# R Programming Basics - part 2

## Outline

- Flow Control
- Functions
- R for statistics

# Flow Control

## Conditional execution

### *if* statement

If a logical condition holds, execute one or more statements

syntax: if (condition) {executable statements}

```
nclasses=2
fulltime=FALSE
if (nclasses >= 3){
  fulltime=TRUE}
fulltime
```

```
## [1] FALSE
```

### *if else* pair

If a condition holds, execute one set of statements, or else execute a second set of statements.

syntax: if (condition) { executable statements 1 } else { executable statements 2 }

```
fulltime=NA
nclasses=1
if (nclasses >= 3) {fulltime=TRUE
} else {fulltime=FALSE}
fulltime
```

```
## [1] FALSE
```

It is essential to begin the second line of this if/else pair as "} else". In this case the R interpreter "waits" for the "}", and sees it paired with the "else". The "if" statement would be completed if the "}" was included on the end of the first line.

Often one wants to use several "else if" clauses. The following example takes a numerical grade, and assigns a letter grade according to the Dalhousie common grade scale.

```
  grade=91.5
  Lgrade=NA
if (grade >=90) {Lgrade="A+"
                 print("Good job!")
} else
    if (grade >= 85) {Lgrade="A"
} else
```

```r
    if (grade >= 80) {Lgrade="A-"
} else
    if (grade >= 77) {Lgrade="B+"
} else
    if (grade >= 73) {Lgrade="B"
} else
    if (grade >= 70) {Lgrade="B-"
} else
    if (grade >= 65) {Lgrade="C+"
} else
    if (grade >= 60) {Lgrade="C"
} else
    if (grade >= 55) {Lgrade="C-"
} else
    if (grade >= 50) {Lgrade="D"
    } else {Lgrade="F"; print("Too bad!")}
```

```
## [1] "Good job!"
```

```r
Lgrade
```

```
## [1] "A+"
```

Example: determine if a number is positive, negative or 0.

```r
a <- 0
if (a < 0) {
   print("a is a negative number")
} else if (a > 0) {
   print("a is a positive number")
} else {
   print("a is zero")
}
```

```
## [1] "a is zero"
```

### ifelse

***ifelse* is a compact version of *if . . . else . . .***

### Syntax

```r
ifelse(condition, statement1, statement2)
```

There is a subtle difference in that *ifelse* works on vectors.

### Examples

```r
#find the minimum of scalars a and b
a=rnorm(1); a
```

```
## [1] 1.221303
```

```r
b=rnorm(1); b
```

```
## [1] -1.36175
```

```r
minab=ifelse(a<b,a,b); print(minab)
```

```
## [1] -1.36175
```

```r
#find the component by component minimum of vectors a and b
a=rnorm(10);
b=rnorm(10);
vecminab=ifelse(a<b,a,b); print(cbind(a,b,vecminab))
```

```
##                a          b   vecminab
##  [1,]  0.2382978  1.4205132  0.2382978
##  [2,]  0.3469096 -0.2634322 -0.2634322
##  [3,]  1.2586368  1.3490283  1.2586368
##  [4,] -1.2703117 -0.2585920 -1.2703117
##  [5,] -0.3756399 -0.1392408 -0.3756399
##  [6,]  0.2513194 -1.9331005 -1.9331005
##  [7,]  0.5255681  1.7536043  0.5255681
##  [8,]  0.8685623  0.9200573  0.8685623
##  [9,]  1.2893349 -0.7981538 -0.7981538
## [10,]  1.5769835 -1.3398801 -1.3398801
```

```r
#determine if the elements of a vector are evenly divisible by 2
vec = 1:12
evenOdd = ifelse(vec %% 2 == 0, "even", "odd")
print(evenOdd)
```

```
##  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even"
## [11] "odd"  "even"
```

## Logical Operators

& (and), | (or), == (equals), ! (not)

- get help on logical operators

```r
#?"=="
```

## Truth tables

Suppose that $x$, $y$ are logical variables, taking values $T$ or $F$. The following "truth table" gives the values of the functions $x\&y$ (x and y), $x|y$ (x or y), and $!x$ (not x).

```
##          x     y   x&y   x|y    !x
## [1,]  TRUE  TRUE  TRUE  TRUE FALSE
## [2,]  TRUE FALSE FALSE  TRUE FALSE
## [3,] FALSE  TRUE FALSE  TRUE  TRUE
## [4,] FALSE FALSE FALSE FALSE  TRUE
```

These basic functions can be built up into more complex expressions.

Example: $(x|y)\&z$

```
##        x|y     z (x|y)&z
## [1,]  TRUE  TRUE    TRUE
## [2,]  TRUE FALSE   FALSE
## [3,] FALSE  TRUE   FALSE
## [4,] FALSE FALSE   FALSE
```

Example:$(x/&y)%7Cz$

```
##        x&y     z (x&y)|z
## [1,]  TRUE  TRUE    TRUE
```

```
## [2,]  TRUE FALSE    TRUE
## [3,] FALSE  TRUE    TRUE
## [4,] FALSE FALSE   FALSE
```

It is often useful to include brackets in order to specify the logical expression that you want. For example, suppose that you are interested to know whether (one or the other of x and y are TRUE) and (z is TRUE). Then you want to evaluate (x|y)&z. Suppose that you forget to include the brackets, and instead evaluate x|y&z. Are these the same? Sometimes yes, sometimes no.

```
x=c(rep(T,4),rep(F,4))
y=rep(c(rep(T,2),rep(F,2)),2)
z=rep(c(T,F),4)
cbind(x,y,z,(x|y)&z,x|y&z,x|(y&z))
```

```
##          x     y     z
## [1,]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [2,]  TRUE  TRUE FALSE FALSE  TRUE  TRUE
## [3,]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
## [4,]  TRUE FALSE FALSE FALSE  TRUE  TRUE
## [5,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
## [6,] FALSE  TRUE FALSE FALSE FALSE FALSE
## [7,] FALSE FALSE  TRUE FALSE FALSE FALSE
## [8,] FALSE FALSE FALSE FALSE FALSE FALSE
```

### Using logical values to subset data

- Examples:

```
vec = c(1, 3, 6, 2, 5, 10, 11, 9)

#return all values which are evenly divisible by 3
vec[vec%%3 == 0]
```

```
## [1] 3 6 9
```

```
#return all values which are not evenly divisible by 3
vec[!vec%%3 == 0]
```

```
## [1]  1  2  5 10 11
```

```
vec[!((vec%%3) == 0)]  #better to use brackets if you're unsure of operator precedence
```

```
## [1]  1  2  5 10 11
```

```
# return all the values in vec which are greater than 5
vec[vec > 5]
```

```
## [1]  6 10 11  9
```

```
# use "cars" data set, return all the rows where dist is less than 20
cars[cars$dist < 20, ]
```

```
##    speed dist
## 1      4    2
## 2      4   10
## 3      7    4
## 5      8   16
## 6      9   10
## 7     10   18
## 10    11   17
```

```
## 12    12   14
```

```r
# return all the rows where speed is 10
cars[cars[,1] == 10, ]
```

```
##   speed dist
## 7    10   18
## 8    10   26
## 9    10   34
```

###Example: is a specified year a leap year? (Uses &, |, and !)

```r
year = 2018
if ( (year %% 4 == 0 & year %% 100 != 0) | year %% 400 ==0){
    print(paste(year,"is a leap year"))
  } else {
    print("no")
  }
```

```
## [1] "no"
```

Here is essentially the same thing, where the logical operations are more clearly identified.

```r
  year=2018
  x=year %% 4 == 0
  y=year  %% 100 != 0
  z=year %% 400 ==0
  condition=x&y|z
  c(x,y,z,condition)
```

```
## [1] FALSE  TRUE FALSE FALSE
```

```r
  if(condition) print(paste(year,"is a leap year"))
```

####Example: find records in a database

```r
rm(list=ls()) #remove all objects
ls()
```

```
## character(0)
```

```r
subjectno=c(1:8)
#enter some data for first and last name, age
firstname=c("Dick","Jane","","Jian","jing","Li","John","Li")
lastname=c("Tracy","Doe","Smith","Yuan","Xian","Li","Doe","")
age=sample(c(18:35),8) #assign random ages from 18 through 35
data=data.frame(subject=subjectno,firstname=firstname,
      surname=lastname,age=age)
rm("subjectno","firstname","lastname","age")
ls()
```

```
## [1] "data"
```

```r
attach(data) #attach the dataframe data
data
```

```
##   subject firstname surname age
## 1       1      Dick   Tracy  20
## 2       2      Jane     Doe  23
## 3       3             Smith  32
## 4       4      Jian    Yuan  34
```

```
## 5       5      jing    Xian  31
## 6       6        Li      Li  33
## 7       7      John     Doe  24
## 8       8        Li          19
```

```r
#find subjects whose surname is "Li"
subset1=data[surname=="Li",];subset1
```

```
##   subject firstname surname age
## 6       6        Li      Li  33
```

```r
#Is there a Jane Doe in the database?
subset2=data[surname=="Doe"&firstname=="Jane",];subset2
```

```
##   subject firstname surname age
## 2       2      Jane     Doe  23
```

```r
#find subjects whose given or surname is missing ("")
subset3=data[surname=="" | firstname=="",];subset3
```

```
##   subject firstname surname age
## 3       3             Smith  32
## 8       8        Li          19
```

```r
#Find the subjects who are older than 29 years?
data[age>29,]
```

```
##   subject firstname surname age
## 3       3             Smith  32
## 4       4      Jian    Yuan  34
## 5       5      jing    Xian  31
## 6       6        Li      Li  33
```

```r
#find all subjects whose first name starts with "J"
#use substr(chvector,1,1) to extract the first character
#of each element of the character vector chvector
subset4=data[substr(firstname,1,1)=="J",];subset4
```

```
##   subject firstname surname age
## 2       2      Jane     Doe  23
## 4       4      Jian    Yuan  34
## 7       7      John     Doe  24
```

```r
#find all subjects whose first name starts with either "J" or "j"
subset5=data[substr(firstname,1,1)=="J"|substr(firstname,1,1)=="j",];subset5
```

```
##   subject firstname surname age
## 2       2      Jane     Doe  23
## 4       4      Jian    Yuan  34
## 5       5      jing    Xian  31
## 7       7      John     Doe  24
```

```r
detach(data) #detach the dataframe data
```

# Flow control

*for*, *while*, *repeat*, *break* and *next*

*for*, *while* and *repeat* are iterative constructs, meaning that a collection of R statements are repeated.

***for* is the most common loop structure**

-Syntax

```
for (var in range) {
    statements
}
```

- Examples:

```
# sum the numbers 1:k
k=12
mysum=0  #initialize the sum to 0
for (i in 1:k) {
  mysum=mysum+i}
print(mysum)
```

```
## [1] 78
```

- Example: Find which of the years 2000 through 2020 are leap years

```
for (i in 2000:2020){
  if ( (i %% 4 == 0 & i %% 100 != 0) | i %% 400 ==0){
    print(paste(i,"is a leap year"))
  }
}
```

```
## [1] "2000 is a leap year"
## [1] "2004 is a leap year"
## [1] "2008 is a leap year"
## [1] "2012 is a leap year"
## [1] "2016 is a leap year"
## [1] "2020 is a leap year"
```

Here is essentially the same thing, where the logical operations are more clearly

```
for (i in 2000:2020){
  x=i %% 4 == 0
  y=i %% 100 != 0
  z=i %% 400 ==0
  condition=x&y|z
  print(c(x,y,z,condition))
    if(condition) print(paste(i,"is a leap year"))

}
```

```
## [1]  TRUE FALSE  TRUE  TRUE
## [1] "2000 is a leap year"
## [1] FALSE  TRUE FALSE FALSE
## [1] FALSE  TRUE FALSE FALSE
## [1] FALSE  TRUE FALSE FALSE
## [1]  TRUE  TRUE FALSE  TRUE
## [1] "2004 is a leap year"
## [1] FALSE  TRUE FALSE FALSE
## [1] FALSE  TRUE FALSE FALSE
## [1] FALSE  TRUE FALSE FALSE
## [1]  TRUE  TRUE FALSE  TRUE
## [1] "2008 is a leap year"
## [1] FALSE  TRUE FALSE FALSE
```

```
## [1] FALSE  TRUE FALSE FALSE
## [1] FALSE  TRUE FALSE FALSE
## [1]  TRUE  TRUE FALSE  TRUE
## [1] "2012 is a leap year"
## [1] FALSE  TRUE FALSE FALSE
## [1] FALSE  TRUE FALSE FALSE
## [1] FALSE  TRUE FALSE FALSE
## [1]  TRUE  TRUE FALSE  TRUE
## [1] "2016 is a leap year"
## [1] FALSE  TRUE FALSE FALSE
## [1] FALSE  TRUE FALSE FALSE
## [1] FALSE  TRUE FALSE FALSE
## [1]  TRUE  TRUE FALSE  TRUE
## [1] "2020 is a leap year"
```

*Example*: Calculate the *inner product* of two vectors. Suppose that $u$ and $v$ are each numeric vectors of length $n$. Then the inner product of $u$ and $v$ is defined as

$$\sum_{i=1}^{n} u_i v_i$$

That is, we sum the products of the corresponding elements of the two vectors. This is easy to program using a *for* loop.

```
u=1:10
v=-10:-1
u*v
```

```
##  [1] -10 -18 -24 -28 -30 -30 -28 -24 -18 -10
```

```
prod=0
for (i in 1:10)prod=prod+u[i]*v[i]  #note no {} needed if everying on same line
print(prod)
```

```
## [1] -220
```

```
#verify
sum(u*v)
```

```
## [1] -220
```

Example: *matrix multiplication* Suppose that $U$ is a matrix with $m$ rows and $n$ columns, and that $V$ is a matrix with $n$ rows and $p$ columns. The the matrix product $P = UV$ is a matrix with $m$ rows and $p$ columns, and the entry in the $i$'th row and $j$'th column of $P$ is

$$P_{ij} = \sum_{k=1}^{n} U_{ik} V_{kj}$$

Define some conforming matrices U and V, in this example, U being 4 by 3 and V being 3 by 3 The product $P$ is a 4 by 3 matrix. Find $P_{2,3}$, the element in the 2nd row and 3rd column of $P$. By definition

$$P_{2,3} = \sum_{k=1}^{n} U_{2k} V_{k3}$$

```
U=matrix(1:12,byrow=T,ncol=3)
U # a 4 by 3 matrix
```

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

```
V=matrix(c(rep(1,3),c(-1,0,1),-3:-1),byrow=T,ncol=3)
V # a 3 by 3 matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]   -1    0    1
## [3,]   -3   -2   -1
```

```
U2=U[2,] #second row of U
V3=V[,3] #third column of V
P23=0
for (k in 1:3) P23=P23+U2[k]*V3[k]
P23
```

```
## [1] 3
```

We want to calculate $P_{i,j}$ for each pair of indices (i,j). Essentially

```
 for (i in 1:4){
   #for each i in (1,2,3,4), do something
  for (j in 1:3){
    #for each j in (1,2,3) do something
      #calculate Pij here
  } # end of j for loop
    } # end of i for loop
```

This can be calculated in R using three "nested" for loops.

```
#result will be a 4 by 3 matrix, having 12 elements
P=matrix(rep(0,12),byrow=T,ncol=3) #initialize product with 0's
for (i in 1:4){ #for each i, execute code until the closing }
  for (j in 1:3){ #for each j, execute code until the closing }
    for (k in 1:3){ # for each k, execute code until the closing }
      P[i,j]=P[i,j]+U[i,k]*V[k,j]
      }}}
P
```

```
##      [,1] [,2] [,3]
## [1,]  -10   -5    0
## [2,]  -19   -8    3
## [3,]  -28  -11    6
## [4,]  -37  -14    9
```

```
#verify using the builtin R matrix multiplication operator "%*%".

U%*%V
```

```
##      [,1] [,2] [,3]
## [1,]  -10   -5    0
## [2,]  -19   -8    3
## [3,]  -28  -11    6
## [4,]  -37  -14    9
```

Executing loops, in this case, "for" loops, in R is very inefficent as compared to many computer languages, such as Java or C, and using the built in functions (in this case %*%) is always recommended. That said, you need to know how to use loops effectively, in order to be able to do things in addition to calling the built in procedures.

The for loop in R is more general than the simlar construct in most programming languages. The general structure is

```
for (var in set) {
    statements
}
```

Where "set" is an arbitrary set, and the instructions are evaluated for each element of the set.

```
set=c("this","that","cat","mouse","male","female")
for (var in set) {
   print(var)
}
```

```
## [1] "this"
## [1] "that"
## [1] "cat"
## [1] "mouse"
## [1] "male"
## [1] "female"
```

### The basic definitions and a few simple examples are given below for the *while*, *repeat*, *break*, *next*

#### statements/constructs.

More meaningful examples will be given in the material on user defined functions.

#### *while*

#### Syntax

```
while (condition){
  statement
}
```

#### Example

```
x = 5
while(x <= 20){
  print(x)
  x = x+5
}
```

```
## [1] 5
## [1] 10
## [1] 15
## [1] 20
```

#### *repeat*

#### Syntax: repeat {statement}

**need to use break**

```
x = 5
repeat{
  print(x)
  x = x+5
  if (x > 20) break
}
```

```
## [1] 5
## [1] 10
## [1] 15
## [1] 20
```

**"break" and "next"**

**"break": break the current loop**

```
for (i in 1:6){
  if (i==5){
    break
  }
  print(i)
}
```

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

**"next": skip to next iteration**

```
for (i in 1:6){
  if (i==5){
    next
  }
  print(i)
}
```

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

# User-defined functions

-Syntax

```
funtionname= function(arg1,arg2,...){
  statements
  return(something)
}
```

-Examples:

```r
SumSquare = function(x,y){
    val = x^2+y^2
    return(val)
}
SumSquare(3,4)
```

```
## [1] 25
```

```r
SumSquares = function(x){
# function to create the sum of squares of elements of x
    temp=0
    for (i in 1:length(x)){
#     print(c(i,x[i],x[i]^2))
    temp=temp+x[i]^2}
    return(temp)
}

data=c(4,1,-2,5)
SumSquares(data)
```

```
## [1] 46
```

```r
SumSquares1 = function(x){
# another way to do the same thing
    temp=0
    for (i in x){
    print(c(i,i^2))
    temp=temp+i^2}
    return(temp)
}

SumSquares1(data)
```

```
## [1]  4 16
## [1] 1 1
## [1] -2  4
## [1]  5 25
```

```
## [1] 46
```

```r
SumSquares2 = function(x){
# if statement checks that the input argument x
# is a numeric vector.  If it is not, print an
# error message, and return a NULL value
    if(!is.vector(x)|!is.numeric(x)){
       print("x should be a numeric vector")
       return(NULL)}
# otherwise, return the sum of the squared elements of x
    temp=0
    for (i in 1:length(x))temp=temp+x[i]^2
    return(temp)
}

SumSquares2(data)
```

```
## [1] 46
```

```r
SumSquares2(c("a","b"))
```

```
## [1] "x should be a numeric vector"
## NULL
```

```r
SumSquares2(matrix(1:4,byrow=T,ncol=2))
```

```
## [1] "x should be a numeric vector"
## NULL
```

## Exercise: create a function for finding leap years

- input: startYear, endYear
- output: return a vector of all the leap years between startYear and endYear
- Using a for loop

```r
leapYears1 = function(startYear,endYear){
  output = NULL
#uses a for loop
  for (year in c(startYear:endYear)){
  if ( (year %% 4 == 0 & year %% 100 != 0) | year %% 400 ==0){
    output = c(output,year)
  }}
  return(output)
}

leapYears1(2018,2028)
```

```
## [1] 2020 2024 2028
```

- Use *while* or *repeat* for looping

```r
leapYears2 = function(startYear,endYear){
  results = c()
#uses a while loop, as opposed to a for loop
  year = startYear
  while (year <= endYear){
  if ( (year %% 4 == 0 & year %% 100 != 0) | year %% 400 ==0){
    results = c(results,year)
  }
    year = year + 1
  }
  return(results)
}

leapYears2(2018,2028)
```

```
## [1] 2020 2024 2028
```

```r
leapYears3 = function(years=2019){
  #uses built in vectorized logical indexing
  return(years[(years %% 4 == 0 & years %% 100 != 0) | years %% 400 ==0])}

leapYears3(2018:2028)
```

```
## [1] 2020 2024 2028
```

```
test=leapYears3(1867:2019)
test
```

```
##  [1] 1868 1872 1876 1880 1884 1888 1892 1896 1904 1908 1912 1916 1920 1924 1928
## [16] 1932 1936 1940 1944 1948 1952 1956 1960 1964 1968 1972 1976 1980 1984 1988
## [31] 1992 1996 2000 2004 2008 2012 2016
```

-Example: function calling another function

Recall that the formula for the sample variance of $x_1, x_2, \ldots, x_n$ is

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where $\bar{x} = \sum_{i=1}^{n} x_i/n$.

```
mymean=function(x){
#returns the sample mean of the values in the vector x
  mysum=0
  for (i in x){
    mysum=mysum+i
  }
  return(mysum/length(x))
}

myvar=function(x){
  n=length(x)
# subtract the mean from the values in x
  data=x-mymean(x)
# sum the squares of the entries of data
  myvar=SumSquares(data)
# divide the sum of squares by n-1
  myvar=myvar/(n-1)
  return(myvar)
  }

myvar(data)
```

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

```
#check using the built in function var

var(1:7)
```

```
## [1] 4.666667
```

-How fast is our user defined function as compared to the builtin function *var*?

```
v=rnorm(50000) #vector of 50000 observations from the standard normal
start1=Sys.time()
var(v) #sample variance of v using builtin function
```

```
## [1] 0.9928466
```

```
end1=Sys.time()
tm1=end1-start1
tm1 #elapsed time
```

```
## Time difference of 0.001680136 secs
```

```
start2=Sys.time()
myvar(v) #sample variance of v using user defined function
```

```
## [1] 0.9928466
```

```
end2=Sys.time()
tm2=end2-start2
tm2 #elapsed time
```

```
## Time difference of 0.009307146 secs
```

It is important to be able to write user defined functions in order to make extensions to the language. However, the built in functions, which use vectorized arithmetic, with calls to more efficient languages, are typically much faster, and so the recommended choice with even moderate sized data sets.

##Scope: the *scope* of a variable tells us which version of the variable is being used. Variables can be local or global. A variable defined within a function is local to that function.

TRICK to remember: Variables go in but do not go out.

By this we mean that:

1. variables that are defined in the main R program keep their values (if not redefined) inside functions (variables go in)
2. variables only defined in a function have a scope limited to the function, and disappear (are undefined) back in the main program (variables do not go out)

The following exercises are just ways to familiarize you with variable scopes. You can create your own variants.

For example, test the following:

- define a and b in the main program
- modify a and b inside a function (not passing a or b as arguments)
- print a and b in the main program
- what are your conclusions?

Example: only z exists in the global environment. *a* and *b* are defined within the function *test*. They are local to *test* and not available in the global environment after the function is run.

```
rm(list=ls())  #clear everything in the global environment

z=10
ls()
```

```
## [1] "z"
```

```
test=function(x){
  a=1
  b=2
  y=a+b*x
  return(y)
  }

test(z)
```

```
## [1] 21
```

```r
ls()
```

```
## [1] "test" "z"
```

Example 2: $a$, $b$ and $z$ exist in the global environment. Only $y$, and the argument to the function, $x$, are available within the function. When the function *test* is defined, the values of $a$ and $b$ available at that time are used in the definition.

```r
rm(list=ls())  #clear everything in the global environment

a=1
b=2
z=10
ls()
```

```
## [1] "a" "b" "z"
```

```r
test=function(x){
  y=a+b*x
  print(ls()) #ls lists the variables in the local environment
              #note there is no a or b local to the function
  print(a)    #the values printed are those from the global
  print(b)    #environment when the function was defined
  return(y)
  }

test(z)
```

```
## [1] "x" "y"
## [1] 1
## [1] 2
```

```
## [1] 21
```

```r
ls()
```

```
## [1] "a"    "b"    "test" "z"
```

Example 3: $a$, $b$ and $z$ exists in the global environment. $a$ and $b$ are also defined within the environment which is local to the function. The local versions are used within the function.

```r
rm(list=ls())  #clear everything in the global environment

a=1
b=2
z=10
ls()
```

```
## [1] "a" "b" "z"
```

```r
test=function(x){
  a=10
  b=20
  y=a+b*x
  print(ls()) #ls lists the variables in the local environment
  return(y)
  }

output=test(z)  #now there will be a variable *output* in the global environment.
```

```
## [1] "a" "b" "x" "y"
```

```r
ls()
```

```
## [1] "a"      "b"      "output" "test"   "z"
```

```r
a   #these were he original, global versions of *a* and *b*
```

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

```r
b   #the local versions were local to the function test only
```

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

- A more advanced example: suppose we have one function defined within another, and the same variable name used in each function, and globally. Before evaluating the following code, see if you can understand what the final result *f(10)+a* will be, and also, any intermediate outputs. What would the result be if you remove the line *a=3* in function g? Which value of *a* will function *g* use? In this case, the value of *a* used in function *g* is that value which was present in the in the environment where *g* was defined, namely *a=2*.

```r
rm(list=ls())
a=1
z=10

f=function(x){
  a=2
  print(c(x,a))

  g=function(x){
  a=3
  print(c(x,a))
  resultg=a+x
  print(paste("function g returns ",resultg))
  return(resultg)}

    resultf=g(x+5)+a
    print(paste("function f returns ",resultf))

    return(resultf)
    }

f(10)+a
```

### Advanced topic: the argument ...

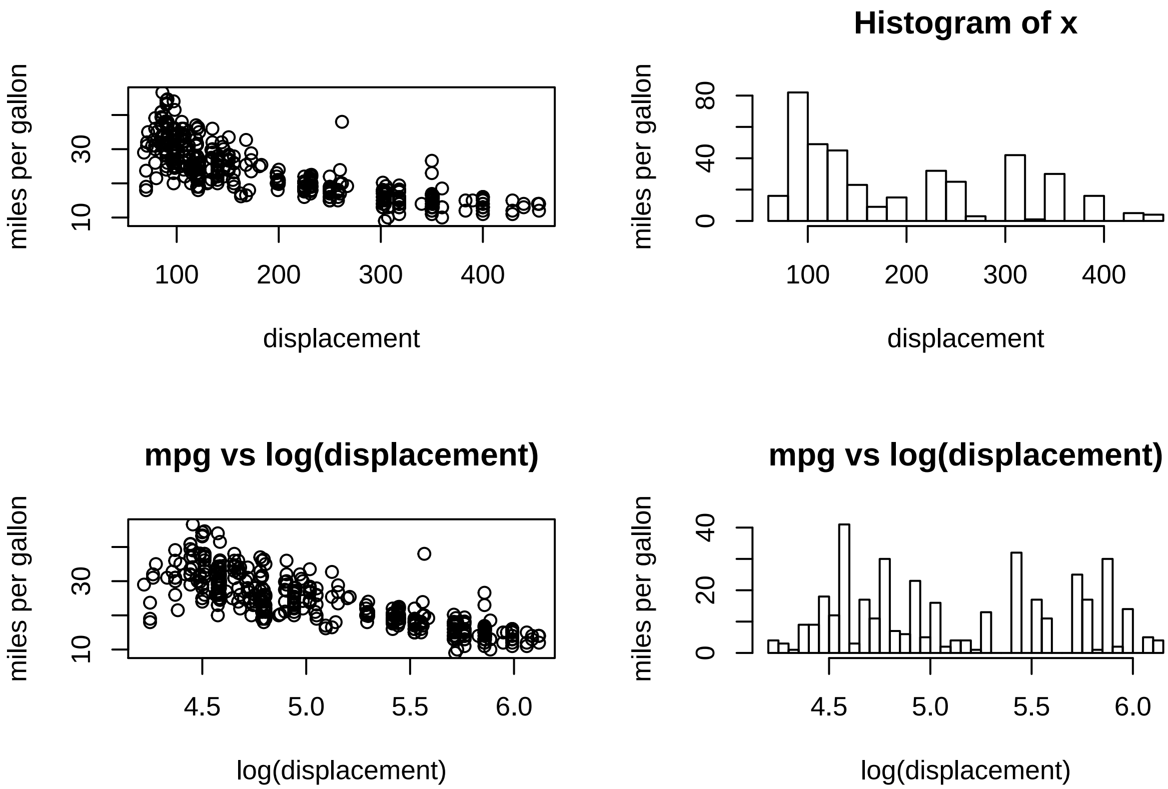
Usually a function is called with a fixed number of arguments. In more advanced applications, you may want to pass in arguments only on some occasions, and pass them directly through to other functions within the function you have written. The argument ... is a placeholder for any number of named argments, which can be passed through to other functions.

For example, suppose that we want a function which makes plots of both $y$ vs $x$, and also a histogram of $x$. We would like to use the function with variables of any name, but we would like to be able to pass in labels to the function to better identify the variables. Let's work with the *Auto* data, and make plots using *mpg* as the y-variable , both vs both *displacement* and *log(displacement)* as the x-variable, and let's pass the labels through to the plot and histogram functions.

```
data=mydata=read.csv("http://faculty.marshall.usc.edu/gareth-james/ISL/Auto.csv")
attach(data)
par(mfrow=c(2,2))  #set graphics region to have two rows, two columns

plot2=function(x,y,nclassin=20,...){
 #this function first makes a plot of y vs x.  Notice that ... is passed through
 #from the plot2 function definition
 plot(x,y,...)
 #next make a histogram of x.  Notice that the default number of histogram bars is 20
 hist(x,nclass=nclassin,...)
  }

plot2(displacement, mpg, xlab="displacement", ylab="miles per gallon")
plot2(log(displacement), mpg, xlab="log(displacement)", ylab="miles per gallon",main="mpg vs log(displa
```




### Recursion - a recursive function calls itself.

- Example 1: for non-negative integer $n$, "n factorial", written as $n!$ is defined as

$$!n = n!(n-1) = n(n-1)!(n-2) = \ldots = n(n-1)(n-2)\ldots(2)(1)$$

with the consistency condition $0! = 1$.

The following gives R code for calculating $n!$ using both recursion, and a calculation using a *for* loop.

```
fact0=function(x){
   if(x==1){return(1)
   } else
    {return(x*fact0(x-1))}
```

```r
}
fact0(6)
```

```
## [1] 720
```

```r
#better to force the input value to be an integer
#and to include some comment as to what the function is doing
fact1=function(x){
  #returns x! for x a positive integer
  x=as.integer(x)
   if(x<0){
   print("Error: x must be a positive integer")
   return(NULL)} else {
   if(x==0|x==1){
   return(1)} else {
   return(x*fact1(x-1))}
   }}

fact1(6)
```

```
## [1] 720
```

```r
#fact2 calculates factorial using a for loop
fact2=function(x){
#returns x! for x a positive integer
  x=as.integer(x)
   if(x<0){
   print("Error: x must be a positive integer")
   return(NULL)} else {
   if(x==0|x==1){
   return(1)} else {
  temp=1
  for (i in 2:x) temp=temp*i
  return(temp)
  }}}
fact2(6)
```

```
## [1] 720
```

```r
start=Sys.time()
fact1(60)
```

```
## [1] 8.320987e+81
```

```r
end=Sys.time()
end-start
```

```
## Time difference of 0.001328945 secs
```

```r
start=Sys.time()
fact2(60)
```

```
## [1] 8.320987e+81
```

```r
end=Sys.time()
end-start
```

```
## Time difference of 0.005215883 secs
```

```r
start=Sys.time()
factorial(60)
```

```
## [1] 8.320987e+81
```

```r
end=Sys.time()
end-start
```

```
## Time difference of 0.001406908 secs
```

#Some Useful Functions for Statistics

## Arithmetic

- Arithmetic Operators
- Mathematic Functions

## Vectorized Arithmetic

```r
testVect = c(1,3,5,2,9,10,7,8,6)

min(testVect) # minimum
```

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

```r
max(testVect) # maximum
```

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

```r
mean(testVect) # mean
```

```
## [1] 5.666667
```

```r
median(testVect) # median
```

```
## [1] 6
```

```r
quantile(testVect) # quantile
```

```
##   0%  25%  50%  75% 100%
##    1    3    6    8   10
```

```r
var(testVect) #variance
```

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

```r
sd(testVect) # standard deviation
```

```
## [1] 3.162278
```

```r
vect1 = cars$speed
vect2 = cars$dist

cov(vect1,vect2) # covariance
```

```
## [1] 109.9469
```

```r
cor(vect1,vect2) # correlation coefficient
```

```
## [1] 0.8068949
```

**apply, inner product, outer product**

The 'apply' function gives you a way to perform flexible operations on arrays.

The syntax is:

apply(X, MARGIN, FUN, ...)

For details on the syntax of the 'apply' family of functions, please go to:

https://www.datacamp.com/community/tutorials/r-tutorial-apply-family

```
# Construct a 5x6 matrix
X <- matrix(rnorm(30), nrow=5, ncol=6)

# Sum the values of each column with `apply()`
apply(X, 2, sum)
```

```
## [1] -0.56756455  0.04624865 -1.97394254  0.25817568  2.64421593 -0.67576105
```

```
product <- outer(0:1, 0:1, "*")
#     [,1] [,2]
#[1,]    0    0
#[2,]    0    1
product2 <- outer(product,product,"*")
product
```

```
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    1
```

```
product2
```

```
## , , 1, 1
##
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
##
## , , 2, 1
##
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
##
## , , 1, 2
##
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
##
## , , 2, 2
##
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    1
```

##Probability Distributions

- key words

  - *d* : density (returns the height of the pdf)
  - *p* : distribution function (returns the cdf)
  - *q* : quantile function (returns the inverse cdf)
  - *r* : random generation
- distributions
  - *binom* : Binomial Distribution
  - *pois* : Poisson Distribution
  - *unif* : Uniform Distribution
  - *exp* : Exponential Distribution
  - *norm* : Normal Distribution
  - *chisq* : Chi-Squared Distribution
  - *t* : t Distribution
  - *f* : F Distribution

What is the probability of four or less questions answered correctly by random in a twelve question multiple choice quiz?

What is the threshold q that I should place on a random variable z drawn from a standard normal distribution to make sure that z is less than q with probability 0.75?

How can I produce a series of independent Poisson distributed counts?

Mathematical expressions of densities (pdfs):

$d \sim e^{-x^2}$

$d = \binom{n}{x} p^x (1-p)^{n-x}$

https://www.calvin.edu/~rpruim/courses/s341/S17/from-class/MathinRmd.html

- Examples

```
# binomial  probability of having 2 successes in 10 Benoulli draws of probability 0.2
dbinom(2, size=10, prob=0.2)
```

```
## [1] 0.3019899
```

```
dbinom(0, size=10, prob=0.2) + dbinom(1, size=10, prob=0.2) + dbinom(2, size=10, prob=0.2)
```

```
## [1] 0.6777995
```

```
pbinom(2,size=10,prob=0.2)
```

```
## [1] 0.6777995
```

```
runif(6,min=1,max=2)
```

```
## [1] 1.477235 1.591193 1.416582 1.448760 1.166077 1.165975
```

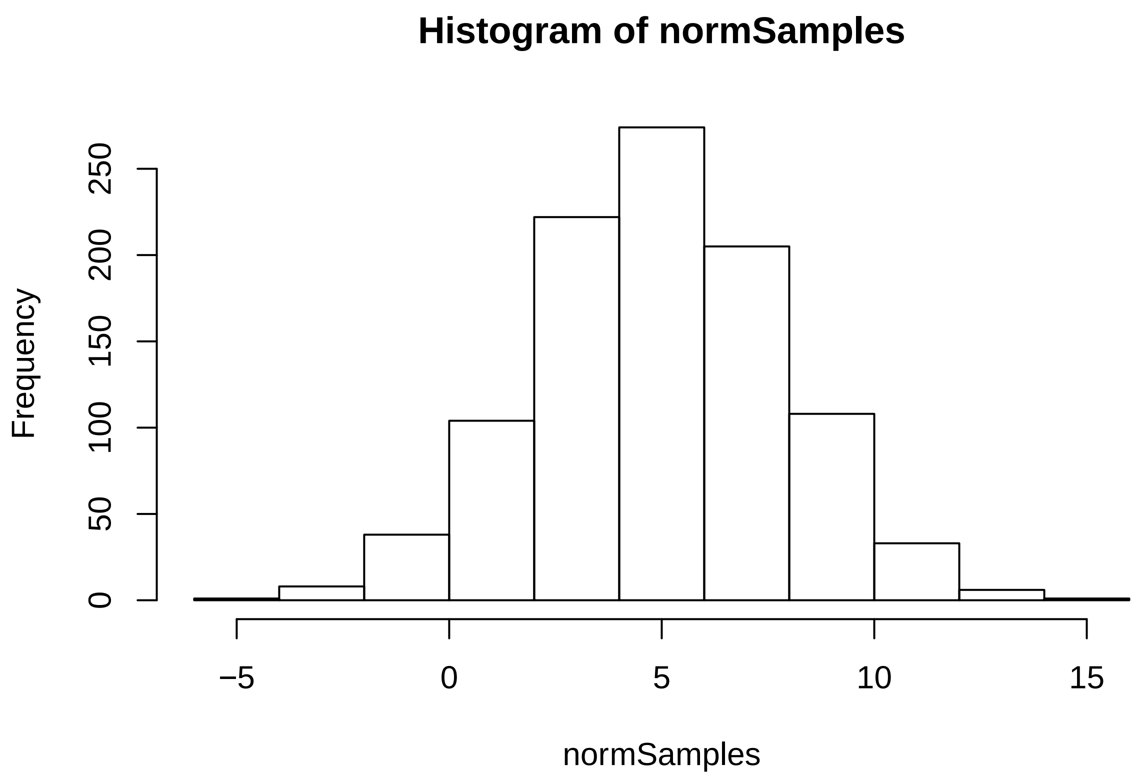
```
qt(c(.025, .975), df=4)
```

```
## [1] -2.776445  2.776445
```

```
qf(.95, df1=3, df2=4)
```

```
## [1] 6.591382
```

```
normSamples = rnorm(1000,mean=5,sd = 3)
hist(normSamples)
```




- *set.seed()* reproduce the results even using random

```r
set.seed(100)
rnorm(5)
```

```
## [1] -0.50219235  0.13153117 -0.07891709  0.88678481  0.11697127
```

```r
rnorm(5)
```

```
## [1]  0.3186301 -0.5817907  0.7145327 -0.8252594 -0.3598621
```

```r
set.seed(100) # reproduce the results
rnorm(5)
```

```
## [1] -0.50219235  0.13153117 -0.07891709  0.88678481  0.11697127
```

```r
rnorm(5)
```

```
## [1]  0.3186301 -0.5817907  0.7145327 -0.8252594 -0.3598621
```