7. Control Structures

Z620: Quantitative Biodiversity, Indiana University

OVERVIEW

In Dante's satire of hell each layer contains a new iterative task for the tortured to endure for eternity. Much of the power of computing stems from the efficiency of computers in performing small tasks in an iterative manner. This lesson introduces the concepts of control structures and random numbers. We will use these tools to understand the diversity of jellybeans flavors, but these basic tools are useful for a numbers of tasks in computing and will help you automate many of your future work flows.

1) INTRODUCTION TO CONTROL STRUCTURES

Control structures allow you to control the flow of execution of a script. Today, we will focus on the "for loop", but other common control structures include: "if, else" "while" and "next" Have a look at `?Control` for documentation.

An another quick tutorial of common control structures in R can be found at https://ramnathv.github.io/pycon2014-r/learn/controls.html

```
rm(list = ls()) # clear working directory
```

A. 'If' and 'Else' Logical Tests

Short intro: when and why to use

1. The general syntax for an "if, else" control stucture in the R language. if (condition) {
   # do something
   } else {
   # do something else
   }
2. An example of the an "if, else" control stucture.

```
x <- 1:15
if (sample(x, 1) <= 10) {
   print(paste("x is",x ,"and is less than 10"))
} else {
   print(paste("x is",x,"and is greater than 10"))
}
```

3. "if" accepts logical values; can accept numbers but not strings.
4. "if" can be complemented by an "else"
5. ifelse as shorthand
6. if-else can be strung to create hirearchy of tests

B. For Loops

Let's introduce the "for loop" as a control structure.

Syntax The specific syntax of for loops will look different in other language but basics syntax remains the same.
1. Start a loop with for

2. Declare variable and number of iteration
3. Tasks to perform in code block

```
                  1.     2.          3.
```

In R this looks like: for(i in 1:10){ print(i) }

```
for(i in 1:10){
  print(i)
}
```

In this sturcture i is a variable that will hold each interger from 1 to 10 during each successive pass through the loop.

You can use this structure to move through each item of vector and perfrom any number of tasks to each element.

Like printing out a list of fruits.

```
x <- c("apples", "oranges", "bananas", "strawberries")

for (i in seq(x)) {
    print(x[i])
}
```

Note that a for loop can iterate over any type of vector: logical, numerical, string. You can also iterate over vectors using indices (R counts from 1 and not form 0 like some other languages) or assign i as a local variable that holds the value of each item in a vector.

```
for (i in x) {
    print(i)
}
```

You can use loops to transform vectors. The two scripts below give use the length of the strings in our vector. However, the first one prints to the screen a local scalar variable, `len`, and the second uses the loop to fill a global scalar vector, `len`, that now holds the length of each strings.

```
for (i in seq(x)) {
  len <- nchar(x[i])
  print(len)
}

len <- ""
for (i in seq(x)) {
  len[i] <- nchar(x[i])
}
print(len)
```

Note on the use of for loops
When possible try to avoid them. They take a long time to execute and are somewhat clumsy. Consider using apply type functions.

The sapply function can be used to get the same results as the loop above. Check out `?apply` and `?lapply`.

```
len <- sapply(x, nchar)
#the output of sapply is a "named int", but this can by easily coerced to a vector.
len <- as.vector(len)
print(len)
```

C. Combining for and if

e.g. print only numbers that can be divided by 3

```r
for(i in 1:10){
  if(i %% 3 == 0){
    print(i)
  }
}
```

D. While loops

Another loop control structure is the “while loop”.

An example of the while loop.

```r
i <- 1
while (i < 10) {
    print(i)
    i <- i + 1
}
```

Notice that a while loop needs an specific break point.

What would happen if the code block was changed to subtract 1 from i in each iteration of the loop?

2) Revisiting the jelly-bean community with control structure

The control structures discussed above have unlimited applications in the relm of quatitative methods. In this section you will learn how to apply these structures in conducting the analyses scheme you carried out for the jelly bean community. But before we can do that we need to get familiar with another powerful family of functions in R: built-in probability ditributions.

A. Statistical distributions in R

Many groups of quantities can be described as frequency distributions. You are probably familiar with the Normal ditribution and its hallmark bell curve. R has built-in functions describing many statistical distributions. You can find them listed in the R Documentation by typing `?Distributions` in the console.

Choose your favorite distribution and look at its specific documentation. Each disribution in R has 4 functions asociated with it: **d**ensity, **p**robability, **q**uantiles and **r**andom sample. For example the uniform distribution (an equal probaility for all values) is called *unif* in R and has the following 4 functions.

```r
dunif(x, min = 0, max = 1, log = FALSE)                        # Density
punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)   # Probaility
qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)   # Qunatile
runif(n, min = 0, max = 1)                                     # Random Sample
```

A quick note on random sampling by computers
Computers can’t generate truly random number, but rather pseudorandom numbers. This is achieved by a function that generates sequences of numbers that behave like random numbers, but are completely determined by the functions input known as a *seed*. Put in the same seed twice and you will get the same sequence of numbers.

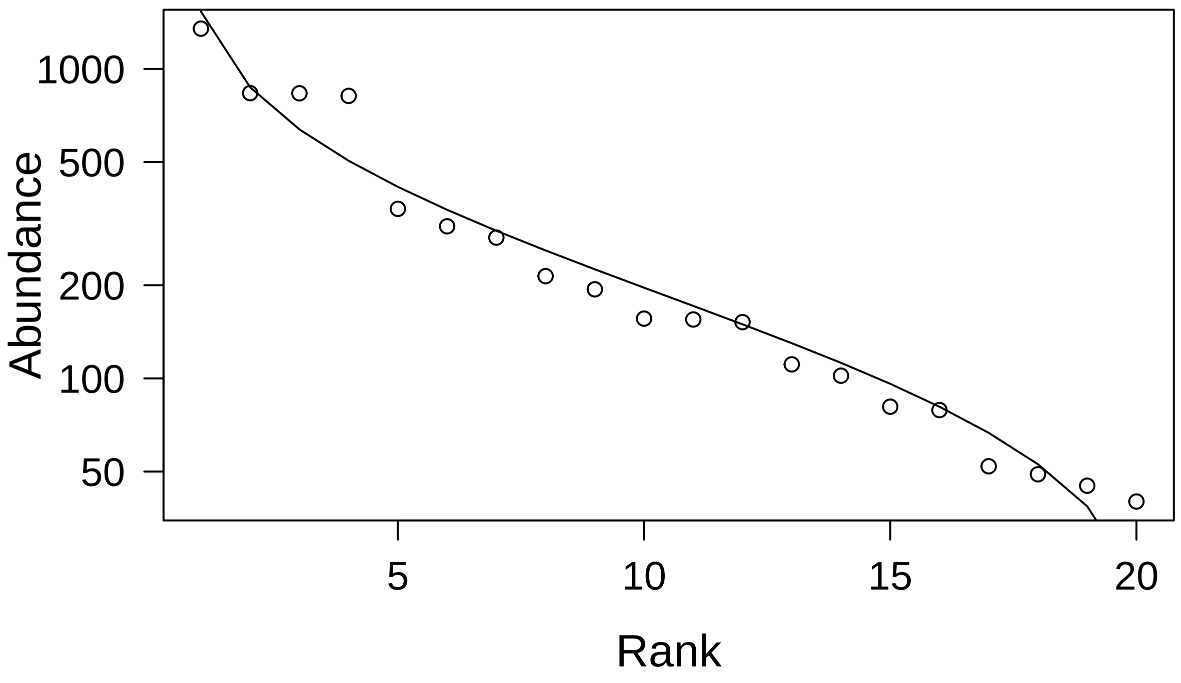
B. generating the source community

In the jelly bean excercise you had a jar of jelly beans simulating a biological community. To generate our digital community we will recall that the species abundances in biological communities often follows the log-normal distribution. To simulate a community we will draw a sequence of random numbers from the log-normal distribution. Each of these numbers will represent the abundance of a single species in our simulated community.

```
# for reproducible results we will set the seed
set.seed(6)
# draw 20 random numbers from lOg-normal distribution with a mean of 5 (log scale)
jelly.jar <- rlnorm (n = 20, meanlog = 5 )
# force numers to integers
jelly.jar <- as.integer (jelly.jar)
```

Let's plot the rank-abundance curve of our community using the `rad` function from the `vegan` package

```
require (vegan)
RACresults <- rad.lognormal(jelly.jar)
plot(RACresults, las = 1, cex.lab = 1.4, cex.axis = 1.25)
```



B. Sampling the Source Community

Next we will write a function that will draw a random sample from our `jelly.jar` community. Take a moment to break down the process sampling the jelly bean jar into a sequence of actions.
One such sequence may be:
1. draw random jelly bean
2. classify jelly bean color
3. add 1 to list of colors by color sampled
4. repeat steps 1-3 N times, where N is your sample size

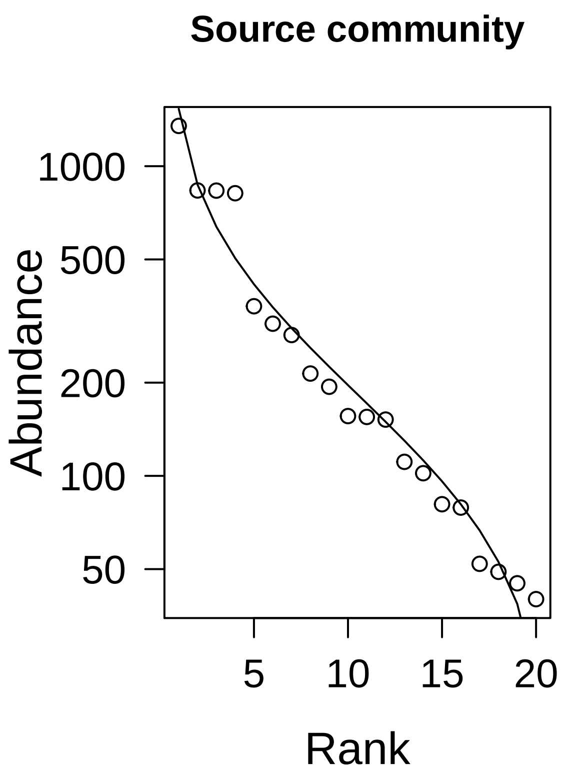
This sequence of actions is an algorithm. Here is how looks in R code:

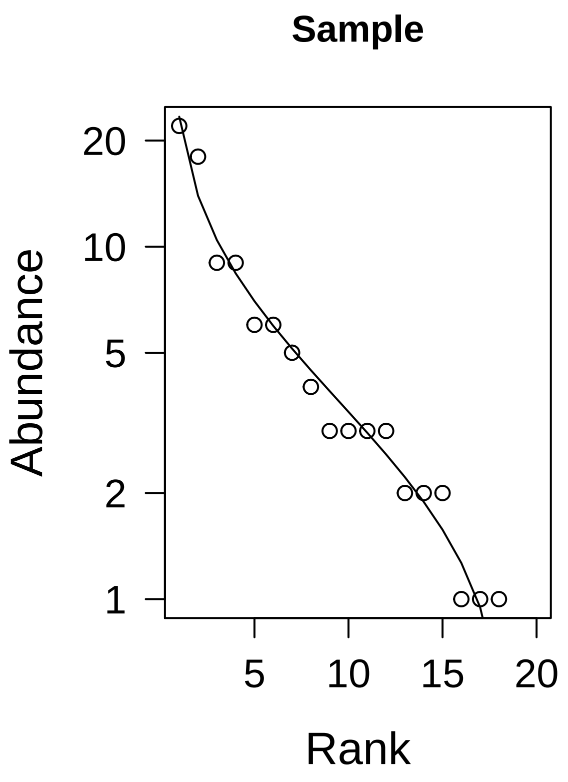
```
sample.community <- function (x,n){
  # write out an explicit vector of the community
  all.individuals <- rep(seq(x), x)
  # take n samples from the community
  survey <- sample(all.individuals,n)
  # prepare vector of species with 0 in each bin
  survey.sum <- rep(0, length(x))
  # add 1 to species bin each time an individual of that species was sampled
    for ( i in survey){
     survey.sum[i] <- survey.sum[i] + 1
    }
  return(survey.sum)
}
```

Note that in this implementation of the algorithm above we drew multiple samples using the `sample` function. This function iterated over the samlping step `n` times. Next we wrote a for loop to iterate over those samples and bin them into our species list.

Let's compare the rank abundance curves of a sample to our source community.

```
# for reproducible results we will set the seed
set.seed(3)
# sample using our fuction
sample.jelly <- sample.community(x=jelly.jar,n=100)
# generate RAC using vegan
RACsample <- rad.lognormal(sample.jelly)
plot.new()
par(mfrow=c(1,2))
plot(RACresults, las = 1, cex.lab = 1.4, cex.axis = 1.25, xlim=c(1,20), main="Source community")
plot(RACsample, las = 1, cex.lab = 1.4, cex.axis = 1.25, xlim=c(1,20), main = "Sample")
```





How similar or different is the sample from the source community?

Testing input to a function using the `if` statement
Functions work very well if they are fed the right kind of input. But pass some inapropriate argument and things will go wrong, and posssibly horribly wrong. To demostrate this idea try and sample the `jelly.jar` with a sample size larger than the size of the community.

```
sample.community(x=jelly.jar,n=(sum(jelly.jar)+1))
```

R produces an error message:

> Error in sample.int(length(x), size, replace, prob) : cannot take a sample larger than the population when 'replace = FALSE'

The first line of this message indicates the function that produced it. In this case it is the `sample.int()` function. This function is nested withhin the `sample()` function (you can see the code for `sample` by typing *sample* without brackets in the console). The second line of the error message tells us that you cannot take more sample the the population size unless you set the `replace` argument to TRUE. This message is very helpful.

To prevent such errors, that could lead to unexpected behavior of your code, it is a good idea to test the input of functions before processing it in the function. Let's rewrite our sampling function to include a test for sample size using the `if` statement. We will also use the `stop` finction to produce our own error message.

```
sample.community <- function (x,n){
  # sample size cannot exceed community size
  if (n > sum(x)) stop ("Sample size cannot exceed total size of community")
  # write out an explicit vector of the community
  all.individuals <- rep(seq(x), x)
  # take n samples from the community
  survey <- sample(all.individuals,n)
  # prepare vector of species with 0 in each bin
  survey.sum <- rep(0, length(x))
  # add 1 to species bin each time an individual of that species was sampled
    for ( i in survey){
     survey.sum[i] <- survey.sum[i] + 1
    }
  return(survey.sum)
}
```

Now if we pass to our function `n` that exceeds the community size we will get an error from our function.

```
sample.community(x=jelly.jar,n=(sum(jelly.jar)+1))
```

> Error in sample.community(x = jelly.jar, n = (sum(jelly.jar) + 1)) : 'Sample size cannot exceed total size of community

Of course, there are many ways by which a functions arfuments mcan be wrong, to rephrase Tolstoy *"All happy functions are alike; each unhappy function is unhappy in its own way"*. We demonstrated a logical error. Another common error is passing data of the wrong type to a function, such as a string when a number is called for. These errors can be caught using the *is.type* functions, where *type* is relaced by the type of interest (e.g.`is.numeric`). While We cannot prevent all errors, when writing a function it is worth taking a moment to think about how to prevent some common errors.

C. Estimating Richness in Repeated Samples

We will next use for loops to explore how well richness indices succeed in estimating the true richness of the source community. We will use The function `estimateR` from the `vegan` package to calculate diversity indices.

```
S <- estimateR(sample.jelly)
S
```

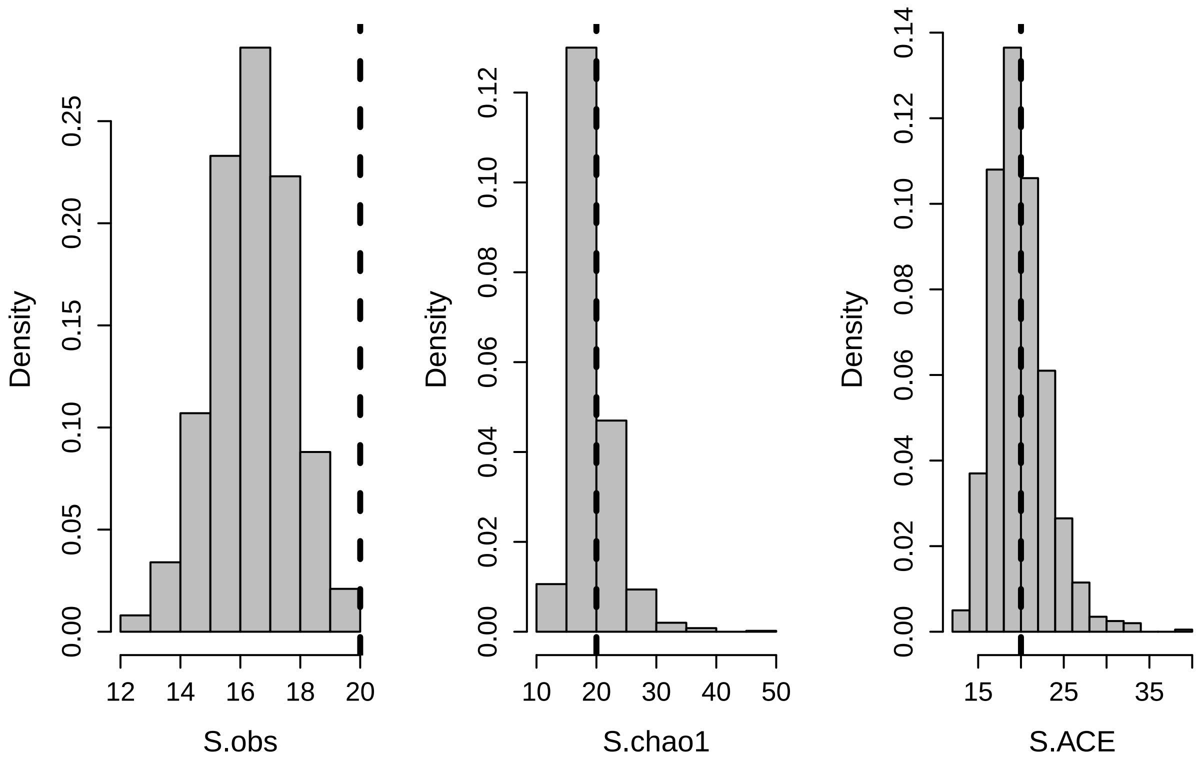
```
##     S.obs   S.chao1  se.chao1     S.ACE    se.ACE
## 18.000000 18.750000  1.417910 19.680642  1.722197
```

How well do these 3 indices estimate the true diversity? Using our simulated community we can test that! We will draw repeated samples from our source community and estimate S for each of them. Lastly we will look at the distribution of the results compared to the true S.

```
# Number of individuales per sample
n <- 100
# Number of repeated samples
N <- 1000
# initialize matrix to store results
repeated.S <- matrix(NA, nrow = N, ncol = length(S))
colnames(repeated.S) <- names(S)
# resample and store results
for (i in seq(N)){
  sample.jelly <- sample.community(x=jelly.jar,n=n)
  repeated.S[i,] <- as.vector(estimateR(sample.jelly))
}

#plot distribution of results as histogram
plot.new()
par(mfrow=c(1,3))
```

```r
for (i in c("S.obs", "S.chao1", "S.ACE")){
  hist(repeated.S[,i],prob=TRUE,  cex.lab = 1.4, cex.axis = 1.25, xlab=i, main = NULL, col="grey")
  #add dashed line to mark true S
  abline(v=length(jelly.jar), lty=2, lwd=3)
}
```



Play around with the sample size (by changing the value of `n`). How does sample size affect observed and estimated richness?

E. Collector's Curve

S.observed as a function of N. First take a single sample of size N from the source population. Subsample you sample with sample sizes increasing from 1 to N and note the observed richness for each subsample.

```r
#first take a single sample of size N from the source population
N<-1000
sample.dj <- sample.community(jelly.jar,n = N)

#generate vector to store results
S.collectors <- rep(NA, N)
#loop through 1:N and check S.obs
for (i in 1:N){
  current <-  sample.community(sample.dj,n = i)
  S.collectors[i] <- S.obs(t(current))
}

plot(1:N, S.collectors, xlab = "Sample size",ylab = "S.obs")
```

What is the samllest sampe size to indicate the true richness?
What is the largest sample size to underestimate richness?
What sample size would you recommend for the next survey of ths community?