Chap I: Introduction to R: More on the syntax

Control Structures

- Control structures are useful to implement repetitive tasks.
- R has control structures similar to C.

For loops

Loops are used to carry out a sequence of related operations without having to write the code for each step explicitly. For instance, suppose we want to calculate

$$\sum_{i=1}^{10} i.$$

```
x = 0
for (i in 1:10)
{
   x = x + i
}
x
```

For loops

- In the above program, x is an accumulator variable, meaning that its value is repeatedly updated while the program runs.
- Always remember to initialize accumulator variables (to zero in this example).

To clarify, we can add a print statement inside the loop body.

```
x = 0
for (i in 1:10)
{
   x = x + i
   print(c(i,x))
}
```

For loops

The general structure of 'for' loops:

```
for (var in seq) expr
```

or

```
for (var in seq){
     expr
}
```

For loops

Activity: Given the matrix A

$$A = \begin{pmatrix} 1 & 0 & 6 & 2 \\ 8 & 0 & -2 & -2 \\ 2 & 9 & 1 & 3 \\ 2 & 1 & -3 & 10 \end{pmatrix},$$

write a for loop that calculates the sum of each row of A.

For loops

- This exercise is good practice, but in R, it is often possible to avoid loops.
- The language provides several ways to avoid loops.
- For instance, to sum the row/columns of a matrix, use rowSums/colSums.

  ```
  rowSums(A)
  colSums(A)
  ```

- Another common way in R to avoid for loops is via the apply functions.

Apply

Sometimes we want to apply a function to each individual row or column of a matrix, or to each element of a list. Instead of wrting a *for loop*, R provides a very convenient class of functions that can result in shorter code: apply (for matrices and arrays) and 'lapply', 'sapply' for lists.

Apply

```
args(apply)
function (X, MARGIN, FUN, ...)
X: the object;
'MARGIN': a vector giving the subscripts which
        the function will be applied over.
        1 indicates rows, 2 indicates columns.
'FUN': the function to be applied. In the case
          of functions like +, %*%, etc., the function
          name must  be backquoted or quoted.
'...': additional optional arguments to FUN.
```

Apply

Here is an example.

```
x <- rnorm(100, -5, 1)
y <- rnorm(100, 5, 1)
X <- cbind(x, y) #another useful way of creating matrices

apply(X, MARGIN=2, FUN=mean)
apply(X, MARGIN=2, FUN=var)
#instead of
for( i in 1:2) print(mean(X[,i]))
```

For loops

Practice: We have seen 3 ways to compute the row sums of a matrix. Write code to compare them. The function system.time can be used to time execution time in R.

```
n=5000; p = 5000;
r_sum=numeric(n)
A=matrix(rnorm(n*p),ncol=p)
system.time(for (i in 1:n) r_sum[i]=sum(A[i,]))
system.time(apply(A,MARGIN=1,FUN=sum))
system.time(rowSums(A))
```

Conclusion: rowSums is fastest, then for loop, then apply.

Apply

The function 'lapply' and 'sapply' work like 'apply' and apply a specified function 'FUN' to each element of a vector or a list. The difference between the two: 'sapply' returns a vector when possible. Again, both allow passing of additional arguments to FUN through the '...' argument.

```
ls <- list(x1 = rnorm(10), x2 = rnorm(1000))
lapply(ls, mean) # result is a list
sapply(ls, mean) #result is a vector
```

Example

Exercise: We have ten localities. In the month of April, the daily amount of rain (in mm) in locality $1 \leq i \leq 10$ is a random variable that has a log-normal distribution $\mathrm{LN}(\mu_i, \sigma_i^2)$, with $\mu_i = \sigma_i = i$. Use simulation to approximate the average amount of rain in April in these localities.

While loop

Now consider the following example:

Example: suppose we wish to calculate the value of the partial sums of the harmonic series $\sum_{j=1}^{n} 1/j$ the first time it exceeds 5. It is a fact that the harmonic series diverges: $\sum_{j=1}^{\infty} 1/j = \infty$, so eventually the partial sum must exceed any given constant.

Here we need a while loop, because the number of time to loop is not known in advance.

```
while (cond){
     expr
}
```

While loop

```
n = 1 ## Don't forget
x = 0 ## To initialize
while (x <= 5)
{
   x = x + 1/n ## The order of these two statements
   n = n+1 ## in the block is important.
}
```

Deduce the smallest value of n for which $\sum_{j=1}^{n} 1/j > 5$.

If-else

An if block can be used to make on-the-fly decisions about what statements of a program get executed. For example,

```
y = 7
if (y < 10)
{
        x = 2
} else
{
       x = 1
}
```

If-else

General syntax:

```
if (cond) expr1 else expr2
```

Or

```
if (cond) {
    expr1
} else {
    expr2
}
```

If-else

Another example: the following program places the sum of the odd integers up to 100 in A and the sum of the even integers up to 100 in B.

```
A = 0;B = 0
for (k in 1:100)
{
   if (k %% 2 == 1) {
         A = A + k
   } else {
         B = B + k
   }
}
#x %% y is "x mod y" and x%/% y is the integer
#division of x by y.
```

next and *break*

A break statement is used to exit a loop when a certain condition is met. A next statement results in the current iteration being aborted, but the loop continues with the next iteration.

next and *break*

```
x = 0
for (k in seq(100)){
    if (k %% 2 == 0) { next }
    ## Skip even numbers, but keep looping.
    if (x >= 50) { break }
    ## Quit looping when the sum exceeds 50.
    x = x + k
}
```

next and *break*

The next statement is often not necessary. We could also do

```
x = 0
for (k in seq(100)){
    if (k %% 2 == 1) {
        x=x+k
        if (x >= 50) { break }
    ## Quit looping when the sum exceeds 50.
    }
}
```

Writing your own functions

- After solving the above example, it would be interesting to give a simple name to these lines of code so that they get executed every times the name is called.
- This is the idea of function in computer programming.
- There is a very simple way of doing this in R.

Writing your own functions

- You can define you own functions with the function construct.
- The body of the function (the code in the braces) is executed whenever the function is called.
- The return statement produces a value that is returned when the function is called.

Writing your own functions

```
f = function(x, y) {
    return(x / (1+y^2))
}
##This is same as

f = function(x, y) {
    x / (1+y^2)
}

## a will be 0.2.
a = f(1, 2)
```

Writing your own functions

Example: Write a R function that returns X_1 defined as follows.

- Generate $U_1 \sim \mathcal{U}(-1,1)$ and $U_2 \sim \mathcal{U}(-1,1)$ independently until $S = U_1^2 + U_2^2 \leq 1$.
- Set $Z = \sqrt{-2\log(S)/S}$ and give back $X_1 = ZU_1$.

Writing your own functions

```
NormGen=function(){
   S=2
   while(S>1){
      U1=runif(n=1,min=-1,max=1);U2=runif(1,-1,1)
      S=U1^2+U2^2
   }
   Z=sqrt(-2*log(S)/S)
   X=Z*U1
   return(X)
}
```

Writing your own functions

A lottery consists in picking randomly 3 distinct numbers from the set $\{1, \ldots, 20\}$. There is a public event at which three distinct "winning" numbers are drawn from the same set. Whoever has the winning numbers in the correct order wins. Write a function that returns how many times someone will play this lottery before winning.

Writing your own functions

In writing your own function, you can use default values for the arguments.

```
f = function(x, y=1) {
    return(x / (1+y^2))
}

#The following 3 calls are equivalent
f(2)
f(2,1)
f(x=2,y=1) #preferred syntax
```

Writing your own functions

- Outside code cannot "see" variables defined inside a function.
- But variables that exist outside the function can be seen from inside.

```
f2 = function(x) {
  y=1
  return(x / (1+y^2))
}
#the variable y is only available inside the function
```

Writing your own functions

- But variables that exist outside the function can be seen from inside.

```
n=10
f3=function(x){
   sum=0
   for ( i in 1:n){
       sum=sum+ sin(2*pi*x[i]/n)
   }
   return(sum)
}
#This is bad coding, but will work.
#better code: make n into a proper argument
```

Writing your own functions

- We can pass a function to another function. Remember the apply and sapply functions.

```
X=cbind(rnorm(n=10,mean=0,sd=1),
                          runif(n=10,min=-1,max=1))
apply(X,1,var)
#here var is a function
```