Module 2

Nathan Cardoso

Control structures in R

Control structures in R allow you to control the flow of execution of teh program, depending on the runtime conditions. Some common structures are if-else. for, while, repeat, next, break and return.

The if control structure

The general syntax is as follows. The else section is optional. Suppose you want to execute something based on a certain condition but do nothing if the condition is not true; it is possible to use just an if.

```
if (<condition>) {
  ## do something
} else {
  ## do something else
}
```

We can also have multiple if conditiions as follows

```
if (<condition>) {
  ## do something
} else if (<condition2>){
  ## do something diff
} else {
  ## do something diff
}
```

This is a simple example of an if-else.

```r
x <- 5
if(x>3) {
  y <- 10
} else {
  y <- 0
}
y
```

```
## [1] 10
```

Another way of doing the same is:

```
x <- 7
y <- if(x>3) {
  10
} else {
  0
}
y
```

```
## [1] 10
```

The for loop

For loops take an iterator variable and assign it successive values from a sequence or vector, For loops are most commonly used for iterating over elements of an object (list, vector, etc).

```
for(i in 1:10){
  print(2*i)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

This loop takes the i variable and in each iteration of the loop assigns it a value starting from 1. It exits when the vector 1:10 comes to an end.

There are different ways to use a for loop when we want to index the elements in various R objects. All the three loops written above have the same behavior.

```
x <- c("a","b","c","d")
for(i in 1:4) {
  print(x[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

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

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

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

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

We can also have nested for loops. A use-case could be iterating through the elements of a matrix.

```
x <- matrix(1:6, 2,3)
for(i in seq_len(nrow(x))){
  for(j in seq_len(ncol(x))){
    print(x[i,j])
  }
}
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 2
## [1] 4
## [1] 6
```

However, nesting is not advisable beyond 3-4 levels as it becomes difficult to read and understand.

The while loop

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed the condition is tested again and so forth.

```
count <- 0
while(count<10){
  print(count)
  count <- count + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

While loops can potentially result in infinite loops if not written properly. You have to make sure that the condition that stops the loop will actually occur. It is considered safer to use a for loop that has a definite limit on the number of times it will execute.

We can also test multiple conditions in a while loop

```
z <- 5
while(z>=3 && z<=10){
  print(z)
  coin <- rbinom(1, 1, 0.5)

  if(coin==1){    ## random walk
      z <- z+1
  } else {
      z <- z-1
    }
}
```

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 9
## [1] 10
```

Conditions are always evaluated from left to right.

repeat, next and break

Repeat initiates an infinite loop; these are not commonly used in statistical applicaitons but they do have their uses. The only way to exit a repeat loop is to call break.

```
x0 <- 1
tol <- 1e-8

repeat {
    x1 <- computeEstimate()
    if(abs(x1-x0) < tol){
      break
    } else {
      x0 <- x1
  }
}
```

The loop above is a bit dangerous because there is no guarantee it will stop. It depends entirely on the result of the computeEstimate() function. It is better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence between x0 and x1 was achieved or not.

next is used to skip an iteration of a loop

```
for(i in 1:100){
  if(i<=10){
    ## Skip the first 20 iterations
    next
  }
  ## Do something here
}
```

The return signals that a function should exit and return a given value. It is sometimes used with loops as well.

Functions

Functions are created by using the function( ) directive and are stored as R objects like anything else. In particular, they are R objects of the class "function".

```
f <- function(<arguments){
      ##Do something interesting
}
```

Some important points

- Functions can be passed as arguments to other functions.
- Functions can be nested, so that you can define a function inside of another function.
- The return value of a function is the last expression in the function body to be evaluated

Lets take an example of a simple additive function.

```
add2 <- function(x,y){
          x+y
}
```

In this function, we did not explicitly use return because R will automatically return the last expression. This function had only one expression which is the first / last.

```
add2(4, 7)
```

```
## [1] 11
```

```
add2(-1, 3)
```

```
## [1] 2
```

Arguments in functions

Functions have named arguments which potentially have default values

- The formal arguments are the arguments included in the function definition.

- The formals() function returns a list of all the formal arguments of the function.
- Not every function call in R makes use of all the formal arguments.
- Function arguments can be missing or have default values.

A slightly more complicated function, that will subset a vector of numbers and return a subset of those numbers that are greater than 10. Here, a vector is the argument.

```
# A vector for illustrations
myvec <- (c(3,5,11,6,13,19,8))

above10 <- function(x){
  vector1 <- x > 10
  x[vector1]
}
above10(myvec)
```

```
## [1] 11 13 19
```

This can be generalized into a more useful function, with no value hardcoded in it. Now this function has two named arguments; a vector and the integer.

```
above <- function(x,n){
  vector1 <- x > n
  x[vector1]
}
above(myvec, 7)
```

```
## [1] 11 13 19  8
```

```
above(myvec, 15)
```

```
## [1] 19
```

The function can be improved by specifying a default value for n to avoid errors if a user forgets to enter the value

```
above <- function(x,n=10){
  vector1 <- x > n
  x[vector1]
}
above(myvec, 7)
```

```
## [1] 11 13 19  8
```

```
above(myvec, 15)
```

```
## [1] 19
```

```r
above(myvec)
```

```
## [1] 11 13 19
```

Argument matching

R functions arguments can be matched positionally or by name. So the following calls to sd are all equivalent.

```r
mydata <- rnorm(100)
sd(mydata)
```

```
## [1] 0.8899533
```

```r
sd(x = mydata)
```

```
## [1] 0.8899533
```

```r
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 0.8899533
```

```r
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 0.8899533
```

Even though it is permitted, mixing up the positions of function arguments is not recommended as it becomes more confusing to the person reading or reviewing the code.

You can mix positional matching with matching by name .When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition

```r
args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

The following two function calls are equivalent.

```
lm(data = mydata, y-x, model=FALSE, 1:100)
lm(y-x, mydata, 1:100, model=FALSE)
```

Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults except for an argument near the end of the list. Named arguments can also help if you remember the name of the argument and not its position in ht e argument list.

Function arguments can be partially matched, which is useful for interactive work. The order of operations when given an argument is: Check for exact match of a named argument → Check for a partial match → Check for a posiitonal match.

Some practice on writing and improving functions

A function to compute the mean of each column in a dataframe.

```
colmean <- function(x){
    cols <- ncol(x)
    means <- numeric(cols)
    for(i in 1:cols){
      means[i] <- mean(x[, i])
    }
    means
}
```

Using the airquality dataset to test the function.

```
colmean(airquality)
```

```
## [1]        NA        NA  9.957516 77.882353  6.993464 15.803922
```

We do not get valid numbers for the firsst two columns. If any of the entries in a row or column are NA, then the result of mathematical operations like sum, mean, etc on those rows or columns will also return NA. A simple solution to this problem is to first eliminate the rows or entries having NA values in the dataset. We can add a boolean argument to the function; if TRUE the NAs will be removed from the dataset. The default value is set to TRUE as this is a common operation.

```
colmean <- function(x, removeNA=TRUE){
    cols <- ncol(x)
    means <- numeric(cols)
    for(i in 1:cols){
      means[i] <- mean(x[, i], na.rm = removeNA)
    }
    means
}
```

Now, when we call this function, the result is

```
colmean(airquality, TRUE)
```

```
## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```