R Tutorial

Operators in R

R's binary and logical operators will look very familiar to programmers. Note that binary operators work on vectors and matrices as well as scalars.

Arithmetic Operators include:

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation

Logical Operators include:

Operator	Description
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to

Data Types

R has a wide variety of data types including scalars, vectors (numerical, character, logical), matrices, data frames, and lists.

Creating New Variables

Use the assignment operator **<-** to create new variables.

```
> a <- 10
> b <- 20
> c <- a + b
```

Vectors

```
a <- c(1,2,5.3,6,-2,4) # numeric vector
b <- c("one","two","three") # character vector
c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE) #logical vector
```

Refer to elements of a vector using subscripts.

```
a[c(2,4)] # 2nd and 4th elements of vector
```

Matrices

All columns in a matrix must have the same mode(numeric, character, etc.) and the same length. The general format is

mymatrix <- **matrix(***vector*, **nrow=***r*, **ncol=***c*, **byrow=***FALSE*,
dimnames=list(*char_vector_rownames*, *char_vector_colnames***))**

byrow=TRUE indicates that the matrix should be filled by rows. **byrow=FALSE** indicates that the matrix should be filled by columns (the default). **dimnames** provides optional labels for the columns and rows.

```
# generates 5 x 4 numeric matrix
y<-matrix(1:20, nrow=5,ncol=4)

# another example
cells <- c(1,26,24,68)
rnames <- c("R1", "R2")
cnames <- c("C1", "C2")
mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=TRUE,
  dimnames=list(rnames, cnames))
```

Specific column or row extraction

```
x[,4] # 4th column of matrix
x[3,] # 3rd row of matrix
x[2:4,1:3] # rows 2,3,4 of columns 1,2,3
```

Practicing

Generating a matrix

```
matrix(1:12, 3, 4, byrow = T)

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Remove current variable from R

```
rm(list = ls())
```

Loads specified data sets, or list the available data sets.

```
data()
```

Loads ‘mtcars’

```
data('mtcars')
```

Help ‘mtcars’

```
?mtcars
```

Data Frames

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.). This is similar to SAS and SPSS datasets.

```
d <- c(1,2,3,4)
e <- c("red", "white", "red", NA)
f <- c(TRUE,TRUE,TRUE,FALSE)
mydata <- data.frame(d,e,f)
names(mydata) <- c("ID","Color","Passed") # variable names
```

There are a variety of ways to identify the elements of a data frame .

```
myframe[3:5] # columns 3,4,5 of data frame
myframe[c("ID","Age")] # columns ID and Age from data frame
myframe$X1 # variable x1 in the data frame
```

Lists

An ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name.

```
# example of a list with 4 components -
# a string, a numeric vector, a matrix, and a scaler
w <- list(name="Fred", mynumbers=a, mymatrix=y, age=5.3)

# example of a list containing two lists
v <- c(list1,list2)
```

Identify elements of a list using the [[]] convention.

```
mylist[[2]] # 2nd component of the list
mylist[["mynumbers"]] # component named mynumbers in list
```

Practicing

Show the difference between vector and list.

```
c(7, 'list', 8)

list(name = 'fred', mynumber = c(1,2,3), mymatrix = 1:10, age = 5.3)
```

Factors

Tell R that a variable is **nominal** by making it a factor. The factor stores the nominal values as a vector of integers in the range [1... k] (where k is the number of unique values in the nominal variable), and an internal vector of character strings (the original values) mapped to these integers.

```
# variable gender with 20 "male" entries and
# 30 "female" entries
gender <- c(rep("male",20), rep("female", 30))
gender <- factor(gender)
# stores gender as 20 1s and 30 2s and associates
# 1=female, 2=male internally (alphabetically)
# R now treats gender as a nominal variable
summary(gender)
```

Useful Functions

```
length(object) # number of elements or components
str(object)    # structure of an object
class(object)  # class or type of an object
names(object)  # names

c(object,object,...)       # combine objects into a vector
cbind(object, object, ...) # combine objects as columns
rbind(object, object, ...) # combine objects as rows

object     # prints the object

ls()       # list current objects
rm(object) # delete an object
```

Getting Information on a Dataset

There are a number of functions for listing the contents of an object or dataset.

```
# list objects in the working environment
ls()
```

```
# list the variables in mydata
names(mydata)
```

```
# list the structure of mydata
str(mydata)
```

```
# list levels of factor v1 in mydata
levels(mydata$v1)
```

```
# dimensions of an object
dim(object)
```

Data saving

```
write.table(mtcars, file = '~/Desktop/mtcars.txt', row.names = T, col.names = T,sep = '\t', quote = F)
```

Data loading

```
read.table(file = '~/Desktop/mtcars.txt',sep = '\t', header = T)
```

From A Comma Delimited Text File

```
# first row contains variable names, comma is separator
# assign the variable id to row names
# note the / instead of \ on mswindows systems

mydata <- read.table("c:/mydata.csv", header=TRUE,
   sep=",", row.names="id")
```

From Excel

One of the best ways to read an Excel file is to export it to a comma delimited file and import it using the method above. Alternatively you can use the **xlsx** package to access Excel files. The first row should contain variable/column names.

```
# read in the first worksheet from the workbook myexcel.xlsx
# first row contains variable names
library(xlsx)
mydata <- read.xlsx("c:/myexcel.xlsx", 1)

# read in the worksheet named mysheet
mydata <- read.xlsx("c:/myexcel.xlsx", sheetName = "mysheet")
```

R if statement

The syntax of if statement is:

```
if (test_expression) {
statement
}
```

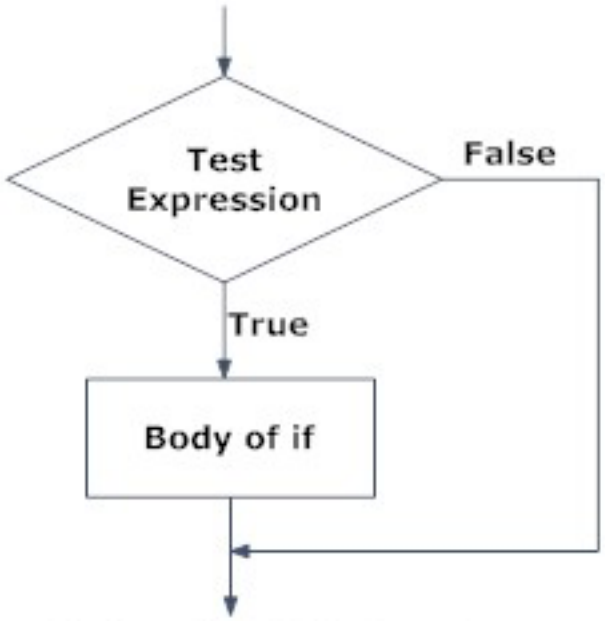

Fig: Operation of if statement

```
X <- 5
If (x > 0)
{
      print("Positive number")
}
```

if...else statement

The syntax of if...else statement is:

```
if (test_expression) {
statement1
} else {
statement2
}
```

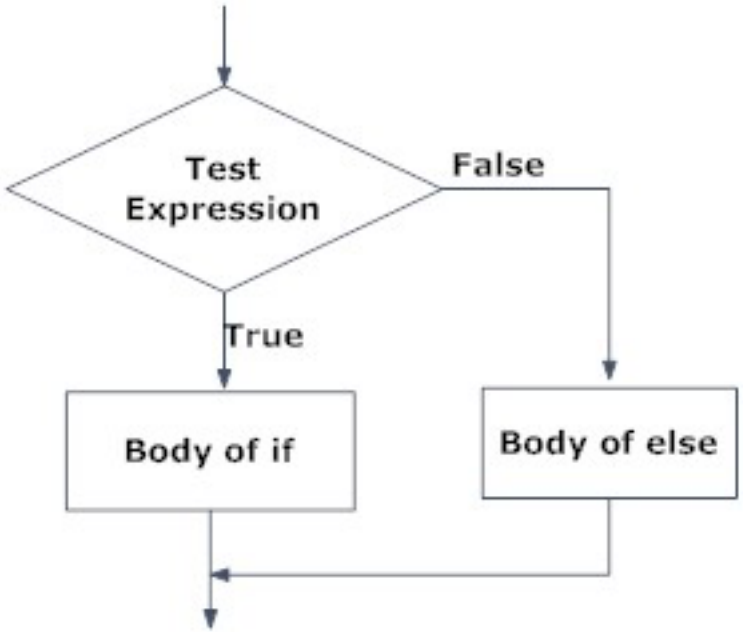

Fig: Operation of if...else statement

```
x <- -5
If  (x > 0)
{
    print("Non-negative number")
} else
{
    print("Negative number")
}
```

if...else Ladder

The if...else ladder (if...else...if) statement allows you execute a block of code among more than 2 alternatives

The syntax of if...else statement is:

```
if ( test_expression1) {
statement1
} else if ( test_expression2) {
statement2
} else if ( test_expression3) {
statement3
} else {
statement4
}
```

```
x <- 0
if (x < 0)
{
    print("Negative number")
}  else if (x > 0)
{
   print("Positive number")
}   else
print("Zero")
```

Syntax of for loop

```
for (val in sequence)
{
statement
}
```

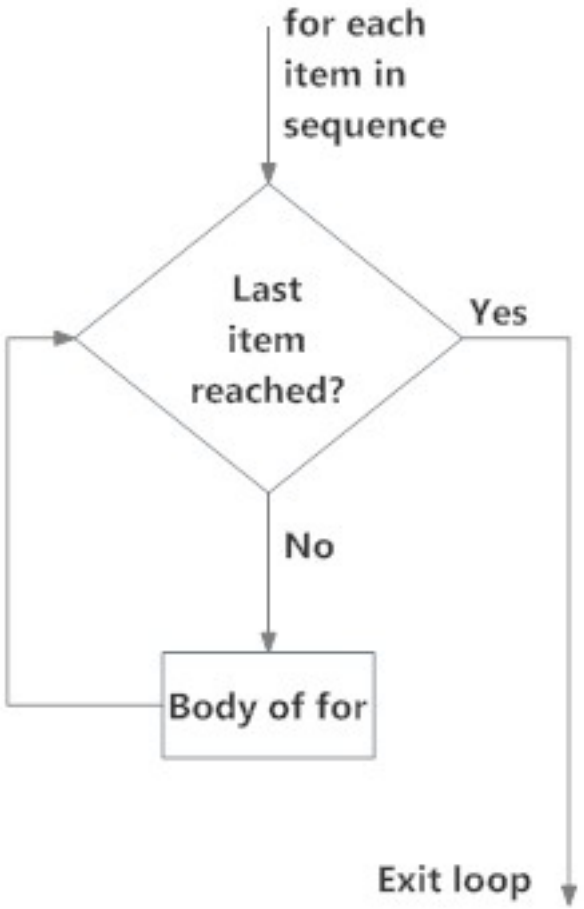

Fig: operation of for loop

```
x <- c(2,5,3,9,8,11,6)
count <- 0
for (val in x)
{
      if(val %% 2 == 0)
            count = count+1
}
print(count)
```

Syntax for Writing Functions in R

```
func_name <- function (argument) {
statement
}
```

Here, we can see that the reserved word `function` is used to declare a function in R.

The statements within the curly braces form the body of the function. These braces are optional if the body contains only a single expression.

Finally, this function object is given a name by assigning it to a variable, `func_name`.

Example of a Function

```
pow <- function(x, y) {
# function to print x raised to the power y
result <- x^y
print(paste(x,"raised to the power", y, "is", result))
}
```

Here, we created a function called `pow()`.

It takes two arguments, finds the first argument raised to the power of second argument and prints the result in appropriate format.

We have used a built-in function `paste()` which is used to concatenate strings.

We can call the above function as follows.

```
>pow(8, 2)
[1] "8 raised to the power 2 is 64"
> pow(2, 8)
[1] "2 raised to the power 8 is 256"
```

Apply

How to calculate the mean(median) value for each column(row) of matrix

```
medianMatrix <- function(x, id = 'column')
{
    if (id != 'column')
    {
      for (i in 1:nrow(x))
      {
           med <- median(x[i,])
           print (med)
      }
    } else
    {
      for (i in 1:ncol(x))
      {
        med <- median(x[,i])
        print (med)
      }
    }
}
```

```
medianMatrix(matrix(1:30, 5, 6),id = 'row')
```

Apply

```
apply (matrix(1:30, 5, 6),  1, median)

apply (matrix(1:30, 5, 6),  2, median)
```

Practicing

Extract the element in each column with value bigger than 10

```
apply (matrix(1:30, 5, 6),  2, function(x) {x[which(x > 10)]})
```

Sapply

```
s <- list(1:10,
          runif(10,0,1),
          c(10,8,1,2,100))

sapply(s, median)
```

Sapply()

```
#install ‘gapminder’
library(gapminder)

head(gapminder)

# we want the maximum life expectancy for each
#continent


sapply(unique(gapminder$continent),  function(x)
{max(gapminder$lifeExp[which(gapminder$continent ==
x)])})
```

ddply()

```
install.packages("plyr", dependencies =
TRUE)

library(plyr)

ddply(gapminder, ~(continent), function(x)
{median(x$lifeExp)})
```

Descriptive Statistics

mean, sd, var, min, max, median, range, and quantile

How to calculate FPKM, or TPM????

Chi-Square Test

For 2-way tables you can use **chisq.test(***mytable***)** to test independence of the row and column variable. By default, the p-value is calculated from the asymptotic chi-squared distribution of the test statistic. Optionally, the p-value can be derived via Monte Carlo simultation.

Fisher Exact Test

fisher.test(*x***)** provides an exact test of independence. *x* is a two dimensional contingency table in matrix form.

wilcoxon test

wilcox.test(1:10, 1:20)

Question: Why we don't use t test

Discussion :

Scatter plot
Bar plot
Box plot
Bar plot