Lesson 04 - Loops, logic, and (l)apply

Jul 5, 2016

In this lesson we will cover logic, loops, and the apply family of functions in R–the nuts and bolts of coding in R. This lesson uses this R script.

Logical comparisons

if statements

"If statements" evaluate a logical expression, and if the logical expression is `TRUE`, executes code in a block.

```
> president <- 'John Adams'
> if (president %in% characters$Person) {
+     print('in data frame')
+ }
```

```
> president <- 'Thomas Jefferson'
> if (president %in% characters$Person) {
+     print('in data frame')
```

```
+ }
[1] "in data frame"
```

if / else if / else

Adding an `else` block after an if statement only runs if the statement is evaluated as `FALSE`.

```
> year <- 1776
> if (year < 1776) {
+     america <- FALSE
+ } else {
+     america <- TRUE
+ }
> america
[1] TRUE
```

If you have multiple conditions that you want to check sequentially, you can use `else if`, with or without an `else` statement at the end.

```
> candidate <- 'John Adams'
> if (candidate == 'Thomas Jefferson') {
+     print('The election of 1800')
+ } else if (candidate == 'John Adams') {
+     print('Welcome, folks, to the Adams administration')
+ } else if (candidate == 'George Washington') {
+     print('Here comes the General')
+ } else {
+     print('Never gonna be President now')
+ }
[1] "Welcome, folks, to the Adams administration"
```

You will probably not use if statements as often in R as you would in other languages, because R has many built-in commands to handle logical questions. However, you will use logic frequently.

logical operations

The following operations all return `TRUE` or `FALSE`:

Operation	Description	Example
`==`	equals	`x == 5`
`!=`	does not equal	`x != 5`

`>`	greater than	`x > 5`
`>=`	greater than or equal to	`x >= 5`
`<`	less than	`x < 5`
`<=`	less than or equal to	`x <= 5`
`%in%`	in	`x %in% vec`
`!`	not	`! x %in% vec`
`&&`	and (within an if statement)	`x > 4 && x < 6`
`||`	or (within an if statement)	`x == 5 || x == 10`

Element-wise operations

In addition, you can perform element-wise logical operations, which return a logical vector as output. All of the above commands are the same, except for and/or, which are slightly different:

Operation	Description	Example
`&`	and (elementwise)	`vec > 5 & vec < 10`
`|`	or (elementwise)	`vec == 5 | vec == 10`

The logical vector output from element-wise operations can be used to subset data:

```
> years[! is.na(years)]
[1] 1776 1780 1781 1789 1800
```

It can also be used with the `ifelse` function, which takes three arguments: a logical expression as the first argument, which returns the second argument for all `TRUE` values and the third argument for all `FALSE` values. For example:

```
> characters$Can.Vote <- ifelse(characters$Gender == 'Male', 'Yes', 'No')
> characters
                      Person Born Died Gender    State Can.Vote
1          Alexander Hamilton 1755 1804   Male New York      Yes
2 Elizabeth Schuyler Hamilton 1757 1854 Female New York       No
3                 Aaron Burr 1756 1836   Male New York      Yes
4    Angelica Schuyler Church 1756 1814 Female New York       No
5           George Washington 1732 1799   Male Virginia      Yes
6            Thomas Jefferson 1743 1826   Male Virginia      Yes
```

(Ladies, tell your husbands, vote for Burr!)

In addition, you can perform logical tests on logical vectors. The `any` command takes a logical vector and returns `TRUE` if any element is `TRUE`, and `FALSE` otherwise (e.g. `any(is.na(years))`). The `all` command is similar, except it returns true only if **all** elements are `TRUE`

What questions can you ask about the `years` vector?

For loops

For loops iterate over a series of numbers, performing an operation for each number. For example:

```
> for (x in 3:10) {
+    print(x * (x - 1) / 2)
+ }
```

prints the result of `x * (x - 1) / 2` for each number from 3 to 10.

In general, you should *say no to this*. While for loops are a staple of other languages, **they are bad form in R**. R is built to perform operations vector-wise, not a single element at a time. In R, loops are slower than vector-based operations and require special assignment operators if you want to save variables in a for loop.

Instead, in R we use the **apply** family of functions. These functions accomplish the same thing that for loops do, but R processes them much faster.

The apply family of functions

You ready for more yet? Here is a brief and incomplete introduction to some of the most important members of the apply family of functions. If you ever want to perform some kind of repetitive action in R, there is probably a way of accomplishing the task using a member of the apply family.

lapply and sapply

`lapply` (list-apply) performs a function for each element in a vector, and returns a list. To run the above operation using lapply:

```
> lapply(3:10, function(x) { x * (x - 1) / 2 })
```

Here, list format is inconvenient for us. The function `sapply` (simplified apply) simplifies the output into a one-dimensional vector or a two-dimensional matrix, if possible.

```
> sapply(3:10, function(x) { x * (x - 1) / 2 })
[1]  3  6 10 15 21 28 36 45
```

And we get a vector output!

Functions

The first argument of `lapply` and `sapply` is a vector, and the second is a function. Here, we've been creating a function on the fly. We can also use named functions:

```
> sapply(3:10, sqrt)
[1] 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000 3.16227
```

We can also create our own functions, which is useful if you want to perform the same operation for multipe instances.

```
> calculate <- function(x) {
+     return(x * (x - 1) / 2)
+ }
>
> sapply(3:10, calculate)
[1]  3  6 10 15 21 28 36 45
```

This creates a new function `calculate`, which takes a single argument `x`, and returns a single value. This gives us the same result as before.

Can you create a function that takes two arguments, `x` and `y`, and returns a single value? (Hint: `function(x, y)`.)

apply

`apply` performs a function over the margins (dimensions) of a matrix. It takes three arguments: a matrix, the margin to operate on, and the function.

Rows are dimension 1, columns are dimension 2. You may even have a multi-dimensional matrix, in which you can apply a function in dimensions larger than 2.

Let's create a matrix and use `apply` to get the sums of each row.

```
> mat <- matrix(runif(36), 6, 6)
> mat
```

```
            [,1]      [,2]       [,3]      [,4]      [,5]       [,6]
[1,] 0.002867928 0.1100434 0.03687472 0.4496658 0.8361962 0.33399994
[2,] 0.583620251 0.2020315 0.93322561 0.6536612 0.8086006 0.84139340
[3,] 0.760687128 0.6532561 0.52838671 0.6673850 0.9317638 0.98515174
[4,] 0.511206046 0.6005721 0.03271370 0.8100123 0.8883558 0.10507092
[5,] 0.117145436 0.9188314 0.45514699 0.9381366 0.1823394 0.98072499
[6,] 0.334364618 0.9936215 0.57287663 0.6996659 0.8466637 0.05897952
> apply(mat, 1, sum)
[1] 1.769648 4.022533 4.526631 2.947931 3.592325 3.506172
```

How can we get the sums of each column using apply? How can we get the median of each column?

There are four functions to find the means and the sums of a two-dimensional matrix built-in: `rowSums`, `rowMeans`, `colSums`, `colMeans`. Do you get the same results with these functions that you do with `apply`?

mapply

`mapply` (multivariate apply) calls a function repeatedly that takes multiple arguments, calculating the output for each set of arguments in the list. Unlike the previous apply functions we've learned, `mapply` takes the function *first* and then the variables to calculate over.

For example, we can calculate the age of each of our characters using `mapply`:

```
> characters$Age <- mapply(
+     function(x, y) { y - x },
+     characters$Born,
+     characters$Died
+ )
>
> characters
                      Person Born Died Gender    State Can.Vote Age
1          Alexander Hamilton 1755 1804   Male New York      Yes  49
2 Elizabeth Schuyler Hamilton 1757 1854 Female New York       No  97
3                 Aaron Burr 1756 1836   Male New York      Yes  80
4   Angelica Schuyler Church 1756 1814 Female New York       No  58
5           George Washington 1732 1799   Male Virginia      Yes  67
6            Thomas Jefferson 1743 1826   Male Virginia      Yes  83
```

(This is a simple example. An easier way to do this specific task would be `characters$Age <- characters$Died - characters$Born`.)

Homework

Save the following calculations in a script:

1. Create a 10 x 10 matrix with numbers fitting a normal distribution (`matrix(rnorm(100, mean=X, sd=Y), 10, 10)` , where X and Y are numbers of your choice). Find the means and standard deviations of each row and column. Do these correspond to the values you chose for the `rnorm` inputs?
2. How many values in your matrix are more than one standard deviation above the mean?
3. Which values in your matrix are between one and two standard deviations below the mean?
4. Plot a histogram (`?hist`) of petal lengths in the `iris` data set, excluding flowers of the setosa species.

Resources

- Statistical Analysis: an Introduction using R/R/Logical operations
- A brief introduction to “apply” in R
