Lecture 8

Lists, continued

Most of the things that we know work on vectors, work on lists.

Let us work on an example: working with populations of unequal sizes

```r
> pop = list(1:10,1:20,1:50)
> pop
> lapply(pop, sample, rep=T)
> lapply()
```

`lapply()` is like `sapply()` for vectors and `apply()` for matrices. It calls a function for each entry of the list.

Actually, `sapply()` and `lapply()` are the same, except that `lapply()` always returns a list, and `sapply()` returns a matrix if all results are of the same size.

```r
> sapply(pop,sample, rep=T)
> pop = sapply(pop,sample, rep=T)
> pop
> sapply(pop, var)
> lapply(pop, var)

Or something more interesting: let us track how many different individuals there are

```r
> sapply( pop, function(x) length(unique(x)))
```

`unique()` takes out just the unique elements of a vector

```r
> pop[[1]]
> unique(pop[[1]])

Interesting. Let us track the number of different individuals over time

```r
> res = c(); pop = list(1:10,1:20, 1:50, 1:100)
> for( i in 1:100) {
>   pop = lapply(pop, sample, rep=T)
>   len = sapply( pop, function(x) length(unique(x)))
>   res = rbind( res, len)
> }
> plot(0:1,0:1,xlim=c(0,100),ylim=c(0,1),type="n")
> res = t(res)
> res = res/(c(10,20,50,100))
> res
> apply(res,1,lines)
```

We see that sampling quickly reduces the diversity in the population (we saw this before...)
Just as `for` can loop over vectors, it can loop over lists:

```r
> for(i in pop) print( i)
```

Let us restore the original `pop`

```r
data(pop) # Load the data
> pop = list(1:10, 1:20, 1:50)
> pop

I’d like to convert it to a matrix that has, as the first column the individual, and the second the group.

```r
> index = sapply(1:3, function(i) rep(i, length(pop[[i]])))
> index
> cbind( unlist(pop), unlist(index))
> pop.mat = cbind( unlist(pop), unlist(index))
```

`tapply()` is another way to apply a function to data. `tapply` takes a vector and an index, and applies the function to the different categories of the index.

```r
> tapply( pop.mat[,1], pop.mat[,2], c)
> tapply( pop.mat[,1], pop.mat[,2], length)
```

We just got back the original list.

```r
> tapply( pop.mat[,1], pop.mat[,2], length)
   1 2 3
   10 20 50

> tapply( pop.mat[,1], new.place, c)
> tapply( pop.mat[,1], new.place, length)
```

So, now, we could, for example, do migration.

```r
> pop.mat[,2]
> sample( -1:1, length(pop.mat[,2]), rep=T, p=c(0.1,0.8,0.1))
> pop.mat[,2] + sample( -1:1, length(pop.mat[,2]), rep=T, p=c(0.05,0.9,0.05))
> new.place = (pop.mat[,2] + sample( -1:1, length(pop.mat[,2]), rep=T, p=c(0.1,0.8,0.1))) %% 3
> tapply( pop.mat[,1], new.place, c)
> tapply( pop.mat[,1], new.place, length)
```

This method of doing migration doesn’t keep the population sizes constant, and the rate of migration between the populations doesn’t depend on the population.

**A comment about indexing**

We saw how to index vectors, matrices, lists.
> m = matrix(1:40,4,10)
> m
> m[1:2,-3]
>
It is possible to serially index again and again:

> m[-1,-1]
> m[-1,-1][1:2,3:5]
> m[-1,-1][1:2,3:5][2,2]
>
Similarly, when the matrix is in a list, we can access it in the same way

> l = list(m,m,m)
> l
> l[[1]][1:2,3:4]
> l[1:2][[2]][-1,-1]
>
Strings

As we saw, strings in R can be created like this:

> a = "Hello"
>
To insert various control characters into the string, we use backslash:

> a = "Hello\n"
> a
> cat(a,a,a,a)
> a = "Hello\t"
> a
Hello Hello Hello Hello > cat(a,a,a,a,"\n")
> a = "Hello \"Michael\"
> a
Hello "Michael"> cat(a,"\n")
> a = "hello \"
> a
> cat(a,"\n")
>
R is actually quite powerful for handling strings. To manage strings in R, one has to master “regular expressions”.

Let us look again at the results of ms:

> system("./ms.osx 20 3 -t 1 -r 1 10000 >out.txt")

>
When we want to read more complicated data, we can use the `readLines` command:

```r
> lines= readLines( "out.txt"
> length(lines)
> lines
>
```

Each stretch of results starts with `//`. Let us look for that

```r
> grep("//",lines)
> lines[ grep("//",lines)]
> i=grep("//",lines)
> i
> lines[i+3]
>
```

```r
> i
> length( lines )
> cbind( i, c(i[-1], length(lines)) )
> rmat = cbind( i, c(i[-1]-2, length(lines)) )
> res = apply( rmat, 1, function(r) lines[ r[1]:r[2] ] )
> strsplit( res[-(1:3),1],split="")
> res[,3]
> res[[2]]
> res[[3]]
>
```

The function `nchar()` counts the length of a string, as `length()` does for a vector or list

```r
> nchar("hello")
> res[,1]
> nchar(res[,1])
> nchar(res)
>
```

A somewhat strange feature of R is `char` on NA:

```r
> res[2:5,2]
> x = res[2:5,2]
> x
> x[3] = NA
> x
> nchar(x)
> nchar( c("a","b",NA,"c"))
>
```

Strange.
**substr()**

Another important function for handling strings in R is `substr()`. It allows you to take or set a certain part of a string.

```r
> pos
> substr(pos,1,5)
> x=pos
> x
> substr(x,1,10) = "location"
> x
> substr(x,1,4) = "LOC"
> x
> x=pos
> x
> substr(x,1,3) = "PO"
> x

> pos
> 
```

How can we get at those lines?

Now, let us say, we also want to get the positions

```r
> grep("positions:",lines)
> pos = lines[ grep("positions:",lines) ]
> pos
> pos
> spos = strsplit(pos," "); spos
> lapply( spos, function(i) as.numeric(i[-1]) )
>
```

**gsub**

With `gsub`, you specify what you want to find, and what you want to replace it with.

For example:

```r
> pos
> gsub("positions: ","",pos)
>
```

We got rid of the first string.

Let us say we would like to get all entries that have a ‘:’ in them

```r
> res = lines[ grep(":",lines) ]
```

92 msec
Now, we would like only the string before ":".

```r
> gsub("(.*)::.*","\1",res)
```

In a regular expression, . (dot) matches anything. * means 0 or more.

```r
> res
> gsub(".*","",res)
```

^ matches the start of the string, $ the end:

```r
> res
> gsub("^.*","",res)
> gsub("^.*","",res)
> gsub(".*$"," end",res)
```

We can put certain characters we’d like to match in []:

```r
> gsub("[0123]\",","",res)
> gsub("[0-9]\",","",res)
> gsub("[0-9.]\",","",res)
> gsub("[0-9 \:]\",","",res)
```

If we put an expression in (), we can refer to it later

```r
> gsub("([^0-9]*[.,][0-9]*)","{\1}"",res)
> gsub("([^0-9]*[.,][0-9]*)([^0-9]*)","{\1}.{\2}"",res)
```

Just a comment. `grep()` found those strings that match a pattern. If you want to know where the pattern matched, you should use `gregexpr()`:

```r
> gregexpr("[.][0-9]*",x)
```

**Libraries and packages**

R has lots of libraries that can do various stuff
In addition to all the installed packages, we can install packages from CRAN.

```r
> install.packages("phyclust")
> help.start()
> ?ms
> (ret.ms <- ms(nsam = 5, opts = "-t 1 -T"))
> (tree.anc <- read.tree(text = ret.ms[3]))
> ret.ms
> plot(tree.anc)
> 
> 
> 
> 
> ```