Short Introduction to R

Jul 2, 2015

Short Introduction to R

Liam Revell and Janet Buckner

30 de junho de 2015

Some preliminary topics

1. R is both a scientific computing software package and a programming language. R is distributed free & open source under the GNU General Public License. Although in principal anyone can download and modify the source code of R, the development & distribution of R is primarily done by a set of core maintainers called the R Core Team.
2. The fact that R is also a programming language might intimidate some users; however most R users are not programming. However - in its most typical flavor, R is command line driven. That means to get results, we need to type them into a command-line interface; rather than selecting options from drop-down menus.
3. The two main components of an R work session are objects and functions. Objects are variables, data, and results that we have input, uploaded, or created and are stored in the memory of our currently active R session.

Functions are a special type of object in R that takes one or more arguments and does something (e.g., creates a new object, opens a plotting object, writes a file, etc.).

4. R is built on contributed packages, which mostly contain new R functions. Almost all but the most basic functionality within R is created by contributed packages - of which there are literally thousands. Most contributed packages are stored in public repositories - principally the Comprehensive R Archive Network (CRAN).
5. Finding and Setting your Working Directory

Before you get started on an R session, you want to make sure you are accessing the files you want and saving any output to a desired location. This is easily done by setting your directory to the folder containing the files you want to manipulate. You can see what directory you are currently in by using getwd(). I'm assuming you have a folder for this workshop and will set the directory there:

```
getwd()
setwd("~/Desktop/PCM_workshop2015")
getwd()
```

2 Creating Objects in R

Objects in R are created by assigning values to names. In order to perform further manipulations on an object, it must be named. This can be done in the following ways: use the function assign()

```
assign("n", 15)
```

or use the operator "<-" which has the sam function as assign(). It can be thought of as a shortcut.

```
nn <- 15
no <- c(1, 2, 3, 4, 5)  #What is the function of “c()”?
oo <- c(1:5, 7, 19) #What is the function of “:”?
```

Object names are case sensitive:

```
OO <- oo + 3
```

A single object name cannot be used for more than one object at a time. If used again the original value will be replaced.

```
oo <- 45
```

After creating an object, you can enter the name into the R console to see what you’ve created. You can also scroll through the previous commands you’ve entered using the up arrow key.

```
no
oo
OO
```

We may want a reminder of all the objects in our current R session. The funtions ls() and objects() will report the named objects we have created. Try it.

```
ls()
objects()
```

Maybe you’ve forgotten what your named objects are. You can get more information about all the objects by using ls.str() or even more detailed descriptions of single objects using str().

```
ls.str()
str(oo)
```

We can look at only objects with a specific pattern as well:

```
ls.str(pat="o")
```

How did we know which arguments could be put into ls() or ls.str()? Well, if these were functions that we were already familiar with and we just wanted to remind ourselves, we could use args():

```
args(ls)
args(str)
args(args)
```

However, if we are just learning to use a function, it would be useful to have more information about the function and its specific arguments. For this we can use help() to look at a function or help.search() to search R help for any functions or topic related to our query:

```
help(ls)
help.search("phylogeny")
```

Object Types There are five main types of data objects in R: vectors, factors, matrices, data.frames and lists. All data objects have attributes and values. Understanding the type of objects you are working with is important, as it will determine what kinds of manipulations you can perform on them. There are many ways to find out what type of objects are in our workspace, including is._() functions which will give a logical response. Try these with one or more of your objects:

```
is.vector(oo)
is.factor(oo)
is.matrix(oo)
is.data.frame(oo)
is.list(oo)
```

So far, we have created and manipulated vectors. But what is a vector in R?

Vectors

A vector is a series of elements of the same type. It has two attributes: mode and length. We can obtain the mode nad length of a vector by using mode() and length() respctively. Let's look at some different modes of vectors.

```
xx <- 1:5
mode(xx)
length(xx)

yy <- c("order", "superfamily", "family", "genus", "species")
mode(yy)
length(yy)

zz <- c(FALSE, FALSE, TRUE, FALSE, TRUE)
mode(zz)
length(zz)
```

Logical vectors can also result from logical operations:

```
xx>=3
```

We can access individual element in a vector using numerical indexing. For example:

```
xx[3]
yy[c(1:3)]
yy[c(1,3)]
zz[c(1:3, 5)]
```

We can also remove elements with indexing using the negative "-":

```
xx[-3]
```

Or with logical indexing:

```
xx[c(TRUE, TRUE, FALSE, TRUE)]
```

Logical indexing combined with functions is a useful way to select some data from a vector, but not others:

```
x <- runif(n=6, min=0, max=10) # remember that if you are unsure what a function is
doing, use help()
  which(x>=5)
    y <- x[(!is.na(x)) & x>7]   #What is this command doing to vector x?
```

We can perform functions on or between entire vectors. Try some of these:

```
OO <- oo + 3
oo - OO
mean(oo + OO)/2
max(OO)
min(OO)
sum(oo)
```

Vectors and other R objects sometimes (but not always) have a third attribute, names. In phylogenetic analysis, our vectors will frequently have names.

```
X <- 1:5
names(X) <- zz  #Remember that you can always check the attributes of an object if y
ou are unsure by using attributes() or str()
```

Factors

Factors provide compact ways to handle categorical data. A factor is derived from a vector, but has the additional attribute of levels. Let’s create some factors.

```
ff <- c("Male", "Male", "Female", "Female", "Female")
str(ff)
ff <- factor(ff)
str(ff)
```

Another way to consider factors:

```
ff <- c(0,0,1,1,1)
ff <- factor(ff)
levels(ff) <- c("Male","Female")
```

We can check what the levels are in case we forget them with the same function levels() or with our handy str().

```
levels(ff)
str(ff)
```

Matrices

A matrix is a vector arranged in a tabular way. It has the additional attribute dim. This can be seen in the following example:

```
X <- matrix(1:9, 3, 3)
X
```

We could also get the same result this way:

```
X <- 1:9
dim(X) <- c(3,3)
```

In both these examples the values 1 – 9 are arranged by column. If we want them arranged by row, we simply tell R:

```
X <- matrix(1:9, 3, 3, byrow=TRUE)
```

Let's look at some more numerical indexing, this time with matrices. What is the result of each of the following commands?

```
X[3,2]
X[ ,3]
X[2, ]
```

Data Frames

A data frame is a very important type of object in R. It looks like a matrix but is actually stored as lists. It is the data object that is created by reading in information such as spreadsheets from a file. Let’s have a look:

```
    Y <- data.frame(zz, y=1:5, x=5:1)
    Y
    Y$zz
    Y$y
```

Lists An R list is the most general data structure and is an object consisting of an ordered collection of objects known as components. It is not necessary that the components be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a function and even a phylogenetic tree. Lists can be created with the function list():

```
L <- list(zz=zz, 1:2, Y)
length(L)
names(L)
```

The components of a list can be accessed in the following way:

```
    L[[1]]
    L$z
```

3. Reading Information into R from Files

Much of the time you are working with phylogenetic methods in R, you will need to read in data from multiple types of files. The very minimum will likely be a data file that describes some character in your species of interest, and the phylogenetic tree with information on the relationships between said species.

There are multiple functions for reading in character data, normally stored in some kind of tabular format. Two of the most commonly used functions are read.table() and read.csv(). These functions correspond to files of a certain type as implied by the name read.csv() for example. Let's try reading in some data.

Information read into R using the read.table() function should have a header (the names for each variable) and row names (this will typically correspond to species names for phylogenetic studies). If your data does not have a header, you can tell R by setting header=FALSE, see below. Lets get started:

We can preview our data before reading it into R using readLines():

```
cat(readLines("~/Desktop/PCM_workshop2015/anole.data.txt", 10), sep= "\n")
```

Remember to name your data when you read it in!

```
anole_data <- read.table("~/Desktop/PCM_workshop2015/anole.data.txt", header=TRUE, r
ow.names=1)
```

We can have a preview of the data after reading it in by using head():

```
head(anole_data)
```

We just read in a tab-delimited text file. Another common format is .csv files, or comma separated values. We can read this in with read.csv():

```
anole_data <- read.csv("~/Desktop/PCM_workshop2015/anole.data.csv", header=TRUE, row
.names=1)
head(anole_data)
```

Basic Scripting in R

Now that we've learned how to get data into R, we can start to manipulate it with basic commands. Let's say we are interested in the average values for all the morphological variables in our dataset. First, we will create an empty vector:

```
averages <- vector(mode = "numeric", length=ncol(anole_data))
```

Then, using a for loop, we will tell R to calculate the average of each variable (column) in "anole_data" and store the information in the empty vector "averages" that we created:

```
for (i in 1:ncol(anole_data)) averages[i] <- mean(anole_data[ ,i])
names(averages) <- colnames(anole_data)
averages
```

Another way to do this is to use the function apply(), which will perform a function over multiple entities, in this case the columns in anole_data:

```
averages <- apply(anole_data, 2, mean)
averages
```

Sometimes, R has functions that already do exactly what you want:

```
averages <- colMeans(anole_data)
averages
```

We covered a simple example of a for loop in the exercise above. But we can loop over multiple lines of code using slightly different syntax. Let's explore this by adding the print() function to our loop which can be handy in debugging code:

```
averages <- setNames(vector(mode="numeric", length=ncol(anole_data)), colnames(anole
_data)) # here again    we have created our empty vector, but this time we have assi
gned names to the vector at the same time
for (i in 1:ncol(anole_data)){
    averages[i] <- mean(anole_data[,i])
    print(averages[1:i])
  }
```

We have talked a lot about functions that are already built into R. But you can make your own functions! We saw above that conveniently, the function colMeans exists for a task we might frequently want to perform on our data. Well what if it didn't exist and you could create the function yourself to avoid writing that pesky for loop every time? Let's recreate it with function() under the name col_means:

```
col_means <- function(x, na.rm=TRUE){
    obj <- vector(mode="numeric", length=ncol(x)) #create and empty vector equal in
length to columns
    for (i in 1:ncol(x)) # for each column in the dataset
        obj[i] <- sum(x[,i], na.rm=na.rm)/sum(!is.na(X[,i])) #obj of the current ind
ex is the sum of the column/the number of values with not equal to NA
    setNames(obj, colnames(x)) # report the names of the means stored in obj as the
column names in x
  }
```

4. Writing Information from R to Files

After performing various tasks on our data in R, we might want to store it in a file. We can do this using various functions in R. Let's use write.table() to save our averages to a file:

```
write.table(averages, file = "anole_averages")
```

For illustrative purposes, let's say we wanted to look at slopes between our morphological variables and we wanted to save the figure as a pdf.

```
pdf()
plot(anole_data$SVL, anole_data$HL)
dev.off()
```

share

(https://facebook.com/sharer.php?u=http://lukejharmon.github.io/ilhabela/instruction/2015/07/02/Intro_to_R/)
(https://twitter.com/intent/tweet?text=Short Introduction to R&url=http://lukejharmon.github.io/ilhabela/instruction/2015/07/02/Intro_to_R/)
(https://plus.google.com/share?url=http://lukejharmon.github.io/ilhabela/instruction/2015/07/02/Intro_to_R/)