new.ol1$Income)
```



Knitting `.RMD` Files

To knit your final document you just need to hit the `Knit` button up top. On PC, you can hit `Ctrl + Shift + K` to knit your document while on Mac, you can hit `Cmd + Shift + K`.

Once you begin the knitting process, your document will be run from the very top and checked for any errors or issues and then a final document will be printed out and opened for you!