Data Science Crash Course

Session 1 - Introduction to R and RStudio

Aaron Scherf

June 28, 2019

Contents

Today's New Functions

- `summary()`
- `install.packages()`
- `require()`
- `print()`
- `sqrt()`
- `==`
- `=`
- `<-`
- `ls()`
- `remove()`
- `head()`
- `c()`
- `data.frame()`

Welcome to R!

R and RStudio

The R Project has developed R into an incredibly diverse, open-source statistical programming language that is rapidly becoming a mainstay of data scientists and researchers around the world.

There is a large and growing online community offering tutorials (like this one), user-created packages, and entire user interfaces (like RStudio).

Using RStudio and Running Code

R operates like most programming languages (and is quite similar to Python or Stata, if you've ever used those), in that it has its own grammar and vocabulary based on commands and functions.

Code is usually written in R scripts or R Markdown documents (like this one) and executed from the script or in the command console (found below the script editor).

RStudio offers a traditional navigation tab at the top, with "File" and "Edit" options that should be familiar to anyone that has used a word processor.

R Markdown

This file you are reading is an R Markdown document (Rmd). Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents that is shared by many programs.

We will mostly be using it for simple text and "chunks" of code. For more details on using R Markdown see the RStudio Guide or the Definitive Guide to R Markdown.

Knitting Output Documents

R Markdown is amazing for many reasons, but one of the biggest is that it can automatically create documents and presentations from a script. It does this using a package built into RStudio called `Knitr`.

When you click the **Knit** button (at the top of the session window) a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. For more on `Knitr` check out this Intro Guide.

Embedding Code Chunks

You can embed an R code chunk like as follows (best viewed inside the Rmd script, rather than the presentation slides). In presentations it often appears as a gray box with text inside, often in a different font or color.

```r
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00
```

Running Code Chunks

Anything written inside of the code chunk (between the three back-ticks ', which are usually found below the tilde $\sim$ on a keyboard) can be run as R code by pressing the little green play button at the top right of the chunk (or pressing ctrl-enter for Windows, cmd-enter for Macs).

The play button will run the entire chunk, whereas clicking and pressing ctrl-enter will just run one line (unless you select multiple lines).

Markdown Text

Everything else is normal text that will appear in the output document. You can format the text using Markdown syntax, a common language shared by many programs (like GitHub and Jupyter Notebooks).

Some common Markdown formatting options are titles (made by starting a line with a #, ##, or up to six #'s) and hyperlinks, such as this link to a Markdown cheatsheet.

Go ahead and try running the {r cars} chunk above within the Rmd script in RStudio. A summary table of the example dataset `cars` should appear, showing statistics for two variables, `speed` and `dist`. The `cars` dataset is pre-loaded with RStudio so it's good for introductory examples.

HTML Output Formats

There are a lot of options for output documents. Of the standard styles that come with RStudio, I prefer Slidy presentations for their functionality. Beamer PDFs are also useful if you want a more accessible version. HTML documents have the most flexibility in formatting but are less conducive to presentations, since they don't have that "slideshow" feel.

The format of the Rmd script is very important for making presentations. New slides are defined using the Header 2 format, with two ## followed by the slide title. Hence why the Rmd script is broken into so many sections by headers. After you get the feel for the formatting options, however, making slides or reports is incredibly quick and easy.

Reveal.JS Presentations

Another great option is a **Reveal.JS** presentation, which is the default format for Jupyter Notebook presentations in Python, and therefore great for this cross-platform course to remain consistent.

Check out the Definitive R Markdown Guide for an introduction to Reveal.JS and this quick tutorial by Vincent Bauer on how to make Reveal.JS presentations. The cascading style sheet (css) file that contains the preferences for this presentation was made by Dr. Bauer and is available for download via their Stanford website. I've also included it in every session folder in the GitHub repo for this course.

Installing the revealjs Package

The first step to create a **Reveal.JS** presentation is to install the revealjs package. Packages are user-generated sets of commands and data that can be downloaded easily within the RStudio environment.

The standard function to install pacakages is, naturally, install.packages(). We put it in a fancy if statement to check that the package is not already installed, preventing potential errors when sharing the Rmd file. This is generally good practice but you can just use the basic install.packages() function too.

```r
if (!require(revealjs)) install.packages('revealjs')
```

```
## Loading required package: revealjs
```

library() and require()

After installing a package, make sure that it is loaded in your instance of R by using either the `library()` or `require()` function. They are functionally similar, though `require()` outputs a logical value of `TRUE` if the package is already loaded, making it useful for `if` statements like the one above (more on `if` statements in the next section).

```r
require(revealjs)
```

If this is your first time in R, don't worry too much about the `revealjs` package or Knitting presentations. We just wanted to show how useful RStudio and Rmd files can be, as well as introduce the concept of installing packages.

Commands and Objects

Materialist Programs

R is an object-oriented programming language. This just means that much of our work is done by creating temporary objects out of data like numbers, character strings, files, etc.

This lets us reference the same data multiple times in a script quickly without having to specify the source. Next session, when we import data from a comma-separated value (CSV) file, we only have to import it once and save it as an object, then we can call on the data many times by using the object name.

Running Commands and Reproducibility

You can make objects (or run any code) using the console (similar to the command line, usually at the bottom of the RStudio interface). Most programmers prefer to write code in a script (like this R Markdown document) so that they can run it again or share it with others (the console by itself doesn't “save” your work in a file, though you can see a history of your commands).

Let's get used to running commands inside our script, since this is generally best practice for reproducibility.

Some Example Commands

Here are a few simple commands that can be executed in R.

```r
2+2
```

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

```r
2+2 == 4
```

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

```r
print("hello_world")
```

```
## [1] "hello_world"
```

```r
sqrt(4)
```

```
## [1] 2
```

Types of Commands

As you can see, commands in R can be:

- mathematic (adding, multiplying, etc.)
- logical (testing if things are `TRUE` or `FALSE`)
- functions like `print()` or `sqrt()`
- a combination of these

Much of learning R is knowing different functions and how they fit together. Few people have memorized every possible function, so most of us rely extensively on the "Help" search on the bottom right window.

Help!

If you don't know how to do something, start typing that thing in the help search bar, typically at the bottom right of the RStudio interface. Chances are you'll find the function you need. If not, Google has fantastic resources from the global R-community, including StackOverflow answers to common questions.

About 80% of your programming time, at least at the start, will be spent looking up functions or example code and adapting it to your needs. This isn't stealing or cheating (unless you take code and pretend it's yours) so please borrow liberally from other sources (like these guides)!

Storing and Printing Results

When you execute a command, R will print out the result, as you saw above. You can also use the `print()` function to achieve the same output. This output can also be saved to an object and re-used later by storing it with an assignment operator, often = or `<-`.

Let's try an example to show how we can save a command to an object and re-use it.

Storing Values in Objects

```r
four = 2+2
```

R doesn't print the results when you assign values to an object, unless you call the object (either directly or with `print()`).

```r
four
```

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

```r
print(four)
```

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

Assignment Operators

Note how the = was used in the first line. In this case, `four` is just a string of letters until it is assigned to the output of the function `2+2`.

By writing out `four` and assigning it with the = we are telling R to save the operation `2+2` in a shorthand that we named `four`.

Many people also use `<-` in place of the = as an assignment operator. There is a convincing post on RBloggers about `<-` being better than = for readability purposes, but technically they are identical (as assignment operators)

Using Object Names as Values

Objects are helpful mostly as a “shorthand” for the values they contain. Even though the object `four` takes longer to type than the value it represents, more complex objects (like dataframes) are only accessible when stored. No one wants to re-type a spreadsheet manually.

```
four / 2
```

```
## [1] 2
```

```
sqrt(four)
```

```
## [1] 2
```

```
sqrt(four*four)
```

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

How R Treats Objects

Notice how the output wasn't printed until you called the object `four` by running it by itself. R won't automatically print the contents of an object when you make it.

Also note that you can then use the object in subsequent code, treating it just like you would the contents 2+2 or, as it evaluates the math, `4`.

Not all = are Equal

Note the difference between the single = that is used to assign objects and the == that runs a logical test of equivalency. == checks to see if the surrounding values are equal, and if so prints the logical value `TRUE`. If they are not equal, it will print the logical value `FALSE`.

```r
four == 4
```

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

```r
four == 3
```

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

Computers are Dumb

By itself, unless you previously assign it as an object, `four` means nothing arithmetically to R. We can see this if you try it with another character string.

```
six + 4
```

```
## Error in eval(expr, envir, enclos): object 'six' not fou
```

We receive an error message that the object `six` is not found. R does not know that `six` is English for the value 6 unless you tell it so.

Unless the Operator is Smart

`six` by itself means nothing to R (just a random set of characters), unless you assign it as an object.

```
six = 12 / 2
six + four
```

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

Now R can add the two objects `four` and `six` because you assigned them values.

Silly Objects

If you assign other values to the same name, it will write over the object you made before, so be careful with re-using names.

```
four = 3
six + four
```

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

R won't judge you for making silly objects, like `four` being 3, but anyone you share your code with will, so be sure to use sensible names that would be comprehensible to others.

This isn't required, necessarily, but it's good coding practice and will help make things easier for you and other programmers. We're all in this together, after all.

The Global Environment

Programmers Do Care About the Environment

Whenever you create an object by assigning values to a name, you should see the object in your **Global Environment** window at the top right. Your environment should currently contain all the objects we made previously, with a small preview or description of their values next to their names.

Your global environment keeps track of all the objects you've created within the R session. These objects are stored in your computer's temporary memory, so if you quit the R session (even if you save the script file) the objects won't stay.

Workspace Images and RData

You should get a prompt asking if you want to save the **workspace image** to a **.RData** file. You can save the objects in the environment to this type of file, to make opening and editing a script faster, but generally you don't need to do so, as the script will be able to reproduce the environment if you run it in its entirety.

The whole idea behind code scripts is that you can send them to anyone and (with a few modifications) they can run them from start to finish. This lets them reproduce your code (and results) to verify what you've done or add to it. This is also why we make lots of comments explaining our code.

Clearing the Global Environment

A common way to start all of your R scripts and Markdown documents is with a command to clear the global environment. This clears away any variables (objects) saved to your workspace from previous commands, which can help avoid errors when working with lots of objects.

The command to clear the environment is a bit complex at first so we'll break it down into pieces, starting with the command to list all objects in the environment.

List Objects with `ls()`

The first building block of the command to clear the environment is calling all the objects in your environment, using the **List Objects** command `ls()`:

```r
ls()
```

```
## [1] "four" "six"
```

ls() Output

ls() should print out a list of character strings for all the object names in your environment. We prefer to use the names rather than the values within objects, since some objects can contain thousands (or millions) of values.

Notice that ls() calls a list, not just a single value, so you can use the square bracket to index a specific item. More on indexing later.

```r
ls()[2]
```

```
## [1] "six"
```

Removing All Objects from the Environment

Now that we can call all of our objects in the global environment, you can then clear the environment with the following command:

```r
remove(list = ls())
```

This command may look a bit obtuse at first, but it is pretty intuitive if you break it down. `ls()` calls the objects in your environment, the `list =` portion is an input argument that specifies the objects called by `ls()`, and the `remove()` function removes them.

Note how the "outer" function is `remove()`.

`list =` is an argument being input to that function, while `ls()` is another function that uses the default arguments (and is therefore blank within the parentheses, where arguments are specified).

Empty Environments

If you run the `ls()` command with an empty environment it will report an empty `character(0)` string.

```r
ls()
```

```
## character(0)
```

Calling Objects that Aren't There

The objects we assigned earlier are gone, so if we tried to call `four` R won't know what to do:

```
two
```

```
## Error in eval(expr, envir, enclos): object 'two' not fou
```

See the error message? The object `four` is not found in the environment. Since R is object-oriented, however, you can scroll back up to the chunk that created the object `four`, run that, and then come call it with the code chunk above and it will print the value.

Commands can be run out-of-order, though it's good practice to put them in the "right" order so that you can re-run the script later on.

Removing Individual Objects Pt.1

What if we only wanted to remove certain objects but not others? Let's make two objects and see.

```
two = 2
hello_world = "hello_world"
```

Removing Individual Objects Pt.2

Now we have two objects: `two` (which is numeric) and a new `hello_world` (which is a character string). We can call either using their name.

```
two
```

```
## [1] 2
```

```
hello_world
```

```
## [1] "hello_world"
```

Removing Individual Objects Pt.3

If we just want to remove the `two` object we can use the `remove()` function. We don't need to use the `list =` argument since we are specifying particular objects.

```r
remove(two)
```

Note that R doesn't print any output. It removes the object but doesn't say anything about it, just like it doesn't print any output when you assign the object.

Creating a Dataframe and Indexing

Intro to Dataframes

Most of your analysis will be on pre-existing data. It's rare to actually build a full data set using R, unless you automate it with a function. However, it's helpful to learn about making dataframes to better understand the structure and how the indexing system works.

Dataframes are objects which can contain different types of data (numeric, string, logical, etc.) organized into columns. You can think of an Excel spreadsheet made up of rows and columns, where rows typically represent a single observation, while columns represent variables. Columns are also often called **series**.

Creating Series with `c()`

Series can be saved as objects in R by assigning them to a name with the combine function `c()`.

```r
city_names = c('San Francisco', 'New York City', 'Austin')
population = c(884363, 8623000, 950715)
```

Making a Dataframe with `data.frame()`

Now we have two new objects, which are both groups of three data points. `city_names` contains three text strings, while `population` has numeric values.

These can then be combined into a dataframe using the `data.frame()` function, which orders the two series into two columns, with the position of each data point reflecting its row.

You can think of transposing the two lines above (turning them sidewise) to make a dataframe with 3 rows and 2 columns

Making the City_Data Object

We combine the two objects in a dataframe named City_Data and then view the first few values with head(City_Data).

```r
City_Data = data.frame(city_names,population)
head(City_Data)
```

```
##      city_names population
## 1 San Francisco     884363
## 2 New York City    8623000
## 3        Austin     950715
```

Calling an Entire Dataframe

We can also call the entire dataframe using its name, though browsing values like this is rarely useful for larger datasets. Since we only have 3 rows, however, we can see the entire dataframe, so the results are identical to `head()`.

```
City_Data
```

```
##       city_names population
## 1 San Francisco     884363
## 2 New York City    8623000
## 3        Austin     950715
```

Indexing Dataframes

See how the dataframe is structured? The numbers to the left of the rows indicate the row position of each value. You can call specific rows, columns, or values from within the dataframe using the index position. The simplest way is to indicate the position of the value in the list.

```
City_Data[1]
```

```
##       city_names
## 1 San Francisco
## 2 New York City
## 3        Austin
```

```
City_Data[2]
```

```
##   population
## 1     884363
## 2    8623000
## 3     950715
```

Understanding Indexing Pt.1

Let's break down the index positions above:

- `City_Data[1]` calls the first series (or column), which is identical to the `city_names`.
- `City_Data[2]` calls the second series, which is identical to `population`.

Indexing Continued

You can also index according to the table position (like a cell in Excel) using matrix notation.

```
City_Data[1,]
```

```
##      city_names population
## 1 San Francisco     884363
```

```
City_Data[,1]
```

```
## [1] San Francisco New York City Austin
## Levels: Austin New York City San Francisco
```

```
City_Data[1,1]
```

```
## [1] San Francisco
## Levels: Austin New York City San Francisco
```

Understanding Indexing Pt.2

- `City_Data[1,]` calls the first row, with San Francisco and its population.
- `City_Data[,1]` calls the first column, with all the city names. Notice that the first command, ‘City_Data[1]’ calls the series as an object, while this calls it as a text string.
- `City_Data[1,1]` calls the data value which is in the first row and first column, which is just the text string for “San Francisco”

R vs. Python

Don't worry if you get confused which is which. You can always test it out or look it up. Just knowing how the numerical index system works is the important thing.

Also, fun fact, R starts its index at 1 while Python and some other langauges start at 0. I'm sure there is a reason for this but mostly it's just a funny argument between R and Python programmers.

Show me the $variable

Another common way to index within a dataframe is to call the name of a column using the $ sign. When you construct a dataframe from series, the name of the series by default becomes the name of the column (also referred to as a variable, in typical survey data form).

```
City_Data$city_names
```

```
## [1] San Francisco New York City Austin
## Levels: Austin New York City San Francisco
```

$variables and Indexing

You can also use indexing with the variable call $ to specify the first value in the series `city_names`. This is equivalent to calling the first column and first cell numerically via `[1,1]`.

```
City_Data$city_names[1]
```

```
## [1] San Francisco
## Levels: Austin New York City San Francisco
```

```
City_Data[1,1]
```

```
## [1] San Francisco
## Levels: Austin New York City San Francisco
```

Conclusion and Review

Congrats!

This process of clearing the environment, setting the working directory, and creating objects is the first part of every data analysis project you will do. Congratulations, you're officially doing data science!

Review

- `summary()`
- `install.packages()`
- `require()`
- `print()`
- `sqrt()`
- `==`
- `=`
- `<-`
- `ls()`
- `remove()`
- `head()`
- `c()`
- `data.frame()`

Help is Your Best Friend

If you don’t recognize any of these or what they do please feel free to go back up and review. These are all “bread and butter” commands that you will be using quite a lot, so make sure to know what they are.

If you want to explore them in even more detail you can also look them up in the **Help** files search to the right. The help-file for each function will give you a description, list of possible arguments and their default values, and some example code. It’s always good to check the help-file whenever you are using a new function!

Next Time on the **Crash Course**

Again, welcome to the R community! Our next lesson will focus on importing data and generating summary statistics!