Introduction to R for Natural Resource Scientists

Ben Staton

with contributions from Henry Hershey

Contents

Overview

This is intended to be a first course in R programming. It is by no means comprehensive, but instead attempts to introduce the main topics needed to get a beginner up and running with applying R to their own work. There will be no prior knowledge assumed on the part of the student regarding R or programming. In the later chapters, e.g., Chapters 13 and 14, an understanding of statistics at the introductory undergraduate level would be helpful.

Chapter 1

Introduction to the R Environment

1.0.1 Instructor: Ben Staton

Session Overview

In this first session, we will get familiar with the basics of using R. We will cover:

- the use of R as a basic calculator
- object types
- data classes
- data structures
- how to read in data
- how to write out data
- how to write your own functions

1.1 Getting Started: Install R and RStudio

First off, you will need to get R and RStudio[1] onto your computer. Please see **Appendix A** for details on installing these programs on your operating system.

1.2 The R Studio Interface

Once you open up R Studio for the first time, you will see three panes: the left hand side is the **console** where results from executed commands are printed, and the two panes on the right are for additional information to help you code more efficiently - don't worry too much about what these are at the moment. For now, focus your attention on the console.

[1] While it is possible to run R on it's own, it rather clunky and you are strongly advised to use RStudio given its compactness, neat features, code tools (like syntax and parentheses highlighting). This workshop will assume you are using RStudio

1.2.1 Write Some Simple Code

To start off, we will just use R as a calculator. Type these commands one at a time and hit **CTRL + Enter** to run it. The spaces don't matter at all, they are used here for clairity and for styling.

```
3 + 3
```

```
## [1] 6
```

```
2 - 3
```

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

```
3 * 12
```

```
## [1] 36
```

```
8 / 2
```

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

```
2 ^ 3
```

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

Notice that when you run each line, it prints the command and the output to the console. The format I'm using prints the output with `##` in front of each line, but that will not happen on your computer screen.

R is an **object oriented language**, which means that you fill objects with data do things with them. Let's make some objects and call them `a`, `b`, and `c`.

```
a = 3 + 3
a
```

```
## [1] 6
```

```
b <- 5 + 2
b
```

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

```
c = a + b
```

Notice that running the first line did not return a value to the console. This is because in that line you are **assigning** a value. You can view your object by typing its name alone and running it (or by double clicking the name where you assigned it to highlight it, which is much faster). The `=` sign or `<-` can both be used to assign values to objects. This is a matter of personal preference: I often see hardcore programmers use the `<-`, but I prefer the `=` because it is only one key as opposed to three. You can run multiple lines at a time by highlighting them before running. You may also run portions of a line at a time by highlighting only the desired portion. You can run a single entire line by simply placing your cursor on that line and hitting CTRL + R. CTRL + ENTER will run code as well (unless you are on a Mac, in which case I think you switch out the CTRL for CMD).

Chapter 2

Saving Scripts and the Working Directory

You will want to save your hard work. But first, let's talk about the **working directory**. The working directory is a folder on your computer that R looks for files by default. Working directories become very useful when bringing in datasets and saving output so you should get used to using them now. To choose the working directory, go to Session > Set Working Directory > Choose Directory. I recommend having distinct working directories for different tasks. For example, for a class with several lab assignments, I would have a folder for each assignment that serves as the working directory for that assignment and stores my script(s) and data set(s). Once your working directory is set, go to File > Save. The working directory opens up automatically now and you can call your script whatever you would like. It will be saved as an .R file, which is essentially a text file.

Chapter 3

R Object and Data Types

R has a variety of object types that you will need to become familiar with. The primary types for storing data are **vectors**, **matrices**, and **data frames**. There are others (e.g., lists, arrays, tables, etc.) but we will not go into those just yet for simplicity's sake. **Vectors** are a list of values (numeric or character), in only one dimension, and are thus the simplest of the object types. Above, we made `a`, which was a numeric vector with one element. **Matrices** are values in two dimensions, and can be thought of as rows or columns of combined vectors. Vectors and matrices hold values of all the same type, for example, a vector can be a character vector where all the elements are words or letters or it can be numeric, where they are all numbers. The same goes for matrices. **Data frames**, by contrast, can have multiple data types between different columns. Every row and column of matrices and data frames must have the same number of elements (length). We will use all three of these object types extensively in this workshop.

Functions are built-in objects in R that perform specific tasks. For example, there is a `mean` function that calculates the mean of a vector of numbers. There are tons of functions in R that range in their complexity. We will start with basic functions (like `mean`, `sd`, `length`, etc.) and build up from there. One part about R that makes it really versatile and cool is that you can write your own functions to carry out your own specific tasks. We will cover this topic in more detail later.

It is very important to know what kind of object type you are using, since R treats them differently. For example, some functions can only use a certain object type. The same holds true for data types (numeric vs. character vs. factor). You will get some very interesting results if you attempt to take the mean of a factor (i.e., a categorical number or character). Note that you can quickly determine what kind of object you are dealing with by using the `class` function. Simply run `class(object.name)`.

Chapter 4

Making Some Data

We will begin by making some vectors. The way you do this is by using the `c()` function. The "c" stands for "concatenate" or combine some values into one object.

```
num = c(4, 7, 8, 10, 15)
ponds = c("F11", "S28", "S30", "S8", 'S11')
```

There are shortcuts to make patterns of numbers:

```
2010:2014
```

```
## [1] 2010 2011 2012 2013 2014
```

```
seq(from = 2001, to = 2005, by = 1)
```

```
## [1] 2001 2002 2003 2004 2005
```

```
seq(2001, 2005, 1)
```

```
## [1] 2001 2002 2003 2004 2005
```

```
rep(10, 5)
```

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

Notice that lines 2 and 3 above do the exact same thing. Line 2 uses named arguments. An **argument** is a command you give to a function to tell it what to do. All functions have arguments, and they all have names. By specifying the names within the function, you don't have to remember what order the arguments go in (i.e., `seq(to=5, from=1, by=1)` is the same as `seq(1, 5, 1)` and they are both different from `seq(1,1,5)`).

R has lots of information to help you learn how to use a function. Let's look at the help file for the mean function. Type `help(mean)`. The help file tells you what goes into a function and what comes out. For more complex functions it also tells you what all of the options (i.e., arguments) can do. Now let us use the mean function.

```
mean(num)
```

```
## [1] 8.8
```

A quick aside. One of the really nice things about R is that it can do vectorized calculations. This means that if you tell it to, it will do something to every element in the vector. For example, if we wanted to add one number to every element of the vector `num` and call the result `num2`, then we would just type:

```
num2 = num+1
num2
```

```
## [1]  5  8  9 11 16
```

Similarly, if we add two vectors, it will take the first element of the first vector and add it to the first element of the second vector and so on and so forth. This works for adding, subtracting, multiplying, dividing, taking exponents, etc. Note that the two vectors must have the same length.

Okay, now let us make some matrices. The way I like to make matrices by hand is by combining several vectors (again of the same length). There are two functions that allow us to do this: `rbind` (by row) and `cbind` (by column).

```
m1 = cbind(num, num2, 2010:2014)
m1
```

```
##      num num2
## [1,]   4    5 2010
## [2,]   7    8 2011
## [3,]   8    9 2012
## [4,]  10   11 2013
## [5,]  15   16 2014
```

```
m2 = rbind(num, num2, 2010:2014)
m2
```

```
##      [,1] [,2] [,3] [,4] [,5]
## num     4    7    8   10   15
## num2    5    8    9   11   16
##      2010 2011 2012 2013 2014
```

Note that when we used named objects (`num`, `num2`) it kept the names. You can return these names by using `colnames` to get column names and `rownames` to get row names. If it returns `NULL`, then there are no row or column names.

Now let's look at a point I mentioned earlier. Remember how all of the elements in a matrix (and vectors) must be of the same type (e.g., character vs. numeric)? Let's try to combine a character vector and a numeric vector into a matrix:

```
cbind(ponds, num)
```

```
##      ponds num
## [1,] "F11" "4"
## [2,] "S28" "7"
## [3,] "S30" "8"
## [4,] "S8"  "10"
## [5,] "S11" "15"
```

The quotes mean that R is now treating them all as characters. It **coerced** (or turned) the numbers into characters. Is this what we wanted? Probably not because once things are characters, you can no longer do math on them. To combine vectors of data that have different types we will need to use a data frame.

```
df1 = data.frame(ponds, years = 2010:2014, num)
df1
```

```
##   ponds years num
## 1   F11  2010   4
## 2   S28  2011   7
## 3   S30  2012   8
## 4    S8  2013  10
## 5   S11  2014  15
```

Data frames are what we usually use when analyzing data because of this property.

Chapter 5

Retrieving data from objects

This is one of the most important and versatile skills to know in R. So we have an object with data in it and we want to use it for analysis. But we don't want the whole dataset, we just want one column. How do we pull out that one column? Another term for pulling out only data we want is *query*. Well it turns out there are two main ways to query data.

1. Indexing. This method allows you to pull out specific columns/rows by their location in an object. However, you must know exactly where in the object the desired data are. To index, you specify the object, then what rows, then what columns. The syntax is: `object[row, column]`. For example, let's pull out the first row of the data frame `df1` using indexing:

```
df1[1,]
```

```
##   ponds years num
## 1   F11  2010   4
```

This works on matrices as well. Here are some examples:

```
m1[1, 2]          # pulls out the element in the first row, second column from m1
```

```
## num2
##    5
```

```
m1[, 2]           # pulls out the whole second column (every row) from m1
```

```
## [1]  5  8  9 11 16
```

```
m2[, 2:4]         # pulls out columns 2, 3, and 4 from m2
```

```
##      [,1] [,2] [,3]
## num     7    8   10
## num2    8    9   11
##      2011 2012 2013
```

```
m1[c(1, 2, 4), ]  # pulls out the 1st, 2nd, and 4th, rows, with every column from m1
```

```
##      num num2
## [1,]   4    5 2010
## [2,]   7    8 2011
## [3,]  10   11 2013
```

Vectors are done the same way, except you only provide one number because there is only one dimension. For example: `num[3]` pulls out the 3rd element of the vector `num`.

2. By name – This method allows you to pull out a specific column of data based on what the column name is. Of course, the column must have a name first. The name method uses the $ operator:

```
df1$num
```

```
## [1]  4  7  8 10 15
```

You can combine these two methods:

```
df1$num[3]
```

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

```
# or
df1[, "years"]
```

```
## [1] 2010 2011 2012 2013 2014
```

The by name ($) method is useful because it can be used to add columns to a data frame:

```
df1$num3 = seq(1, 10, 2)
df1
```

```
##   ponds years num num3
## 1   F11  2010   4    1
## 2   S28  2011   7    3
## 3   S30  2012   8    5
## 4    S8  2013  10    7
## 5   S11  2014  15    9
```

Both are very useful, and we will be making frequent use of these methods for the rest of the workshop. The $ operator only works on data frames, not matrices. I'll mention briefly that there is another way to query data: logically. We will cover this in the next section. Now go and try your hand at applying these skills in **EXERCISE 1**.

Chapter 6

Bringing in Your Data

It is rare that we will enter our data by hand as we did in Exercise 1. Often, we have a dataset that we wish to analyze or manipulate. R has several ways to bring in data and in this workshop, we will be using the most common and flexible approach: reading in .csv files. CSV files are data files that separate columns with commas. If your data are in an Excel spreadsheet, save your Excel file as a CSV file (File > Save As > Save as Type> CSV (Comma Delimited)). Excel will give lots of dialog boxes asking if you are sure you want to save it as a CSV (you do) and if you wish to save it. A few things to note:

- No cells can contain the "#VALUE!" or "NUM" result that often happens when the result of an Excel function is invalid. If one of these is found in a column, R will treat that *entire* column as a factor (because it contains characters), which causes problems when you do not want that column to be a factor (remember a factor is a grouping variable, like treatment, location, or species).
- If a cell in the first row with text contains a space between two words, R will insert a `"."` between the words.
- R can deal with missing values by turning them into `NA`. However, everywhere you want R to have an `NA`, those cells must contain the same information in the CSV file. These can be completely empty cells, or you could put a `"."` into every cell that you want R to treat as an `NA`. If you use "`.`" to signify `NA` in your CSV file, include `na.strings = "."` as an argument to the `read.csv()` function described below. Of course, you can also simply type "NA" in the CSV file wherever NA's occur.
- If your data have column names (i.e., characters) in the first row, R will bring those in as column names by default.
- R brings in CSV files in as data frames by default.

The website I'm using won't let me upload CSV files, so we will have to convert Excel workbooks (.xlsx) to CSV files. Once that is done, to bring (read) in a CSV file we use the function `read.csv()`. `read.csv()` has several arguments (look at the help file for details), but only one is really necessary: the file name (or location). Let's convert and read in the provided dataset "streams.xlsx" (there are no problems with this dataset, so don't worry):

```
dat = read.csv("streams.csv")
```

If the data set is in your working directory, all you need to provide is the file name (remember case, quotes, and .csv at the end!). If the file is *not* in your working directory, you must type (or copy/paste) the *full* file path. Note that R requires the use of `/` and not `\` in file paths. For example:

```
dat = read.csv("C:/Users/Ben/Desktop/Auburn R Workshops/Session 1/Session 1 MarkDown/Session-1-data.csv
```

If you did not get any errors, then the data are in the object you named and that object is a data frame. If you get errors (e.g., `...No such file or directory`) then make sure you have spelled everything correctly,

used the right case and symbols, and have included a ".csv" at the end of the file name. Don't worry, this is probably one of the most common errors I get in R.

There is a neat trick to simplify this. Running `read.csv(file.choose())` pulls up a window that allows you to interactively choose the file you want. (running `file.choose` returns the file path of the file you select, so it is really just the same as typing in the file path, but easier). I don't use this for complex analyses, because when I come back to that script later, I want to make sure I'm using the right data file. Typing the file name instead of using `file.choose`helps me with this.

Let's look at the data. We could just run the object, but it will show the whole thing, which may be undesirable if the dataset is large. To view the first six rows, run `head(dat)` or the last six rows with `tail(dat)`.

Okay, now we will use some basic functions to explore our data before any analysis. The `summary` function is very useful, and it can be used on a large variety of different objects.

```
summary(dat)
```

We can now see the spread of our numeric data and see the different levels of our factor (`state`) as well as how many data points belong to each category. Note that we have one `NA` in our `flow` variable.

To count the number of elements in a variable (or any vector), we use the `length` function:

```
length(dat$stream_width)
```

Note that R counts missing values as elements as well:

```
length(dat$flow)
```

To get the dimensions of an object with more than one dimension (i.e., a data frame, matrix, or array) we can use the `dim` function. This returns a vector of length 2: the first number is the number of rows and the second is the number of columns. If you only want one of these, use the `nrow` or `ncol` functions (but remember, only for objects with more than one dimension; vectors don't have rows or columns!).

```
dim(dat)
```

Let's calculate the mean of all of our flow data:

```
mean(dat$flow)
```

It returned an `NA` because there is an `NA` in this variable. There is a way to tell R to ignore this `NA`. Include the argument `na.rm = TRUE` in the `mean` function (separate arguments are always separated by commas). This is a **logical** argument, meaning that it asks a question. Do you want to remove `NAs` before calculating the mean? `TRUE` means yes and `FALSE` means no. These can be abbreviated as `T` or `F`. Lots of R's functions have the `na.rm` argument (e.g. `mean`, `sd`, `var`, `min`, `max`, `sum`, etc.).

```
mean(dat$flow, na.rm = T)
```

What if we want to do something to more than one variable at a time? The best way to do this (as with lots of things in R, there are many) is by using the `apply` function. This applies the same function to a subset of data. Note that we must specify the `na.rm` argument to the `var` function within the `apply` function:

```
apply(dat[,2:3], 2, FUN = var, na.rm = T)
```

The first argument is the data we want to apply the function to. The second argument (the number `2`) specifies that we want to apply the function to columns, `1` would tell R to apply it to rows. The `FUN` argument specifies what function we wish to apply; here we are calculating the variance. There is a whole family of `apply` functions, the base `apply` is the most basic but a more sophisticated one is `tapply`, which applies a function based on some grouping variable. Let's calculate the mean stream width for every state:

```
tapply(dat$stream_width, dat$state, mean)
```

The first argument is the data you want to apply the `mean` function to, the second is the grouping variable, and the third is what function you wish to apply.

Chapter 7

Logic, `if`, `else`, and `ifelse`

R is a very logical language, both in a literal a figurative sense. When we say "logical", as alluded to before, we are really talking about a question. You will also see these referred to as **conditionals**. Which values in a vector meet some condition (greater than some number, are equal to some number, have the same characters, etc.)? Let's start with some very basic R logic (note that reassigning a value to `num` will write over what it was before, be careful with this):

```
num = 5   # assign a value to the object num
num < 5   # ask if it is less than 5
```

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

```
num <= 5  # ask if it is less than or equal to 5
```

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

```
num == 5  # ask if it is exactly equal to 5
```

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

Each of these lines (after the first which assigns a value to value) asks a question: does `num` meet the specified condition? It returns `TRUE` if it does, and `FALSE` if it doesn't. Note that the logical equals is specified by using `==`.

We can tell R to do something if the result of our logical question is `TRUE`. This is a typical if-then statement.

```
if (num == 5) print("num is equal to 5")
```

```
## [1] "num is equal to 5"
```

This says "if `num` equals 5, then print the phrase 'num is equal to 5' to the console". If the logical returns a `FALSE`, then this command does nothing:

```
if (num != 5) print("num is equal to 5")
```

The `!` is the logical *not*. It asks the question "is `num` not equal to 5?". We can tell R to do multiple things if the logical is `TRUE` by using curly braces:

```
if (num == 5) {
  print("num is equal to 5")
  print("num is supposed to be 6")
}
```

```
## [1] "num is equal to 5"
## [1] "num is supposed to be 6"
```

But what if we want R to do something if the logical is FALSE? Then we use the else command:

```
if (num > 5) print("num is greater than 5") else print("num is not greater than 5")
```

```
## [1] "num is not greater than 5"
```

Or on two lines:

```
if (num > 5) {
  print("num is greater than 5")
} else print("num is not greater than 5")
```

```
## [1] "num is not greater than 5"
```

The if function is very useful, but it can only ask one question at a time. If you supply it with a vector of length greater than 1, it will give a warning that only the first value is used. (Warnings are different than errors in that something still happens, but it tells you that it might not be what you wanted, whereas an error stops R altogether.) To ask multiple questions at once, we must use ifelse. This function is similar, but it combines the syntax into one function:

```
nums = c(-5:-1, 1:5)
ifelse(nums > 0, "positive", "negative")
```

```
##  [1] "negative" "negative" "negative" "negative" "negative" "positive"
##  [7] "positive" "positive" "positive" "positive"
```

The syntax is ifelse(logical condition, do if TRUE, do if FALSE). Let's use ifelse to create a new variable in dat that tells us if each stream is big or small depending on some cut-off we choose:

```
dat$size = ifelse(dat$stream_width > 50, "big", "small")
```

This says "make a new variable in the data frame dat called size and assign each row a 'big' if stream_width is greater than 50 or 'small' if less than 50".

One neat thing about ifelse is that you can nest multiple statements inside another (this is true of ALL R functions, by the way). What if we wanted three categories: small, medium, and large?

```
dat$size_fine = ifelse(dat$stream_width <= 40, "small",
                       ifelse(dat$stream_width > 40 & dat$stream_width <= 70, "medium", "big"))
```

If the value is TRUE, then it will give it a "small". If not, it will start another ifelse to ask if the value is greater than 50 *and* less than or equal to 70. If so, it will give it a "medium", if not it will get a "big". This is confusing at first, I know. Not all nesting or embedding of functions are this complex, but this is a neat example. Without ifelse, you would have to use as many if statements as there are data points, and do you really want to do that? I'll note that in addition to the logical *and* (&) there is a logical *or*, denoted by the bar symbol (|).

Chapter 8

Logical Subsetting

This is my favorite way of subsetting (or querying or retrieving data, however you want to say it) an object. It pulls out all of the elements of an object for which the test is `TRUE`. For example:

```
dat$flow[dat$stream_width > 60]
```

Gives all of the flow values for which stream width is greater than 60. Or to see all of the data from Alabama:

```
dat[dat$state == "Alabama",]
```

The `subset()` function does this same thing:

```
subset(dat, state == "Alabama")
```

The syntax is `subset(object, condition)`. You can specify multiple conditions:

```
big.TN.FL.rivers = subset(dat, dat$size_fine == "big" &
              (dat$state == "Tennessee" | dat$state == "Florida"))
big.TN.FL.rivers
```

This gives all rivers that are big and are in Tennessee or Florida. Note that to do big and either in Tennessee or Florida, the states need to be in parentheses because this last part is its own question.

Chapter 9

Writing Output Files

Okay so now that we have made some new variables in our data frame, we want to save this work in the form of a new CSV file. To do this, we can use the `write.csv` function:

```
write.csv(dat, "updated_output.csv", row.names = F)
```

The first argument is the data frame (or matrix) to write, the second is what you want to call it, and `row.names = F` tells R to not include the row names (because here they are just numbers in this case). R puts the file in your working directory by default. To put it somewhere else, type in the path with the new file name at the end.

Chapter 10

Basic User-Defined Functions

Sometimes we want R to carry out a specific task, but there is no function built in to do it. In this case, we can write our own functions. This is one of the coolest parts of R, and I will only introduce it here. We will be going into more detail on this topic in later sessions.

First, we must think of a name for our function. Then, we specify the function using the `function` function. Then, in parentheses, we specify any arguments that we want to use within the function to carry out the specific task. Open and closed curly braces specify the start and end of our function. Let's write a general function to take any number to any power:

```
power = function(x, y){
  x^y
}
```

Now let's test our function:

```
power(5, 3)
```

```
## [1] 125
```

Remember, we can embed functions:

```
power(power(5,2),2)
```

```
## [1] 625
```

This is the equivalent of $(5^2)^2$.

Chapter 11

That's All! (for now)

So there are the basics of what you need to know to get started in R! I hope you found this useful. I know there is a lot of information here but once you get the hang of it, this stuff will seem trivial. When I first learned this introductory material, I thought I would never be able to remember all the rules and syntax but the more you implement these skills, the less you need to look at reference material like this workbook and help files and the more you see how truly versatile this amazing program is. Keep in mind that there are *tons* of resources out there to help you learn R (see the website). I can't tell you how many times typing out a word-for-word question into Google helped me get past a dead end.

Now go try out **EXERCISE 2** (this one's a bit trickier!).

Chapter 12

Base R Plotting Basics

Chapter 13

Basic Statistics

Chapter 14

Simulation and Randomization

Chapter 15

Large Data Manipulation