R Session 3: Loops and Writing Your Own Functions

ESS 211

Contents

1 Companion Material **1**

2 Previous Session Recap **1**

3 Loops: repeating code without repeating code **2**

3.1 `For`-loops: repeating a fixed number of times 2

3.2 `While`-loops: Keep going until 3

4 Other data structures **4**

4.1 Matrices 4

4.2 Dataframes (written as `data.frame`) 6

4.3 Lists 9

1 Companion Material

DataCamp Intermediate R, Chapter 2 (Loops): all DataCamp Intermediate R, Chapter 3 (Functions): "Writing Functions" through "Different ways to load a package"

2 Previous Session Recap

At this point, you should be comfortable with

1. Using the `c()` function to create an arbitrary vector, the `rep()` function to create a vector with some repeating pattern, and `seq()` to create a sequential vector.
2. Retrieving and/or reassigning vector elements using either numeric indices, or logical vectors.
3. Applying arithmetic and logical operators on vectors, and understanding the result.
4. Calling a function (such as `rep()`) with the appropriate arguments, and using R's help feature to
    - know which arguments are required vs. optional
    - understand what each relevant argument does
    - know what the function returns

Go back to the previous sesson material and Datacamp exercises as needed!

3 Loops: repeating code without repeating code

The job of a loop is to execute a certain block of code repeatedly. If we need a line of code to execute 100 times, with maybe just the value of one variable changing each time, we wouldn't want to have to write 100 lines of code that are basically all the same thing. This is what loops are for. There are two varieties

1. `For`-loops, which execute for a fixed number of iterations.
2. `While`-loops, which execute until some condition is met.

3.1 `For`-loops: repeating a fixed number of times

The job of a for-loop is to repeat the execution of a certain block of code for a set number of times. Here's general form (tip: in Rstudio, type `for` and then hit `tab`, and it will autocomplete this whole syntax):

```
for (variable in vector) {
  # Everything inside these braces gets run on every iteration of the loop.
  # However long 'vector' is, that's how many iterations the loop has.
  # Upon each iteration, the value of 'variable' becomes the next element of 'vector'.
  # Note that 'variable' and 'vector' are just placeholder names; they don't have to
  # literally be called that.
}
```

Here are some concrete examples.

```
# Before the execution of the loop, the variable "word" doesn't exist.
# Confirm this by checking the Environment pane, or with ls().
for (word in c("Hello","World")) {
  # Everything inside this loop will run twice, because the loop vector has length 2.
  # The first time, the variable "word" takes the value "Hello".
  # The second time, it takes the value "World".
  print(word)
}
```

```
[1] "Hello"
[1] "World"
```

```
some_vector = seq(1,10,by=3)
for (i in 1:length(some_vector)) {
  print(paste("The value of i is",i)) # look up what paste() does!
  print(paste("The i'th element of the vector is",some_vector[i]))
}
```

```
[1] "The value of i is 1"
[1] "The i'th element of the vector is 1"
[1] "The value of i is 2"
[1] "The i'th element of the vector is 4"
[1] "The value of i is 3"
[1] "The i'th element of the vector is 7"
[1] "The value of i is 4"
[1] "The i'th element of the vector is 10"
```

Exercise: Defining the loop vector as `1:length(some_vector)` was an example of doing things "programatically", whereas we could have simply "hard-coded" the value 4 in the vector's definition. Even though `1:length(some_vector)` and `1:4` give exactly the same vector, why is the former better (i.e. why do you want to avoid hard-coding in general)?

3.1.1 A real-life example: the Fibonacci sequence

The second example is how for-loops are frequently used in practice. We often want to loop over all elements of a vector, and each time do something specific to that element. Here's a more real-life example that does just that, and calculates the first 10 values of the Fibonacci sequence (a sequence whose first two elements are 1, and each successive element is the sum of the previous two).

```
# Initialize a vector to be all NA's (NA means "not available")
# NA is a special value in R that is used to represent missing data.
fib_seq = rep(NA,10)
fib_seq[c(1,2)] = 1 # assign the value 1 to elements 1 and 2
for (i in 3:length(fib_seq)) { # starting at 3, since we already set elements 1 and 2
  fib_seq[i] = fib_seq[i-2] + fib_seq[i-1]
 }
fib_seq
```

```
 [1]  1  1  2  3  5  8 13 21 34 55
```

```
 # Note: One way to understand (and debug) for loops is to not run the whole loop, but rather manually
  # Set i=3 and run just "fib_seq[i-2]" and "fib_seq[i-1]"
```

Exercise: Write a `for` loop that multiplies each element of a vector by its index. In other words, the vector can be any numeric vector, and element 1 gets multiplied by 1, element 2 gets multiplied by 2, etc. Now, accomplish this same thing with a vector operation, instead of a loop.

3.1.2 Don't loop if you don't have to!

We needed a `for` loop to build the Fibonacci sequence. There's no way we could've done it with just vector operations. In the above exercise, however, a `for` loop was a relatively straightforward solution, though not the best one. Any time you can accomplish a task through vector operations instead of a loop, choose vector operations, because they are much, much more computationally efficient. For really big data, the difference can literally mean seconds vs. hours, or minutes vs. days.

3.2 `While`-loops: Keep going until ...

A `for`-loop only runs for exactly as many times as the length of the loop vector. A `while`-loop, on the other hand, keeps running until a certain condition changes from `TRUE` to `FALSE`.

```
x = 6
while(x < 10) { # Keep looping "while" x is still less than 10
  print(x)
  x = x+1
}
```

```
[1] 6
[1] 7
[1] 8
[1] 9
```

Exercise: Modify the above code to include a variable that counts how many times the `while`-loop ran before terminating.

4 Other data structures

So far you've learned how to do a lot of different things with vectors, like logical operations and defining `for`-loops. These will get you a long way, but vectors have two big limitations:

1) All elements have to be of the same type. See what happens if you try to define the following vectors of mixed type.

```
c(10,TRUE)
```

```
[1] 10  1
```

```
c(10,TRUE,'hello')
```

```
[1] "10"    "TRUE"  "hello"
```

Turning a value of one type into another type is called "coercion", and R coerced certain elements to make the types uniform. In the first case, because `TRUE` and `FALSE` can be equivalently represented by 1 and 0 (see what you get from the statement `0==FALSE`), R saw that it could make a numeric vector by turning `TRUE` into `1`. In the second case, it could've done the same thing with the first two elements, but then it has no way of turning "hello" into a number, so it coerced everything to characters.

2) Vectors are one-dimensional. They can be arbitrarily long, but many of the real-world datasets we work with have two dimensions, like x and y coordinates, or location-time combinations. These are much better represented by a matrix-like object.

4.1 Matrices

As in math, a matrix in R is a rectangular grid of numbers. It's a like a vector that got stacked either horizontally or vertically, and in fact that's the standard way of creating a matrix:

```
matrix(1:6, nrow=2, ncol=3) # fill up a 2x3 matrix with the values from the vector 1:6
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
matrix(1:6, nrow=3, ncol=2) # change the number of rows, columns
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
matrix(1:6, nrow=2, ncol=4) # recycling 1:6 gets cut short; throws a warning
```

```
Warning in matrix(1:6, nrow = 2, ncol = 4): data length [6] is not a
sub-multiple or multiple of the number of columns [4]
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    1
[2,]    2    4    6    2
```

```
matrix(1:6, nrow=2, ncol=6) # recycling 1:6 succeeds, but be sure this is what you had in mind!
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    1    3    5
[2,]    2    4    6    2    4    6
```

Pay special notice to the fact in every case, the vector's values first filled up column 1, then column 2, etc. R is what's called a "column major" language, meaning it will both pack and unpack matrices by column, rather than row. If you want to fill up a matrix row-wise instead, include the `byrow=TRUE` argument.

```
mat1 = matrix(1:6, nrow=2) # why is an ncol argument not necessary?
mat2 = matrix(1:6, nrow=2, byrow=TRUE)
mat1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
mat2
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
c(mat1) # string out mat1 into a vector
[1] 1 2 3 4 5 6
c(mat2) # string out mat2 into a vector
[1] 1 4 2 5 3 6
# Note that c() will string out a matrix by columns, regardless how the matrix was created.
```

Filling a matrix by rows is **not** the same as transposing a matrix (which is done with the function `t()`). Transposition flips the rows and columns, such that the new number of columns is the old number of rows, and vice versa. Including `byrow=TRUE` to fill the matrix row-wise instead of column-wise does not change the shape, only the order of the values.

```
mat = matrix(1:6, nrow=2)
c(mat) # converting the matrix to a vector strings the values out column-wise
```

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

```
mat_transposed = t(mat) # the t() function transposes a matrix
c(mat_transposed) # explain why this is different than c(mat)
```

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

Indexing matrices works exactly like vectors, except now we need two indices (row and column) to specify an element. The syntax `myMatrix[i,j]` means "the element at row i, column j of myMatrix". As with vectors, we can supply vectors of indices to select multiple elements. Lastly, by leaving either the row or column index blank, we're saying we want *all* rows or *all* columns.

```
> mat = matrix(1:6, nrow=2) # why is ncol not needed?
> mat
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> mat[2,3] # element at row 2, column 3
[1] 6
> mat[2,c(1,3)] # elements at row2, columns 1 and 3
[1] 2 6
> mat[,c(2,3)] # all rows, columns 2 and 3
     [,1] [,2]
[1,]    3    5
[2,]    4    6
> mat[1,] # first row, all columns
[1] 1 3 5
> mat[,-1] # all rows, all columns except the first
     [,1] [,2]
[1,]    3    5
[2,]    4    6
```

Exercise: For each of the sets of indices above, reassign those elements with a legal vector or matrix.

Exercise: Which of the above statements return a vector and which return a matrix? For those that returned a vector let's say that instead want a 1-row or 1-column matrix. How could you create each? Pick one vector and convert it to a 1-row vector with `matrix()`. Then do it with the function `as.matrix()`, which doesn't create a matrix from scratch, but rather tries to coerce whatever you give it into one.

4.2 Dataframes (written as `data.frame`)

Dataframes are the signature data structure of R. Some of the more advanced tools you'll see later require your data to be in a data.frame. A data.frame is like a matrix in that it's two-dimensional (i.e. has rows and columns), but it's more flexible than a matrix because its columns may be different types (matrices can only hold numerics).

We can create a date.frame with the `data.frame()` function, where each argument is a vector that's to become a column. Usually the columns each represent separate variables, and each row represents a particular data unit (usually called an "observation"). Let's take the example of cities and populations from the earlier Vectors session. In that session, we had to keep these three pieces of information as separate vectors, because they were different data types. Data.frames let us combine them.

```
cities = c("San Francisco","London","Marrakesh")
population = c(8.37e5,8.31e6,9.29e5)
```

```
have_visited = c(TRUE,TRUE,FALSE)
df = data.frame(cities, population, have_visited)
df
```

```
         cities population have_visited
1 San Francisco     837000         TRUE
2        London    8310000         TRUE
3     Marrakesh     929000        FALSE
```

Notice how it automatically gave the columns the names of the original vectors. This is useful because you'll very frequently refer to data.frame columns by name with the `$` operator.

```
df$cities # the 'cities' variable (i.e. column) of df
```

```
[1] "San Francisco" "London"        "Marrakesh"
```

```
df$cities[df$have_visited==TRUE] # cities that have been visited
```

```
[1] "San Francisco" "London"
```

Exercise: Evaluate each piece of the above statement, both by printing its output and using `str()`, to make sure you understand what happened. Why is `df$cities` a factor, when the original variable was a character? How can you change that column back to a character?

4.2.1 (Re)naming columns

You can set the column names either within the definition itself, or with the `names()` function. This is often useful to give the columns shorter, more type-able names for your own convenience. Very often you'll read in data from a csv file with hideous column names like "NH4-MEASUREMENT-SITE-123875-07/09/2014", and nobody wants carpal tunnel syndrome.

```
# Enforcing particular names during the data.frame's creation
df = data.frame(city=cities, pop=population, visit=have_visited)
df
```

```
           city     pop visit
1 San Francisco  837000  TRUE
2        London 8310000  TRUE
3     Marrakesh  929000 FALSE
```

```
# Alternatively, if the data.frame already exists
names(df) # print the (column) names of the data.frame
```

```
[1] "city"  "pop"   "visit"
```

```
names(df) = c('city','pop','visit') # assign these values to the names of df
df
```

```
           city     pop visit
1 San Francisco  837000  TRUE
2        London 8310000  TRUE
3     Marrakesh  929000 FALSE
```

4.2.2 Indexing a data.frame

4.2.2.1 Like a matrix While the `$` operator is a very common tool for pulling out an individual column, it won't retrieve multiple columns. For this, you can refer to rows and columns in a data frame exactly as you do in a matrix. Importantly, column indices can be column names, not just column integers.

```
df[c(1,2), c(1,3)] # first two rows, first and third column
```

```
           city visit
1 San Francisco  TRUE
2        London  TRUE
```

```
df[c(1,2), c('city','visit')] # exactly the same
```

```
           city visit
1 San Francisco  TRUE
2        London  TRUE
```

4.2.2.2 Using `subset()` The `subset()` function is a great tool for getting the subset of a data.frame according to particular conditions. Very often, as in the above example, you're subsetting by some condition on a particular column (like "where is `have_visited==TRUE`"), and you only want a particular column or columns where that condition is met. You should be comfortable doing this the "manual" way, as we did above, but there

```
df[df$visit==TRUE,c('city','pop')] # keep "city" and "pop" columns, but only the rows where visit==TRUE
```

```
           city     pop
1 San Francisco  837000
2        London 8310000
```

```
subset(df, visit==TRUE, c(city,pop)) # convenient shorthand for exactly the same thing
```

```
           city     pop
1 San Francisco  837000
2        London 8310000
```

```
# Note how we don't need to (and in fact aren't supposed to) quote column names when using subset()
```

Exercise: How can you modify these statements to test for cities that have been visited and that have a population greater than 1 million?

4.2.3 Adding rows and/or columns with `rbind()` and `cbind()`

With both matrices and data.frames, sometimes the obejct already exists, but you need to add rows or columns. The functions `rbind()` (row bind) and `cbind` (column bind) do just that. They'll bind any number of rows or columns together, meaning you can bind matrices with other matrices, data.frames with other data.frames, or even a matrix with a data.frame (though the result will be coerced to a data.frame).

Exercise: Use `rbind()` to add a new observations (a city, its population, and whether it's been visited). Use `cbind()` to add a new variable, like the native language.

4.3 Lists

Lists are the most general data structure of all. List elements can be absolutely anything. One list element could be a matrix, the next element a character vector, the next element a data.frame, and so on. You can even have lists within lists. We create them with the `list()` function, name their elements exactly as we did with data.frames, and index their elements with either the `$` operator, or with double backets, `[[]]`.

```
my_list = list(a_matrix=matrix(1:4,nrow=2),
               char_vec = c("Hello","World"),
               df = data.frame(cbind(matrix(1:4,nrow=2), c("Hello","World"))))
my_list
```

```
$a_matrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$char_vec
[1] "Hello" "World"

$df
  X1 X2    X3
1  1  3 Hello
2  2  4 World
```

```
my_list$char_vec
```

```
[1] "Hello" "World"
```

```
my_list[['char_vec']]
```

```
[1] "Hello" "World"
```

```
my_list$df$X3 # Think of this as (my_list$df)$X3
```

```
[1] "Hello" "World"
```

Exercise: Go through each component of the definition of my_list. In particular, understand what happened in the creation of the list element "df"), and why you can cascade `$` operators in the statement `my_list$df$X3`.

4.3.1 Selecting multiple list elements

Somewhat confusingly, double brackets, `[[]]` are only used for selecting one and only one list element. To select multiple elements, use single brackets, as with vectors.

```
my_list[[1]] # gives list element 1
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
my_list[[1:2]] # so you'd think this would give list elements 1 and 2, but it doesn't
```

```
[1] 2
```

```
my_list[1:2] # this is what you want
```

```
$a_matrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$char_vec
[1] "Hello" "World"
```

4.3.2 Data.frames are in fact lists

While they may look and act like a special matrix, data.frames are implemented under the hood as lists. This explains why the `$` operator works to select an individual column. Each column is in fact a list element. This means you could also select a particular column with the `[[]]` notation.

```
df[['city']] # the "city" element, i.e. the "city" column, of df
```

```
[1] "San Francisco" "London"        "Marrakesh"
```