Lists and For Loops

Marguerite A. Butler

February 12, 2018

1 Lists

Lists in R are vectors like any other vector, but more flexible in that elements of a list can have different data types. This has at least three consequences. First any operation that you can perform on a vector can also be done on a list. Second, any types of objects can be organized together into a list, which are very convenient for things like model fits, where you may want to store the model formula, the data, the coefficients, any likelihood values, and any other relevant information together into one data object. Third, you can use lists as containers for containers, which can be nested indefinitely.

The elements of lists can be named, either upon creation, or using the `names()` function:

```
> applicant <- list(fullname="Mickey Mouse", address="123 Main St.",  state="CA")
> applicant

$fullname
[1] "Mickey Mouse"

$address
[1] "123 Main St."

$state
[1] "CA"

> names(applicant) <- c("fullname", "address", "state")
> applicant

$fullname
[1] "Mickey Mouse"

$address
[1] "123 Main St."

$state
[1] "CA"
```

We can also use all of the standard functions that work on vectors, such as the combine function:

```
> applicant <- c(applicant, list(scores=matrix(1:10, nrow=2)))
> applicant

$fullname
[1] "Mickey Mouse"
```

```
$address
[1] "123 Main St."

$state
[1] "CA"

$scores
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

If we had multiple applicants, we could put them all together in a list of lists.

1.1 Accessing list elements

List elements can be accessed with the usual operators for vectors:

$ If the list is named

[ ] By number or name of the list element with single brackets. Returns a list. Can use a vector of indices or names.

[[ ]] By number or name with double brackets. Returns the element inside the list slot. Must be a single index or name.

```
> applicant$fullname

[1] "Mickey Mouse"

> applicant[1]   ## returns a list of length one

$fullname
[1] "Mickey Mouse"

> applicant[[1]]  ## returns the object within applicant[1]

[1] "Mickey Mouse"
```

Single brackets return lists, and can select multiple elements:

```
> applicant[1:2]

$fullname
[1] "Mickey Mouse"

$address
[1] "123 Main St."

> applicant[c("fullname", "address")]

$fullname
[1] "Mickey Mouse"

$address
[1] "123 Main St."
```

Double brackets return the element within the list slot, but only one:

```
> applicant[[1]]

[1] "Mickey Mouse"

> applicant[["fullname"]]

[1] "Mickey Mouse"

> applicant[[1:2]]  ## cannot subset [[]] with more than one index

Error in applicant[[1:2]] : subscript out of bounds
```

Exclusion index:

```
> applicant

$fullname
[1] "Mickey Mouse"

$address
[1] "123 Main St."

$state
[1] "CA"

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

> applicant[-3]

$fullname
[1] "Mickey Mouse"

$address
[1] "123 Main St."

$scores
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Accessing elements within a matrix in a list:

```
> applicant[4]

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

> applicant[[4]][2,1]  # Take the scores matrix, and grab row 2, column 1.

[1] 2

> applicant[[4]][,3]  # Take the scores matrix, and grab all of column 3.

[1] 5 6
```

1.2 For loops

Because of the flexibility of lists, they are useful containers for the output of loops or other repeated operations on data. What is a loop you may ask? It is a set of code that you want to execute repeatedly. For example, you may have a large number of datasets that you want to perform the same set of operations on.

The easiest type of loop to understand is the `for` loop. It is a counted loop, or repeated a fixed number of times. You may be familiar with for loops (or for-next loops) from other computing languages. In R the for loop operates over a vector, once for each element of the vector. The syntax is:

```
> for (var in seq) expr
```

Where `var` is a variable which takes on values of the vector `seq` and evaluates a block of code `expr`. The loop is evaluated once for each value of `seq`. If we need `expr` to span more than one line, we can do this by enclosing the loop with (even if it's only one line it's often nice for readability).

```
> for (i in 1:3) {
+    print(paste("This is a for loop", i))
+ }

[1] "This is a for loop 1"
[1] "This is a for loop 2"
[1] "This is a for loop 3"
```

It is traditional to use `i, j, or k` as the variable to remember that it's a counting index, but it is often convenient to use names that are meaningful to understand the code. For example, in the context of our earlier example, it might be helpful to iterate over each applicant in our applicant list:

```
> for (applicant in applicant_list) expr
```

1.3 Saving loop output to lists

Often we want to save the result or output of the code to a list. But we don't want to create a list with each iteration of the loop, we just want to fill the list element or add on to the list. So in order to do this, we need to create the list **outside** of the loop and then modify it **inside** the loop.

One strategy is to fill the list element by element using the counter `i` (note that we don't have to tell R how long the list is when we create it. We can just make an empty list, R will just keep adding to `mylist`):

```
> mylist <- vector("list")   ## creates a null (empty) list
> mylist

list()

> for (i in 1:4) {
+    mylist[i] <- list(data.frame(x=rnorm(3), y=rnorm(3)))  ## why does this have to be a list object?
+ }
> mylist

[[1]]
         x           y
1 1.303738  1.31873939
2 1.033293 -0.05714124
3 1.785391  1.30036364

[[2]]
          x          y
```

```
1  1.165251 -0.4425882
2 -1.067809  0.7938287
3 -1.200954 -0.2745065

[[3]]
           x          y
1  0.1558222  0.9267148
2 -0.5606817  0.9663433
3 -0.8270751 -0.6898655

[[4]]
           x          y
1 -1.6968752 -0.4598226
2 -0.7246867  0.5404865
3  0.2417545 -0.4297937
```

This code does the same thing, but uses the `c()` function to add on to `mylist` (what happens when you add on to a null list?):

```
> mylist <- vector("list")   ## creates a null (empty) list
> for (i in 1:4) {
+    mylist <- c(mylist, list(data.frame(x=rnorm(3), y=rnorm(3))))
+ }
> mylist

[[1]]
            x          y
1 -0.36247825  0.2267081
2  0.01171714  1.0033355
3  0.93495098 -0.5352878

[[2]]
           x            y
1  0.8780863  1.104096614
2 -0.9783413 -0.002600151
3 -0.4878411  2.593102907

[[3]]
           x          y
1  0.4784230  1.0967691
2 -0.3593108 -0.3477982
3  1.8336661  2.0872330

[[4]]
           x          y
1 -1.2736768 -0.2192241
2 -1.1987001  0.1155830
3  0.6399114  0.9461062
```

1.4 Reshaping lists

You often want to reshape list output in scientific programming. For example, you may fit models many times on many permutations of your data, for example, and you want to flatten your list and make a dataframe.

When you know that your output is regular, it is often convenient to use the `unlist()` function. Unlist will also work on dataframes, because you know, dataframes are lists of vectors all of the same length.

```
> lm.out <- lm( mylist[[1]]$x  ~ mylist[[1]]$y )  ## calculate a linear regression on dataframe 1 x as a
> aov.out <- anova(lm.out)   ## run anova, save to aov.out
> aov.out

Analysis of Variance Table

Response: mylist[[1]]$x
             Df  Sum Sq Mean Sq F value Pr(>F)
mylist[[1]]$y  1 0.42129 0.42129  0.8952 0.5176
Residuals      1 0.47061 0.47061

> unlist(aov.out)

      Df1       Df2   Sum Sq1   Sum Sq2  Mean Sq1
1.0000000 1.0000000 0.4212897 0.4706121 0.4212897
 Mean Sq2  F value1  F value2   Pr(>F)1   Pr(>F)2
0.4706121 0.8951952        NA 0.5176116        NA
```

1.5 Exercises

1. Take mylist above and name its elements (the dataframes).
2. Write another `for` loop to return the maximum value of x and y in each dataframe. How can you make the code flexible to make it work if mylist has a different length?
3. Write a `for` loop to loop over `mylist`. Within this loop, for each dataset compute an anova on `x  y`, `unlist` the anova output, and add as a row to a final dataframe.