Week 3: Lists, loops, and other stuff R

Dan Rabosky

November 9, 2011

This week (and next) we will continue learning basic datatype and will finish by learning a *serious* programming concept: the `for` loop.

1 The list

R Last week, we studied vectors, matrices, and dataframes. We'll start this week with a more complex data structure: the list. The list is a special R datatype that can store lots of different types of information. Lists can combine vectors, matrices, and much more into a single data object. For example, we can make a list using the `list` function as follows:

```
> myList <- list(animals = c("gator", "anhinga", "armadillo", "moorhen"),
+     rocks = c("limestone", "agatized coral"), plants = "spanish moss")
```

Now we look at `myList`:

```
> myList

$animals
[1] "gator"     "anhinga"   "armadillo" "moorhen"

$rocks
[1] "limestone"      "agatized coral"

$plants
[1] "spanish moss"

> length(myList)

[1] 3
```

`myList` is a data object that is now storing 3 character vectors (hence, `length(myList) == 3`. You see that each *component* of the list has a name: animals, rocks, and plants. To access a component of the list, you use the `$` operator like this:

```
> myList$animals

[1] "gator"     "anhinga"   "armadillo" "moorhen"
```

which is a character vector of length 4. And you can treat this character vector just like any other:

```
> myList$animals[1]

[1] "gator"

> myList$animals[2:3]

[1] "anhinga"   "armadillo"

> myList$rocks[2]

[1] "agatized coral"
```

And so on. Although all 3 components of the `myList` object are character vectors, we can make lists that contain all sorts of data combinations:

```
> myMatrix <- matrix(1:9, nrow = 3, ncol = 3)
> myMatrix

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

> myLogicalVector <- c(TRUE, TRUE, TRUE, FALSE)
> myNewList <- list(stooges = c("larry", "curly", "mo"), theMatrix = myMatrix,
+     theVector = myLogicalVector)
> myNewList

$stooges
[1] "larry" "curly" "mo"

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

$theVector
[1]  TRUE  TRUE  TRUE FALSE
```

And you can access components of this list just like the previous example:

```
> myNewList

$stooges
[1] "larry" "curly" "mo"

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

$theVector
[1]  TRUE  TRUE  TRUE FALSE
```

Each of the components of this list behaves like one of the fundamental datatypes we have already learned. If you want to access the 3rd row and 2nd column of the `myNewList` component *theMatrix*, you simply do the following:

```
> myNewList$theMatrix[3, 2]

[1] 6
```

because `myNewList$theMatrix` **IS** a matrix. Likewise, you can take the 2nd row of the same matrix:

```
> myNewList$theMatrix[2, ]

[1] 2 5 8
```

Exercise 1: Make a list, using the `list` function as above. Your list should contain 4 components: (1) A 5 x 5 matrix of random numbers from a normal distribution; (2) a character vector of your 5 favorite animals; (3) a list of at least several names of current or former pets. Demonstrate that you can access each component of your list. Lists are super

useful for many reasons. Many of the "objects" returned by common functions are in fact lists. For example, look at the help on the `lm` function. You see that the "return value" is actually a list. Never mind that it is of "class" `lm` for the moment. Let's work with this, revisiting the animals dataset:

```
> library(MASS)
> data(Animals)
```

Now we will repeat the simple regression exercise from last week, looking at the logarithm of animal brain size as a function of the logarithm of body size. Check the `mode` (hint: this is a function) of the fitted model object:

```
> library(MASS)
> data(Animals)
> myFit <- lm(log(brain) ~ log(body), data = Animals)
> myFit

Call:
lm(formula = log(brain) ~ log(body), data = Animals)

Coefficients:
(Intercept)    log(body)
      2.555        0.496
```

`myFit` is now a list, and we can access all of the components using the `$` operator. What are the components of `myFit`? Looking at the helpfile, you should be able to see all the components that should be present. We can now do:

```
> myFit$coefficients

(Intercept)   log(body)
  2.5548981   0.4959947
```

gives us the slope and intercept of our fitted model. E.g., `myFit[2]` is the slope (the coefficient for the log(body) term in our fitted model). You can also immediately see the names of all the list components using the `names` function. Do this!

Exercise 2: Generate a plot of log(brain) as a function of log(body), as you did last week. Now, rather than using the function `abline`, I want you to plot your fitted model using the function `lines` . `lines` will draw a line between 2 or more points. You have to specify a vector of at least 2 x-coordinates and 2 y-coordinates. `lines` is infinitely more useful than `abline`, which is at best a wimpy tool for very exploratory data analysis. You may have to look at help for `lines`. This will probably require several (short) steps. Note that `lines` can only plot a line *by adding to an existing plot.*

Exercise 3: Using your fitted model object, plot the *residuals* of the log-brain by log-body relationship as a function of the log-transformed body size. How does it look? Do you think this analysis is valid? Can you think of anything obvious that might explain any outliers in this relationship?

Exercise 4: Now for a final look at lists, in a context that will become very important to us as we continue to learn about phylogenetics in R . Read in the agamid tree that we were working with last week. Look at the components of the phylogenetic tree as stored in R . You can find these components using `names` and/or by looking at the help on `plot.phylo` . Understanding something about these components will be critical to working effectively with trees (we will cover this in much detail down the road!).

2 Writing output to screen (and to file!): `cat`

`cat` is one of the more important functions you will learn as an R programmer. `print` is basically a limited version of `cat` and we won't worry much about it.

```
> x <- 212
> y <- 1:5
> z <- c("alpha", "beta", "gamma")
> print(x)
> print(y)
> print(z)
> cat(x)
> cat(y)
> cat(z)
```

Compare this instance of `cat` to the following:

```
> cat("x")
> cat("y")
> cat("z")
```

What is going on?

Some types of characters are so special that, to print them, we have to use *escape characters* to print them effectively. These include the tab space and the newline character. Here are several instances of `cat`. Observe the differences between these:

```
> cat(y, sep = "")
> cat(y, sep = "\t")
> cat(y, sep = "\n")
> cat(y, sep = "xxxx")
```

`cat` can handle multiple arguments:

```
> cat(y, z, "this is a silly exercise", sep = "\n")
```

And finally, `cat` can write data to file:

```
> x <- rnorm(50)
> outfilename <- "myOutfile.txt"
> cat(x, file = outfilename, sep = "\n")
```

Exercise 5: Repeat this exercise, but write (to file) the sequence of integers between 1 and 10000, separated by a comma.