Week 13: Iteration

Ellen Bledsoe

Iteration

In this week's lesson, we will be talking about how to do repetitive tasks.

As it turns out, computers are really good at repeating tasks. We call this repetition of the same task "iteration."

Overview of Iteration in R

While our lesson today will focus mainly on one way to iterate in R (`for` loops), there are actually many ways to do so. I've included a quick overview below.

1. Writing Vectorized Functions

We can write functions that take either a single value or a vector of values, do element-wise calculations, and return a vector of results.

Many functions automatically work with either a single value or a vector of values. Each function we wrote in our functions lesson in Week 10 can work with vectors.

NOTE! An important exception it when we use `if-else` statements inside our functions; these do not work with vectors.

If you are interested in learning more about vectorized functions, here is a good place to start.

```
F_to_C <- function(F) {
  C <- (F - 32) * (5/9)
  return(round(C, 1))
}

temps <- c(47.2, 72, 100)
F_to_C(temps)
```

```
[1]  8.4 22.2 37.8
```

2. Using `apply` or `map` Functions

Both of these are families of functions. They take a function and apply that function to each item in a list of items.

The `apply` functions (e.g., `apply`, `lapply`, `sapply`) are base R; you can learn more about how to use them here.

The `map` functions come from the `purrr` package, part of the `tidyverse`. Chapter 26: Iteration in the "R for Data Science (2e)" book is a good starting point, as is the `purrr` cheatsheet.

```
# base R
sapply(temps, F_to_C)
```

```
[1]  8.4 22.2 37.8
```

```
# purrr from tidyverse
purrr::map(temps, F_to_C)
```

```
[[1]]
[1] 8.4

[[2]]
[1] 22.2

[[3]]
[1] 37.8
```

3. In data frames, using `dplyr` (mostly)

We've actually already been iterating quite a bit–specifically in data frames–by using functions from `dplyr`; we can also combine our own functions with `dplyr`.

For example, when we use a `filter` function, we are asking R to check the value in a column in the first row, then determine whether that condition is met (and, therefore, whether to retain the row), then to do the same thing for the next row and the next…

```r
temps_df <- as.data.frame(temps)

# most functions are vectorized
temps_df |>
  dplyr::mutate(tempsC = F_to_C(temps))
```

```
  temps tempsC
1  47.2    8.4
2  72.0   22.2
3 100.0   37.8
```

Again, an important exception it when we use `if-else` statements inside our functions; these do not work with vectors.

To do so, we need to use the `rowwise()` function.

```r
hot_or_cold <- function(temp){
  if (temp > 72) {
    category <- "hot"
  } else {
    category <- "cold"
  }
}

# doesn't work; functions with if-else statements are NOT vectorized
# temps_df |>
#   dplyr::mutate(category = hot_or_cold(temps))

# this works!
temps_df |>
  dplyr::rowwise() |>
  dplyr::mutate(category = hot_or_cold(temps))
```

```
# A tibble: 3 x 2
# Rowwise:
  temps category
  <dbl> <chr>
1  47.2 cold
2  72   cold
3 100   hot
```

4. Loops

Loops provide us with complete control to perform of any kind of repetition we want (rather than just one function at a time, for example).

There are two main kinds of loops:

- `for` loops when you know how many times you want to repeat something
- `while` loops when you want the loop to stop for a reason other than number of times to repeat.

```
for (i in 1:length(temps)) {
  print(F_to_C(temps[i]))       # vectorized functions work
  print(hot_or_cold(temps[i]))  # non-vectorized functions also work
}
```

```
[1] 8.4
[1] "cold"
[1] 22.2
[1] "cold"
[1] 37.8
[1] "hot"
```

This week's lesson is going to focus primarily on using loops, specifically `for` loops.

The `for` Loop

Loops are the fundamental structure for repetition in programming.

In particular, `for` loops perform the same action for each item in a list of things.

The general structure of a `for` loop is below:

```
# for (item in list_of_items) {
#   do_something(item)
# }
```

To see an example of this, let's calculate biomasses from volumes using a loop. If we want to see the results of the loop, we need to print them out. In this loop, we are going to use a `print()` function to display our results for each mass.

```r
# vector of volumes
volumes <- c(1.6, 3, 8)

# for each volume value in a vector of volumes
for (vol in volumes){           # perform some task inside the {}
  biomass <- 2.65 * vol ^ 0.9   # calculate a mass
  print(biomass)                # print out the mass
}
```

```
[1] 4.045329
[1] 7.12287
[1] 17.21975
```

The code that is inside of the curly brackets will run once for each value in `volumes`. The `vol` object is a placeholder, representing one value from `volumes` at a time.

Everything between the curly brackets is executed each time through the loop, from top to bottom.

The loop takes the first value from `volumes` and assigns it to the object `vol`. It performs the code inside the curly brackets, calculating the biomass and then printing the result.

Once the code hits the end of of what is enclosed inside the curly brackets, it starts the code all over again from the beginning using the next item in the list (or the `volumes` vector, in this case). The code takes the second value from `volumes`, assigns it to `vol`, does the calculation and prints the output.

It continues to repeat this process–running the code with one value, then starting the whole process over again with the next value–until it has run through every element in the list.

Ultimately, the loop above is doing the exact same thing as the code below, but in a more condensed and more flexible way.

```r
# same as the for loop above
volume <- volumes[1]
biomass <- 2.65 * volume ^ 0.9
print(biomass)
```

```
[1] 4.045329
```

```r
volume <- volumes[2]
biomass <- 2.65 * volume ^ 0.9
print(biomass)
```

```
[1] 7.12287
```

```
volume <- volumes[3]
biomass <- 2.65 * volume ^ 0.9
print(biomass)
```

```
[1] 17.21975
```

Let's Practice!

Work on Questions 1a and 1b in the Assignment.

Looping with an Index

Loops are iterating over a series of elements in a vector or other list-like object. When we use that value directly, this is called looping by value.

There is another way to loop: *looping by index*

You might remember way back at the beginning of the course that we talked about two different ways to subset data: by index or by condition. Sub-setting based on index meant sub-setting *based on the position* of the item(s) we are interested in.

Looping by index does something similar. It loops through a list of integer index values, typically starting at 1. These integers are then used to access values in one or more vectors *based on the position* indicated by the index.

Let's take our previous loop and modify it to use an index.

1. We often use `i` to stand for "index" as the variable we update with each step through the loop.

   ```
   volumes <-  c(1.6, 3, 8)

   for (i ...)
   ```

2. We then create a vector of position values starting at 1 (for the first value) and ending with the length of the object we are looping over.

   ```
   volumes <- c(1.6, 3, 8)

   for (i in 1:3)
   ```

3. What if the length of the object we are looping over is variable or might change in the future? We don't want to be stuck with a certain length.

   Fortunately, we don't have to know the length of the vector. We can add flexibility by using a function to count the number of elements that we will want to loop over.

   - For a vector, the `length()` function is useful for this.
   - For a data frame, the `nrow()` function is often used.

   Because we are currently working with a vector, let's use the `length()` function.

```
volumes <-  c(1.6, 3, 8)

for (i in 1:length(volumes)){

}
```

4. Then, inside the loop, instead of doing the calculation on the index value (which a number between 1 and 3 in our case), we use square brackets and the index as a placeholder. Now, the loop will run through the code, replacing the `i` placeholder with the appropriate index. That will reference the correct value from our `volumes` vector.

```
volumes <- c(1.6, 3, 8)

for (i in 1:length(volumes)){
   biomass <- 2.65 * volumes[i] ^ 0.9
   print(biomass)
}
```

```
[1] 4.045329
[1] 7.12287
[1] 17.21975
```

This new version of the loop gives us the same result as our original loop, but it's more complicated to understand. So why would we want to loop by index?

The advantage to looping by index is that it lets us do more complicated things, as we will see below.

Storing Results

One of the most common reasons we use `for` loops with indices is to store the outputs from the loop. We typically want the results to be in the same order (position) as what we used to calculate them.

To store results, we start by creating an empty object (vector, column, data frame, etc.) before the loop starts. This object should be the same length as the results will be, which is typically the same as the object we are looping over.

To store results in a vector, we can use the function `vector` to create an empty vector of the correct length. For the `vector` function:

- `mode` is the type of data we are going to store
- `length` is the number of items in the vector

```
biomasses <- vector(mode = "numeric", length = length(volumes))
biomasses
```

```
[1] 0 0 0
```

Now that we have a place to store the results, we need to tell the loop to place the results in the correct position in the empty vector. We can use the index to do this.

For each trip through the loop, we want to put the output into the empty vector at the `i`th position.

```
for (i in 1:length(volumes)){
   biomass <- 2.65 * volumes[i] ^ 0.9
   biomasses[i] <- biomass
}

biomasses
```

```
[1]  4.045329  7.122870 17.219751
```

Let's Practice!

Work on Questions 1c and 1d in the Assignment.

Looping over Multiple Values

So far, we've talked about looping through one group of values (in a vector). Looping with an index also allows us to access values from multiple vectors.

```r
# vectors
a_values <- c(2.65, 1.28, 3.29)
b_values <- c(0.9, 1.1, 1.2)
volumes <- c(1.6, 3, 8)

# empty vector for results
biomasses <- vector(mode = "numeric", length = length(volumes))

# for loop
for (i in 1:length(volumes)){
   biomass <- a_values[i] * volumes[i] ^ b_values[i]
   biomasses[i] <- biomass
}
```

Work on Question 1e in the Assignment.

Looping with Functions

Another common thing to do is combine loops with functions by calling one or more functions as a step in our loop.

For example, let's take the non-vectorized version of our `est_biomass` function that returns an estimated biomass if the `volume > 5` and `NA` if it's not.

```r
est_biomass <- function(volume, a, b){
  if (volume < 5) {
    biomass <- a * volume ^ b
  } else {
    biomass <- NA
  }
  return(biomass)
}
```

We can't pass the vector to the function and get back a vector of results because `if` statements are not vectorized.

Instead, we can loop over the values.

First, we'll create an empty vector to store the results. Then we will loop by index, calling the function for each value of `volumes`.

```
# empty vector to store results
biomasses <- vector(mode = "numeric", length = length(volumes))

# for loop using the `est_mass_max` function
for (i in 1:length(volumes)){
   biomass <- est_biomass(volumes[i], a_values[i], b_values[i])
   biomasses[i] <- biomass
}
```

If we hadn't written the `if-else` statement as a function (`est_biomass`), we could still include it! It would look like this:

```
biomasses <- vector(mode = "numeric", length = length(volumes))

for (i in 1:length(volumes)){
  if (volumes[i] < 5) {
    biomass <- a_values[i] * volumes[i] ^ b_values[i]
  } else {
    biomass <- NA
  }
  biomasses[i] <- biomass
}
```

Realistically, I find myself putting `if-else` statements inside `for` loops, like the structure above, rather frequently.

Troubleshooting Loops

Occasionally (okay, maybe often...) when you write a for loop, it will either give you and error or it will successfully run but the results won't be what you anticipated.

Since loops (like functions) have some kind of placeholder in them (such as `i` for index), what is the best way to figure out what the error is?

I often set my placeholder equal to a specific value and run through each line of code between the `{}` to check every step individually.

```
biomasses <- vector(mode = "numeric", length = length(volumes))
biomasses
```

```
[1] 0 0 0
```

```
for (i in 1:length(volumes)){
  if (volumes[i] < 5) {
    biomasses[i] <- a_values[i] * volumes[i] ^ b_values[i]
  } else {
    biomasses <- NA
  }
}

biomasses
```

```
[1] NA
```

```
# in the console, set i = to a number; run the entire code chunk line by line
# i = 2
# i = 3
```

If part of your loop seems to be behaving the way you expect but another part isn't, I often find a value (or row) that should cause the loop to perform the task that isn't working the way you expect and set the index to that value.

Looping over Data Frames

By default, when R loops over a data frame, it loops over the columns. Remember, R is oriented to work in vectors, and data frames are organized by converting multiple vectors into the columns.

```
data <- data.frame(a = a_values, b = b_values, volume = volumes)

for (i in data) {
  print(i)
}
```

```
[1] 2.65 1.28 3.29
[1] 0.9 1.1 1.2
[1] 1.6 3.0 8.0
```

If you want to loop through columns and apply the same code to each (perhaps calculating a summary statistic), this is one way to do that.

To loop over rows, we need to do some extra work. We can loop by index and subset each row.

```
for (i in 1:nrow(data)) {
  print(data[i, ])
}
```

```
     a   b volume
1 2.65 0.9    1.6
     a   b volume
2 1.28 1.1      3
     a   b volume
3 3.29 1.2      8
```

If we want to use a specific column, we need to specify the column via sub-setting, as well.

```
biomasses <- vector(mode = "numeric", length = length(volumes))

for (i in 1:nrow(data)) {
  biomass <- est_biomass(data$volume[i], data$a[i], data$b[i])
  biomasses[i] <- biomass
}
```

An alternative structure (without the created function) might look like this.

We can create a new, empty column in the data frame. We can also place the `if-else` statement directly into the for loop and reference specific columns with `$` operator and subset by index, as we did above.

```
data$biomass <- NA

for (i in 1:nrow(data)){
  if (data$volume[i] < 5) {
    biomass <- data$a[i] * data$volume[i] ^ data$b[i]
  } else {
    biomass <- NA
  }
  data$biomass[i] <- biomass
}
```

Let's Practice!

Work on Question 2 in the Assignment.

Looping over Files

While we often want to repeat things within our data, either in vectors or data frames, there are other instances when we might want to repeat the same task.

For example, if we have many similar files, we might want to repeat the same tasks for all of those files.

To demonstrate, let's download some simulated satellite collar data. Make sure you change the file paths accordingly to match the sub-directories in your RStudio Project, as needed.

```
download.file(url = "http://www.datacarpentry.org/semester-biology/data/locations.zip",
              destfile = "../data_raw/locations.zip") # where to put the zip file

# unzip() function to open the zip file
# zipfile argument is the location of the file to be unzipped
# exdir argument is the location where the "extracted" files should be placed
unzip(zipfile = "../data_raw/locations.zip", exdir = "../data_raw/")
```

Before we can loop through these files, we need to get the names of all of these files.

We can do so by using the `list.files()` function.

If we run it without arguments, it will give us the names of all files in the directory.

```
list.files()
```

```
 [1] "Week13_Assignment_AnswerKey.pdf"
 [2] "Week13_Assignment_AnswerKey.Rmd"
 [3] "Week13_Assignment_key.pdf"
 [4] "Week13_Assignment_key.Rmd"
 [5] "Week13_Assignment.qmd"
 [6] "Week13_Assignment.Rmd"
 [7] "Week13_Iteration_Instructor.pdf"
 [8] "Week13_Iteration_Instructor.qmd"
 [9] "Week13_Iteration_Instructor.rmarkdown"
[10] "Week13_Iteration.qmd"
```

```
list.files(path = "../data_raw/")
```

```
 [1] "collar-data-A1-2016-02-26.txt"  "collar-data-B2-2016-02-26.txt"
 [3] "collar-data-C3-2016-02-26.txt"  "collar-data-D4-2016-02-26.txt"
 [5] "collar-data-E5-2016-02-26.txt"  "collar-data-F6-2016-02-26.txt"
```

```
 [7] "collar-data-G7-2016-02-26.txt"  "collar-data-H8-2016-02-26.txt"
 [9] "collar-data-I9-2016-02-26.txt"  "collar-data-J10-2016-02-26.txt"
[11] "dinosaur_lengths.csv"           "individual_collar_data.zip"
[13] "locations-2016-01-01.txt"       "locations-2016-01-02.txt"
[15] "locations-2016-01-03.txt"       "locations-2016-01-04.txt"
[17] "locations-2016-01-05.txt"       "locations.zip"
```

We, however, just want the data files we downloaded, so we'll add the optional `pattern` argument to only get the text files (end with `.txt`).

The pattern argument can also take regular expressions if you need to be more nuanced about which files to include.

```
# in regex, .* means any character (.) any number of times (*)
# "locations.*.txt" indicates any file starting with "locations" and ending with ".txt"
data_files <- list.files(path = "../data_raw/", pattern = "locations.*.txt")
data_files
```

```
[1] "locations-2016-01-01.txt" "locations-2016-01-02.txt"
[3] "locations-2016-01-03.txt" "locations-2016-01-04.txt"
[5] "locations-2016-01-05.txt"
```

Now that we have our list of file names, we can loop over it.

Perhaps we want to count the number of observations in each file.

First, we create an empty vector to store those counts.

```
num_files <- length(data_files)
results <- vector(mode = "integer", length = num_files)
```

Now that we have created our empty vector of the appropriate length, we can write our loop to perform the task we want–counting the number of observations (rows) in each file.

```
for (i in 1:num_files){
  # specify which file
  filename <- data_files[i]
  # read in the file
  data <- read.csv(paste0("../data_raw/", filename))
  # save the number of rows as an object called `count`
  count <- nrow(data)
  # save the count value in the correct position in the `results` vector
```

```
  results[i] <- count
}

results
```

```
[1]  4  8 10 10 12
```

Let's Practice

Start with Question 3a. Note that the question uses different simulated collar data.

Storing Loop Results in a Data Frame

Often, we want to calculate multiple pieces of information in a loop. When we do this, it is particularly useful to store these results in something other than individual vectors. Instead, we might want to store them in a data frame.

We've actually done a version of this above in the "Looping Over Data Frames" section, where we added an empty column to a data frame.

We can also do so by creating an empty data frame then storing the results from the `for` loop in the `i`th row of the appropriate column.

For example, let's say we want to associate the file name with the observation count and also get the minimum latitude value for each file.

We would start by creating an empty data frame that we will then populate with the results of our loop. We can use the `data.frame` function. Each argument is for one column.

- `column_name` = an empty vector of the correct type and length

```
results <- data.frame(file_name = vector(mode = "character", length = num_files),
                      count = vector(mode = "integer", length = num_files),
                      min_lat = vector(mode = "numeric", length = num_files))
results
```

```
  file_name count min_lat
1                 0       0
2                 0       0
3                 0       0
4                 0       0
5                 0       0
```

Let's modify our previous loop.

Instead of storing `count` in `results[i]`, we will need to first specify which column (`count`) in the `results` data frame the value should go in. We can do this with the `$` operator: `results$count[i]`

We also want to store the file name and the minimum latitude in a similar fashion.

```
for (i in 1:num_files){

  # read in the data file
  filename <- data_files[i]
  data <- read.csv(paste0("../data_raw/", filename))

  # add information about the data file into the data frame
  results$file_name[i] <- filename
  results$count[i] <- nrow(data)
  results$min_lat[i] <- min(data$lat)

}
```

Let's Practice!

You should now have all the skills you need to complete the assignment!