Class 7: Functions, Loops, and Conditional Statements – writing scripts

Goals:

(1) introduce the basics of custom functions
(2) explore the use of a variety of conditional statements
(3) introduce the use of loops

A *function* is a R code that performs a specific task. You use built-in functions all the time (e.g., read.table, lm, plot). Functions generally take one or more objects as input, and then produce an object as output. You can create your own functions to do one of two things:
-make a simple function to apply a complicated formula you use regularly
-organize the output components from a multi-step analysis into a user-friendly format.

Conditional statements allow you to accomplish complex recoding and subsetting tasks. E.g., If the diameter is less than 10 cm, then label as "juvenile"; otherwise, label as "adult".

Loops allow you to do repetitive rules-based applications of functions such as:
-For each stem on a plot, find the distance to the nearest stem of the same species.
-Randomization test when assumptions for other tests are not met
-Iterative matrix multiplication to project population growth over time

#Sample data sets:

```
#2007 data of stem from the UCSC Forest Ecology Research Plot
ferp<-read.csv("http://people.ucsc.edu/~ggilbert/Rclass_docs/FERP07data.csv")

#A small data set of species counts
divdat<-
data.frame(spp=c("a","b","c","d","e","f"),freq=c(4,5,12,8,1,2))

#END OF SAMPLE DATA SETS
```

Notes on efficiency

(1) Anytime you can apply a function to an entire vector, do so.
(2) Functions compiled in C (e.g., `sapply`) are much faster that `for` loops
(3) Operations on matrices are much faster than operations on data frames
(4) If you want to see how long it takes to do something in R, embed it in `system.time()`

(RE)INTRODUCTION TO FUNCTIONS

We had a brief introduction to functions in Class 3, but I'll repeat that here, and extend it. There are two steps. First you create a function, and initialize it in your workspace. Then you can use it in that workspace, calling it as you would any function. If you save that workspace, that function remains available if you re-open that workspace later.

Creating a function.

Recall from Class 3, that to create a function, use this structure:

```
NameOfMyFunction<-function(dataobjects)
{
a<-function1(dataobject) #do something to your dataobject
b<-function2(dataobject) #do something else
a*b #the result of the last line is what is returned by the function
}
```

#note that dataobject is a placeholder name – it takes the value of what you put in the ()

Dataobjects are inside ( ), following the call to *function.* Multiple data objects are separated by commas. Give these variables meaningful, but short names.

What the function actually DOES to the data objects is placed inside curly brackets { }.

The <u>order</u> of the data objects is the order in which you will put the real variables when you apply your function. That is, there is *positional matching*. Alternatively, you can specify which object is which by including specifying it by name when you call the function.

EXAMPLE

Let's make a function to calculate the volume of a rectangular solid. The function must accept values for height, width, and depth of the solid, multiply them together, and return the value. Let's call our function *Solid.Volume*.

```
#Create the function like this
Solid.Volume<-function(h,w,d){h*w*d}

#Now apply the function using positional matching
Solid.Volume(2,4,5)

#Or by directly specifying each element
Solid.Volume(h=2,w=4,d=5)

#Or apply to data in a data frame
rectsol<-data.frame(height=c(1,3,5),width=c(2,3,5),depth=c(2,5,7))
rectsol
rectsol$volume<-Solid.Volume(rectsol$height,rectsol$width,rectsol$depth)
rectsol
```

Some important notes about functions:

(1) Once the data objects are passed to a function, they are internally called by the names in the function definition (here h,w,d) and not by their names outside the function (height, width, depth). This makes the function universally applicable to any object of that type.

(2) You can have multiple lines of code within a function, but anything calculated there is completely *internal* to the function. It does not exist outside the function. Only the result of the last line is returned by the function.

```
#A function that does NOT work
Solid.Volume<-function(h,w,d){vol<-h*w*d}
Solid.Volume(2,4,6)  #testing
vol
Error: object 'vol' not found
```

```
#Make this function work by calling vol as the last line
Solid.Volume<-function(h,w,d)
{
vol<-h*w*d
vol
}
Solid.Volume(3,5,7) #testing
[1] 105
```

(3) A variable assigned inside a function is internally accessible for further calculations:

```
Sqrt.Solid.Volume<-function(h,w,d)
{
vol<-h*w*d
sqrt(vol)
}
Sqrt.Solid.Volume(2,2,9) #testing
[1] 6
```

(4) Create a list to return multiple objects from a single function.

```
Sqrt.Solid.Volume<-function(h,w,d)
{
vol<-h*w*d
sqrt<-sqrt(vol)
list(paste("volume=",vol),paste("square root of volume=",sqrt))
}
Sqrt.Solid.Volume(2,2,9) #testing
[[1]]
[1] "volume= 36"

[[2]]
[1] "square root of volume= 6"
```

A few examples of Functions

#Find the **distance between two points** that are described by x,y coordinates

```
pythag<-function(x1,y1,x2,y2) {sqrt((x1-x2)^2+(y1-y2)^2)}
pythag(2,4,3,7) #testing
```

#Calculate the **Shannon-Weiner Diversity Index**
#H' is the negative of the sum of the p* ln(p) where p is the proportion of individuals in the sample
#There is then a correction factor –[(NumSpecies – 1)/(2*NumIndivs)]

```
Hprime<-function(counts)
{
NumIndivs<-sum(counts)  #find the total number of individuals
ps<-counts/NumIndivs  #divide each species count by NumIndivs to get p
plnp<-ps*log(ps)    #calculate plnp for each species
NumSpecies<-length(counts)   #gets the number of species
-(sum(plnp))-((NumSpecies-1)/(2*NumIndivs))  #put it all together
}

Hprime(divdat$freq) #testing
```

#Maybe you want to **create a more descriptive output** for H'

```
Hprime<-function(counts)
{
NumIndivs<-sum(counts)  #find the total number of individuals
ps<-counts/NumIndivs  #divide each species' count by NumIndivs to get p
plnp<-ps*log(ps)    #calculate plnp for each species
NumSpecies<-length(counts)   #gets the number of species
H<- -(sum(plnp))-((NumSpecies-1)/(2*NumIndivs))  #put it all together
out1<-paste("This sample included",NumSpecies,"species and
",NumIndivs,"individuals.")
out2<-"The Shannon-Weiner diversity index, calculated as -Sum(pi*ln(pi)) -
((S-1)/(2*N)), pi is the proportion of all individuals N that are species i,
and S is the total number of species"
out3<-paste("H'=",round(H,4),".")
paste(out3,out1,out2)
}
Hprime(divdat$freq)
```

```
[1] "H'= 1.5433 . This sample included 6 species and  21 individuals. The
Shannon-Weiner diversity index, calculated as -Sum(pi*ln(pi)) - ((S-
1)/(2*N)), pi is the proportion of all individuals N that are species i, and
S is the total number of species"
```

Conditional Statements WHICH, IF, and IFELSE

```
ferp<-read.csv("http://people.ucsc.edu/~ggilbert/Rclass_docs/FERP07data.csv")
ferp2<-ferp[1:100,c(1,2,4,5,6)]  #get just the columns of FERP data you need
head(ferp2,4)
  tag   code  dbh east north
1   2 QUERPA   31  1.9   6.3
2   3 PSEUME  378  0.6   6.2
3   4 QUERPA   20  0.5   6.8
4   5 SEQUSE 1420  3.1  13.7
```

(1) WHICH

```
which(ferp2$code=='QUERPA')  #returns the line numbers where code==QUERPA
[1]  1  3  5 21 54 63 71 72 73

QUERPA_only<-ferp2[which(ferp2$code=='QUERPA'),] #subsets all lines of QUERPA

#WHICH works across the whole vector at once
```

(2) IF

```
if (condition) {action}
x<-100
if (x<200) {size<-"small"}
x; size

#IF does not work across the whole vector at once
if(ferp2$code=='QUERPA'){ferp2$tree=="Oak"}
logical(0)
Warning message:
In if (ferp2$code == "QUERPA") { :
  the condition has length > 1 and only the first element will be used

#Loop it. Instead, you need to call each line separately in a for loop
ferp2$tree<-"not"
for (i in 1:length(ferp2$code))
{if(ferp2$code[i]=="QUERPA"){ferp2$tree[i]<-"Oak"}}
```

(3) IFELSE

```
#ifelse (condition, ActionIfTrue, ActionIfFalse)
x=243
ifelse(x>=100,"large","small")      # [1] "large"
x=96
ifelse(x>=100,"large","small")      # [1] "small"
```

(4) You can **include conditional statements using "return" inside a function**
For instance, if your function only applies to a particular range of numbers, you can test first if your data objects are within that range. If they are not, return "NULL"; If they are, continue to calculate your function.

```
square.root<-function(x)
{if (x<0) return (NULL)
sqrt(x)}
square.root(9)  #returns 3
square.root(-4) #returns NULL
```

Introduction to Loops

Loops are really common in many programming languages, like FORTRAN, C++, MATLAB, etc. R has a number of great functions (vector functions, sapply, etc.) that do the same thing as a loop, but more efficiently and elegantly. For many R programmers, using loops is poor form, because they are inelegant. However, sometimes loops allow you to do rather complex actions in a way that is intuitive and easy to set up, and useful. Order of preference should be: *vector function > sapply or apply > loops.*

A simple example: *Take a column of numbers in a data frame, and add a new column with the square of that value.*

```
rd<-data.frame(ran_num=runif(25)) #create data frame with random numbers
```

Vector function approach (CLEAN AND FASTEST)

```
rd$squared<-rd$ran_num^2
```

Custom function and sapply (CLEAN AND FAST)

```
square<-function(x) {x^2}  #create a function called square
rd$squared<-sapply(rd$ran_num,square)
```

Loop approach (SLOW, BUT FLEXIBLE)

```
for (i in 1:25) {rd[i,"squared"]<-rd[i,1]^2}
```

GENERAL STRUCTURE OF LOOPS using FOR

Loops take a similar structure to *function*, with a ( ) and { } section

for (while certain conditions apply) {what to do in the loop}

Most often, the "certain conditions" are a counter for a range of repetition you set.

An example using "rd" data frame of 5 random numbers:

```
rd<-data.frame(ran_num=runif(5)) #create data frame with random numbers
```

Use a loop to take each of the 5 lines (for i from 1 to 5) in turn and square rd$ran_num.

```
for (i in 1:5) {rd[i,"squared"]<-rd[i,1]^2}
rd
    ran_num    squared
1  0.3958177 0.15667162
2  0.7372523 0.54354092
3  0.1981701 0.03927139
4  0.9938110 0.98766031
5  0.5583553 0.31176061
```

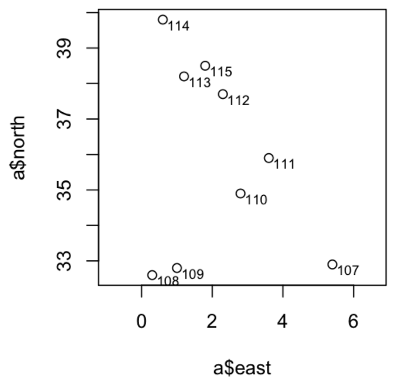
```
What this does is:
rd<-data.frame(ran_num=runif(5))
#Enter the loop
rd[1,"squared"]<-rd[1,1]^2
rd[2,"squared"]<-rd[2,1]^2
rd[3,"squared"]<-rd[3,1]^2
rd[4,"squared"]<-rd[4,1]^2
rd[5,"squared"]<-rd[5,1]^2
#exit the loop
```

An example where a loop would be more useful.

```
Imagine you have a set of individuals, each with a tag number, and
each mapped with x,y coordinates.   Create a small data frame subset
of the ferp data to play with.

a<-ferp[102:110,c(1,5,6)] #get data subset

plot(a$east,a$north,asp=1,cex=1.5,pch=19)
text(a$east-.2,a$north-.2,a$tag,pos=4)
```



For each of the points we want to find the distance to the nearest point. To do this, conceptually, we want to:

1. Take the locations of the first point.
2. Use our pythag function (above) to calculate the distance from that point to every other point.
3. Find which point is closest (taking care to exclude the distance from a point to itself).
4. Record the tag number and distance of the nearest neighbor.
5. Go back to (1) and do the same for the second point.
6. Continue looping until you have all the points.

Your loop needs to run from 1 to the number of individuals in the data frame. You can find how many there are using the length function (i.e., length(a$tag)), and including this in " i in 1: x" part of the function directly.

```
for (i in 1:length(a$tag))
{
east1<-a$east[i]    # gets the east coordinate for the ith point.
north1<-a$north[i]    # gets the north coordinate for the ith point.
tag1<-a$tag[i]   # get the tag number for the ith point.
#create a temp variable in the data frame with the distance from the ith
point to every other point
a$tempdist<-pythag(east1,north1,a$east,a$north)
#we don't want the distance to itself to appear to the be closest neighbor,
so set it to NA
a$tempdist[i]<-NA   #the tempdist of the focal point is NA
#use which function to get the index number of the point with the minimum
tempdist
#na.rm-TRUE says to ignore the point with tempdist=NA
mindist<- min(a$tempdist,na.rm=TRUE)
#then take  tag number of that record, and put
in mintag
mintag<-a$tag[which(a$tempdist==mindist)]
a$nntag[i]<-mintag
a$nndist[i]<-mindist
}
a<-a[,-4]  #get rid of column "tempdist"
```

```
> a
    tag east north nntag    nndist
102 107  5.4  32.9   110 3.2802439
103 108  0.3  32.6   109 0.7280110
104 109  1.0  32.8   108 0.7280110
105 110  2.8  34.9   111 1.2806248
106 111  3.6  35.9   110 1.2806248
107 112  2.3  37.7   115 0.9433981
108 113  1.2  38.2   115 0.6708204
109 114  0.6  39.8   113 1.7088007
110 115  1.8  38.5   113 0.6708204
```

Alternatives to `for` in making loops.

(1a) You can use `while`, instead of for
`While` requires establishing a counter that is incremented with each iteration of the loop.

```
First.Five.Powers<-function(x)
{
i=1   #set the first value of your counter
d<-c(rep(NULL,5))  #create an empty vector of length 5 to hold values
   while(i<=5)
  {
    d[i]<-x^i   #raise the value of x to the current value of i
    i<-i+1        #increment your counter
   }
d  #call the vector of 5 values to return this as the output of the function
}
First.Five.Powers(3)  #testing
```

(1b) Another example using `while`
Here we will find the even numbers in a vector of numbers

```
somedata<-c(2,3,4,5,7,8,9)  #here is a vector of numbers to use.

Find.Evens<-function(x)
{
i<-1   #set your counter to 1
N<-length(x)   #find the number of elements in your vector
evens<-NULL   #establish an empty vector
   while(i<=N)
   {
    if (x[i]%%2==0) {evens<-append(evens,x[i]) }   #if the modulo=0 keep it
    i<-i+1   #advance your counter
    }
evens   #return the list of even numbers
}
Find.Evens(c(2,3,4,5,6)
```

(1c) And another example using `while`.
Calculate generations of exponential growth to a pre-defined limit and then stop.

```
r=1.5; N=10; grow<-NULL  #set value of r and initial N, and create empty
vector called grow
while (N<100000){grow<-append(grow,N); N<-r*N}  #in each iteration, append
new N to grow
grow
```

#personal note: For cases 1a and 1b, I would use a `for` *loop. 1c is easier with* `while`.

(2) You can also use the **`repeat`** and **`break`** functions to create loops. However, they make me nervous because it is easy set up infinite loops if you are careless. I avoid them, because you can do the same thing with **`while`**.

Using loops for repetitive tasks

Get the FERP data from web site. These data include east and north coordinates for each of 8180 stems from 31 woody species, plus the diameter at breast height for each stem.

```
ferp<-read.csv("http://people.ucsc.edu/~ggilbert/Rclass_docs/FERP07data.csv")
ferp<-ferp[,c(1,2,4,5,6)]  #trim down to variables needed here
head(ferp)
  tag   code  dbh east north
1   2 QUERPA   31  1.9   6.3
2   3 PSEUME  378  0.6   6.2
3   4 QUERPA   20  0.5   6.8
4   5 SEQUSE 1420  3.1  13.7
5   6 QUERPA   74  4.7  19.2
6   8 LITHDE   31  5.6  14.8
```

```
NumStems<-length(ferp$tag)    #8180 stems
```

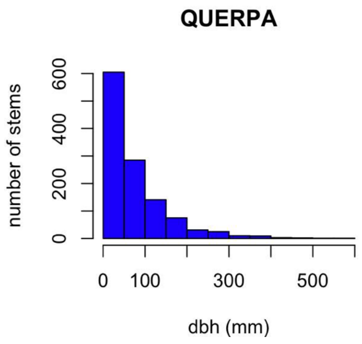
#Extract the data for QUERPA

```
querpa<-ferp[which(ferp$code=="QUERPA"),]
```

```
> head(querpa)
   tag   code dbh east north
1    2 QUERPA  31  1.9   6.3
3    4 QUERPA  20  0.5   6.8
5    6 QUERPA  74  4.7  19.2
21  23 QUERPA  37 16.8  17.9
54  57 QUERPA  34  6.2   0.3
63  67 QUERPA  24 18.8  22.1
```

Make a histogram of dbh for that species

```
hist(querpa$dbh,xlab="dbh (mm)",
ylab="number of
stems",main="QUERPA",col="blue")
```



To save this as a png, embed the hist call in a png device:

```
png(filename="querag_dbh.png")
hist(querpa$dbh); dev.off()
```

Now, how do you do that for all the species, individually?
Three simple steps.

1. Create a folder, and make it your working directory

2. Get a list of all the species.

```
species<-unique(ferp$code); species
 [1] QUERPA PSEUME SEQUSE LITHDE CORYCO …
```

3. Make a loop that creates a png for each one.

#OjO make sure you set your working directory to where you want the pngs to go

```
for (i in 1:length(species))
{
temp<-ferp[which(ferp$code==species[i]),] #subset of species i
png(filename=paste(species[i],"dbh.png",sep="_")) #open png
hist(temp$dbh,xlab="dbh (mm)",ylab="number of
stems",main=paste(species[i],"n=",dim(temp)[1]),col="blue")
dev.off() #close png
}
```

This loop (1) makes a temp data frame including only species i, (2) opens a png with filename created by pasting the contents of species[i] with dbh.png, separated by a _, (3) creates the histogram, including a title with the species[i] name and the number of stems, and (4) closes the png. Rinse and repeat.

Using loops for simulations

Often you can calculate the probability of something happening analytically from information you have in your data, but sometimes the analytical solution is impractical or impossible, and good old brute-force Monte Carlo simulations, based on repeated random sampling, save the day. Here is a simple example for how to go about it. Two notes: First, I'm not sure that I'd actually take this approach for this question, but it is a convenient example. Second, there are a whole range of Monte Carlo Packages for R, so you don't necessarily have to build it yourself. The goal here is to show you how easy it is to build a random resampling algorithm in R.

Goal: How many species in a random sample of 20 stems from the FERP, and how does this change depending on the size class of stems you sample?

```
#read in the data from the ferp, then trim to the variables needed here
ferp<-read.csv("http://people.ucsc.edu/~ggilbert/Rclass_docs/FERP07data.csv")
ferp<-ferp[,c(1,2,4,5,6)]

#make a custom function to categorize stems as small, medium, or large
#based on dbh cutoffs, using conditional ifelse and if statements
makeSize<-function(diam)
{s<-ifelse(diam>=10 & diam<100,s<-"small",s<-"medium")
if(diam>=300) s<-"large"
s}

#apply the makeSize function to the FERP data
#Option 1- use a for loop to apply this function for each stem one at a time
for (i in 1:dim(ferp)[1])
{ferp$size[i]<-makeSize(ferp$dbh[i])}  #system.time() takes 3.706 seconds

#Option 2- use sapply to apply this function to each stem as a vector
ferp$size<-sapply(ferp$dbh,makeSize)   #Takes 0.09 seconds

head(ferp)  #take a look at the categories

#create a blank dataframe, then use this to write
# a blank csv file with column headings to append to
# you always want to write the outputs of simulations to a file
# so you can interrupt it if it takes too long, and not lose it
blank<-
data.frame(run=numeric(0),all=numeric(0),small=numeric(0),medium=numeric(0),l
arge=numeric(0))
blank  #take a look at the blank data frame
write.table(blank,"mcoutfile.csv",append=FALSE,col.names=TRUE,row.names=FALSE
,sep=",")  #write the blank data frame to disk, to get it started


#continued on next page
```

```
#create a loop that takes a random sample of 20 stems, (either from
#all the stems, or only from small, medium, or large stems)
#unique extracts the uniqe codes in that lis
#and then length counts them to
#see how many species were included in the sample.
#Then append those four records of the number of species
#to the csv file.  Repeat this for 1000 runs.

for (i in 1:50)
{
all<-length(unique(sample(ferp$code,20))) #sample 20 from all stems
small<-length(unique(sample(ferp$code[ferp$size=="small"],20)))
medium<-length(unique(sample(ferp$code[ferp$size=="medium"],20)))
large<-length(unique(sample(ferp$code[ferp$size=="large"],20)))
out<-cbind(i,all,small,medium,large)
write.table(out,"mcoutfile.csv",append=TRUE,col.names=FALSE,
row.names=F,sep=",")
      }

#Now read the table back in
mc<-read.csv("mcoutfile.csv")
head(mc) #take a look

#look at the median and 95% CI for each of the size classes
sout<-sapply(mc,quantile,probs=c(0.025,.5,0.975))
sout[,2:5]
```

```
bp<- barplot(sout[2,2:5], ylab="Mean species in 20 stems",xlab="Size class",
ylim=c(0,12))  #draw 95% CI on bar plot of median values
arrows(bp, sout[1,2:5], bp, sout[3,2:5], lwd=1.5, angle=90, length=0.1, code=3)
box()
```