R\_BootCamp\_D1\_AM

Justin French

2024-08-19

## Hello R!

R is an **object-oriented** programming language, which means you make objects that are stored in memory, then do things to them with functions. To really understand this, we will start by making and describing the various kinds of objects that R has to offer. As we go, we will introduce programming concepts in R that really only make sense when you see them applied to objects.

## Vectors

The most fundamental R object is the **vector**. This is just an ordered set of values. You make a vector like this:

vec <- c(7, 8, 9)

The simple line of code that we ust ran did several things. First, it tells R that we are going to make an object called vec. Next, we gave R and **assignment statement** (the <-), which tells R that whatever comes next needs to be stored in the object vec. Finally, we used the c() function to concatenate (group together) the values 7, 8, and 9 into a vector.

A few important things about vectors: First, they have positions that are occupied by values. You can index these positions to retrieve values, modify them, or subset them. For example, if we want to retrieve the value in the second position of vec, we would do this:

vec[2]

## [1] 8

It’s important to remember that the index position of a vector is completely independent of the value that is stored there. **Indexing** is a key con cept in object-oriented programming because it gives you a repeatable means of manipulating objects. Any time that you index an object, you will open square brackets ([]) immediately after the object name.

The second important thing to know about vectors is that each element within them must by of the same **data type**, meaning that you can store numbers, or text, or logical values (e.g. TRUE and FALSE), but you cannot mix those types within one vector. For example:

vec2 <- c("a", 'b', 3)

If you mix data types, like we just did, R will **coerce** all the elements of the vector to the same type, if it can figure out how to do so. It generally, tries to force the elements to the most flexible data type present. In the case above, it forced the numeric 3 to be text, because the "a" and 'b' could not be turned into numbers. R will show you the data type of a vector’s elements in the environment window. Right now, the vec object is shows num which means *numeric*, and vec2 shows chr for *character*.

Now let’s modify some vectors. To get started, let’s replace our "3" in vec2 with a word instead of a number:

vec2[3] <- "Arrakis"

What if we wanted to change the data type? Could turn vec into a set of characters instead of numbers? Why yes we can, however, we aren’t going to use indexing to do this one. Instead, we will introduce the concept of an **object method**, which is a function that does a specific task with a certain object. While we aren’t really to using functions yet, this is very important for objects. To change the data type of vec to character instead of numeric, we will use the as.character() method:

vec <- as.character(vec)

In the code above, we used the as.character() method, which turned our numbers to text, to make a *new* vector, which was then store in the vec object. This is called *modifying an object in place* and is a very common technique. It has the effect of changing the object, but it really just makes an entirely new one.

One final thing on vectors: they have a method to retrieve their length, cryptically called length(). A vector only has one dimension, so that is the only method you need to determine how many elements are in it. You use it like this:

length(vec2)

## [1] 3

## Matrices

So, what do you do if your data have more than 1 dimension? A collection of vectors will not solve your problem. We often record multiple measurements of things, or need an array of numbers, text, etc, to contain the data we collect. The next step up from a vector is a **matrix**, which has 2 dimensions.

In all other respects, it is just like a vector. All of its elements must be of the same type. That means that a matrix is not an analog to a spreadsheet. However, statistics uses a LOT of linear algebra, so R is built with that in mind. Thus, understanding matrices is very important. Here’s how to make one:

mat <-   
 matrix(data = c(1,2,3,  
 4,5,6,  
 7,8,9),  
 nrow = 3,  
 byrow = T)

Breaking that down, we made an object called mat and assigned the matrix to it. We then told the function what data we want in the matrix by giving it a vector of values. We then had to tell it how many rows we wanted (could have told it columns), then we had to tell it how to place the values in the matrix. This is the minimum amount of information R needs to build a matrix.

Indexing a matrix is not much more complicated than indexing a vector. It just has two dimensions instead of 1, so it requires coordinates to retrieve elements. For example, let’s get the element in the first row and third column of mat:

# R C  
 mat[1, 3]

## [1] 3

Always remember that the indices are coordinates, so they must be separated by a comma. R is purpose built to do linear algebra, so the convention is the same as that; row first, column second. *Always*.

That does not mean that you will always specify both the row and the column. Sometimes you want to retrieve a whole column, or a whole row. You can do this like this:

# Whole row  
 mat[2,]

## [1] 4 5 6

# Whole column  
 mat[,2]

## [1] 2 5 8

You can also retrieve submatrices by specifying ranges within each index. For example:

mat[2:3,1:2]

## [,1] [,2]  
## [1,] 4 5  
## [2,] 7 8

The : operator gives a range of numbers from whatever is on the left to whatever is on the right. It is very useful when indexing over large sections of an object, provided they are continuous sections.

You can also get discontinuous subsets of objects (including, but not limited to matrices) by specifying an explicit vector instead of a range. For example, if I wanted to get just the 1st and 3rd columns of mat, I would do this:

mat[,c(1,3)]

## [,1] [,2]  
## [1,] 1 3  
## [2,] 4 6  
## [3,] 7 9

You have to be careful if you try to get fancy with discontinuous subsets. Sometimes you don’t get the behavior you want. If I want the 1st and 2nd columns of the first and third rows and I do the following, I get this:

mat[c(1,3),c(1,2)]

## [,1] [,2]  
## [1,] 1 2  
## [2,] 7 8

Never mind, I got exactly what I wanted. However, in older versions of R, this would give screwy results. If you work on someone else’s computer and this creates problems, vcheck their R version. But hey, at least there is a landmine you don’t have to worry about!

You can, of course, modify the elements you select in place this way, but it is no different than what we did with vectors. If you want a nice challenge, replace the first 2 elements of the 3rd row with the 1st and 3rd elements of the 2nd column. I’ll leave that to you as an exercise.

One thing we did not look at with vectors, but applies to both vecotrs and matrices, is the idea of **vectorized operations**. This means that one task can be performed to all elements of a vector or matrix simultaneously. For example, I can multiply every element of mat by 4 in one line of code:

mat \* 4

## [,1] [,2] [,3]  
## [1,] 4 8 12  
## [2,] 16 20 24  
## [3,] 28 32 36

Vectorized operations are one of the keys to R’s efficiency as a data manipulation and analysis platform; you can do incredibly large numbers of calculations or data operations simultaneously with a single line of code. This will come up over and over again throughout the rest of the boot camp.

Matrices are not really an everyday-use object to most R users. They are mostly found “under the hood”. Most R users want something analogous to a spreadsheet, where measurements of different variables can be stored as columns. Matrices are not suited to this because they force all elements to be of the same data type. This is rare in practice.

## Data Frames

The most common way you will interact with data is in a **data frame**. Data frames are much more flexible than vectors because they allow each *column* to be of its own data type. It is fundamentally built around the idea of each column representing a variable, and each row representing an object on which that variable was measured.

It is not common for you to manually make a dataframe, but it’s handy to know how. More often, you will import a data frame from a file, such as a .csv. While you can load files from excel, it is the devil and you should never, ever, ever, ever use it. You make a data frame like this:

dat.fun <-  
 data.frame(name = c('Justin','Alex',  
 'Eliana','Caitlin',  
 'Matt','Curtis'),  
 shoeSize = c(12,9,8,8,11,11),  
 age = c(33,35,20,29,27,28),  
 likeCats = c(FALSE,TRUE,TRUE,  
 TRUE,FALSE,FALSE))  
  
  
 # Take a look!  
 # View(dat.fun)

Data frames have 2 dimensions, just like a matrix, and their indexing format is exactly the same. However, data frames have an **attribute** called colnames, which stands for column names. These allow you to select columns by name like this:

dat.fun$shoeSize

## [1] 12 9 8 8 11 11

You can also pull multiple columns by name, but the syntax is a little more complicated:

dat.fun[,c("name","likeCats")]

## name likeCats  
## 1 Justin FALSE  
## 2 Alex TRUE  
## 3 Eliana TRUE  
## 4 Caitlin TRUE  
## 5 Matt FALSE  
## 6 Curtis FALSE

This is a blend of the index-based subsetting and the named subsetting. I fed a vector of column names to the column position of the index. This can be used for pulling or modifying a subset, but cannot be used any other way.

That might sound a little confusing, but we’ll demonstrate with an example. Often, you want to drop a column from a data set. Sometimes you want to drop multiple columns. In a surprising variety of cases, you don’t know what all the columns are, or where they are in the object, so it’s often nice to drop them by name, rather than keep a bunch of columns by name or position.

# Drop second column  
 dat.fun[,-c(2)] # Works  
  
 dat.fun[,- c('shoeSize')] # No Works

In the code above, the - operator allows us to drop columns by index position, but not by name. We will see how to do this by name a little later when we talk about functions.

There are a few method functions that we need for data frames though. We often need to retrieve the column names, which we can do with the colnames() method:

colnames(dat.fun)

## [1] "name" "shoeSize" "age" "likeCats"

We also often want to know how many rows or columns a data frame has. We would use nrow() or ncol() to get that information (works for matrices too):

nrow(dat.fun)

## [1] 6

ncol(dat.fun)

## [1] 4

Sometimes the data type of a variable in a data frame is not obvious. There is a method to check this that works on vectors, but is really handy for data frames. It is called typeof(). We can run this on the columns of a data frame like this:

typeof(dat.fun$name)

## [1] "character"

Often we want to write data frames out of R to a file. This is done with the write.csv() method, like so:

write.csv(dat.fun, 'funData.csv',  
 row.names = FALSE)

Even more common that writing a data frame, we usually read them in from a file. Now that we’ve made a file, we can read one in with the read.csv() method:

dat.fun <- read.csv('funData.csv')

## List

A **list**  is an interesting object type. It actually holds no data of its own. Instead, a list is a collection of objects. It’s kind of a meta-object. A vector and a matrix hold sets values of a given data type, and a data frame holds columns of given types. However, a list is type agnostic.

In many, many cases, the output of a function requires data to be returned in different formats. Some outputs need be vectors, while others may be matrices or data frames. You can think of a list as a shoebox to pack all these other objects into. It is a container of assorted objects.

For example, we could make a list that holds all of the objects we have made so far, like this:

myStuff <-  
 list(vec = vec,  
 vec2 = vec2,  
 mat = mat,  
 dat.fun = dat.fun)

Lists have named elements, ust like data frames have named columns. You can look at these names with the names() method:

names(myStuff)

## [1] "vec" "vec2" "mat" "dat.fun"

We can call obects out of the list by name in exactly the same way we called columns out of a data frame:

myStuff$vec

## [1] "7" "8" "9"

You can even call nested things out by name. For example, our list has named elements and so does our data frame. We can call a specific column of our data frame like this:

myStuff$dat.fun$age

## [1] 33 35 20 29 27 28

This is very common with the outputs of R functions. The $ operator is very handy for exploring what a list contains.

Lists have one dimension, just like a vector. They could be considered a vector with objects for elements even. However, you need a syntax that differentiates a list from a vector when indexing. For this reason, indexing a list requires *two* sets of square brackets. Except when it doesn’t. This is the most confusing thing about any base R obect. We’ll break down with some examples.

If I want the 3rd element of a list, all I have to do is this:

myStuff[[3]]

## [,1] [,2] [,3]  
## [1,] 1 2 3  
## [2,] 4 5 6  
## [3,] 7 8 9

If I want a single element by name, I do what we saw above with the $, but if I want multiple elements by name, I do this:

myStuff[c("vec","vec2")]

## $vec  
## [1] "7" "8" "9"  
##   
## $vec2  
## [1] "a" "b" "Arrakis"

Notice, if I try to give it multiple named elements, I have to use *single* square brackets. I can pull a continuous range of elements like this:

myStuff[1:3]

## $vec  
## [1] "7" "8" "9"  
##   
## $vec2  
## [1] "a" "b" "Arrakis"  
##   
## $mat  
## [,1] [,2] [,3]  
## [1,] 1 2 3  
## [2,] 4 5 6  
## [3,] 7 8 9

But it won’t run with double square brackets. This is the thing that generally costs me time when I’m programming; remembering when to use which index syntax with a list.

Lists are extremely common because the vast majority of R functions give the user a list as an output. Lists can be assigned a **class**, which means they have a specific structure that is repeated no matter what the exact structure or values of the inputs are. This how R method functions know what to do with particular objects. We will many examples of this as we go, but you can check the class of an object with the class() function:

class(dat.fun)

## [1] "data.frame"

typeof(dat.fun)

## [1] "list"