Loops

## Learning Objectives

* Use a for loop to repeat operations
* Avoid writing slow loops
* Use apply as an alternative to for loops

Using looping structures allows us to repeat operations, e.g. do something to every row of a data frame. In this lesson we will focus specifically on for loops. Here is the basic structure of a for loop:

for (variable in vector) {  
 do something  
}

The loop is repeated for every element of the vector. Using the names above, in each iteration variable takes the value of one of the elements of vector. Here is a simple example:

for (i in 1:10) {  
 print(i)  
}

[1] 1  
[1] 2  
[1] 3  
[1] 4  
[1] 5  
[1] 6  
[1] 7  
[1] 8  
[1] 9  
[1] 10

Loops have a bad reputation in R, and a common myth is that "loops in R are slow." However, it would be more accurate to say that "poorly written loops in R are slow."

We'll start with a trivial example of a poorly written for loop to review the basics before we move on to more interesting examples. In a previous lesson we added a pseudocount of 1 to every citation count before taking the log. Here's one way we could do that using a for loop.

x <- numeric()  
for (i in 1:length(counts\_raw$wosCountThru2011)) {  
 x <- c(x, counts\_raw$wosCountThru2011[i] + 1)  
}

The function length returns the total number of elements in the vector. Each time through the loop, the variable i increases by 1. Furthermore, i is used to index the vector of citation counts. Thus each time through the loop, the next element of vector gets 1 added to it, and this new value is appended to x.

This is incredibly slow. The main culprit is because each time through the loop, the new vector x grows in size. It is more memory-efficient to pre-allocate the new vector to a given size before starting the loop. To create a numeric vector of a given size, we use the function numeric.

x <- numeric(length = length(counts\_raw$wosCountThru2011))  
for (i in 1:length(counts\_raw$wosCountThru2011)) {  
 x[i] <- counts\_raw$wosCountThru2011[i] + 1  
}

This was much faster because the new vector x did not grow in each iteration. Instead we used i to index both the new vector and the old vector.

Lastly, you can avoid slow loops by using vectorized operations whenever possible. This is what we had done in the previous lesson. Because R is vectorized by default, we do not need to use a for loop when doing something simple like adding a number to a vector. Many times the vectorized version is optimized to be faster than any loop you could write yourself.

x <- counts\_raw$wosCountThru2011 + 1

So in summary, avoid slow loops by:

* Using vectorized operations when possible
* Pre-allocating the size of the new object before the loop begins

As a more useful example, let´s calculate the mean number of citations for articles in the different PLOS journals. There are seven journals in our data set, and they are stored as a factor in the column journal.

levels(counts\_raw$journal)

[1] "pbio" "pcbi" "pgen" "pmed" "pntd" "pone" "ppat"

Thus our result will need to be pre-allocated to a size of 7.

result <- numeric(length = length(levels(counts\_raw$journal)))

In the first example, we used i as the looping variable, and it took on the values from 1 to the length of the vector. Since we want to loop through the seven journals, this time our looping variable will take on these values. In order to index result using this indexing variable, we need to name each element of result.

names(result) <- levels(counts\_raw$journal)  
result

pbio pcbi pgen pmed pntd pone ppat   
 0 0 0 0 0 0 0

result["pone"]

pone   
 0

Now we construct the for loop.

for (j in levels(counts\_raw$journal)) {  
 print(j)  
}

[1] "pbio"  
[1] "pcbi"  
[1] "pgen"  
[1] "pmed"  
[1] "pntd"  
[1] "pone"  
[1] "ppat"

Lastly, we need to calculate the mean number of citations, using a conditional statement to keep only citation counts for articles in a specific journal.

for (j in levels(counts\_raw$journal)) {  
 result[j] <- mean(counts\_raw$wosCountThru2011[counts\_raw$journal == j])  
}  
result

pbio pcbi pgen pmed pntd pone ppat   
28.705905 14.219258 22.928208 18.148110 7.348564 8.306972 20.892613

## Alternative options

Because performing analyses on each level of a factor is such a common practice, R has built-in functions to do this like tapply and aggregate. Also, packages like dplyr, which we will see in future lessons, also provide this functionality.

### Using apply statements

R provides other methods for repeating operations. One useful function is apply, which performs the same operation across all the rows or columns of a data frame.

Let's create a new summary statistic for the articles that is the average of their citation count in 2011 (wosCountThru2011), the number of tweets (backtweetsCount), and the number of PLOS comments (plosCommentCount). We subset the data frame by listing the 3 columns we want:

counts\_sub <- counts\_raw[, c("wosCountThru2011", "backtweetsCount",  
 "plosCommentCount")]  
counts\_sub[1:5, ]

wosCountThru2011 backtweetsCount plosCommentCount  
1 33 0 0  
2 181 0 0  
3 0 0 0  
4 0 0 0  
5 371 0 0

apply takes 3 arguments. The first is the name of the data frame, the second is the number 1 to specify rows or 2 to specify columns, and the third is the function to be applied. We'll name the new summary statistic sum\_stat.

sum\_stat <- apply(counts\_sub, 1, mean)  
summary(sum\_stat)

Min. 1st Qu. Median Mean 3rd Qu. Max.   
 0.0000 0.6667 2.0000 4.7060 5.0000 245.7000

Thus with just one line of code we were able to compute the mean across every row of the data frame.

### Challenges

## Using apply

Using apply and sd, calculate the standard deviation of each row of counts\_sub.  
Using apply and max, calculate the maximum of each column of counts\_sub.

sum\_stat\_sd <- apply(counts\_sub, 1, sd)  
summary(sum\_stat\_sd)

Min. 1st Qu. Median Mean 3rd Qu. Max.   
 0.000 1.155 3.464 7.832 8.563 425.500

apply(counts\_sub, 2, max)

wosCountThru2011 backtweetsCount plosCommentCount   
 737 363 49