R Programming Basics - User defined functions

Contents

0.1 General Syntax

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

The usual function has a first line which provides a name for the function, and zero or more arguments, arg1, arg2, . . . , argk in he above. The arguments are variable names which will be used inside the function, and into which, values will be passed. The last line in a function is usually a return function, which says what value to return when the function terminates.

0.1.1 Some Examples:

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

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

In this case, when the function is entered, the variables x and y take values 3 and 4, respectively. *val*, which was calculated in the body of the function, is returned by the *return* statement.

```
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
```

It is good programming practice to check for validity of the arguments when a function is called. In the following version of the sum of squares function, it is first checked that the input argument x is a numeric vector.

```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
```

- Example: create a function for finding leap years
- input: startYear, endYear
- output: return a vector of all the leap years between startYear and endYear

```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
```

- leap year example using a *while* loop

```
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
```

- leap year example using vectorized logical indexing

```
leapYears3 = function(years=2019){
  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
## [15] 1928 1932 1936 1940 1944 1948 1952 1956 1960 1964 1968 1972 1976 1980
## [29] 1984 1988 1992 1996 2000 2004 2008 2012 2016
```

0.2 Calling a function from within another function.

Many functions call other functions, either user defined, or built in.

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 $\bar{x} = \sum_{i=1}^{n} x_i/n$ is the sample mean.

```r
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
```

```r
#check using the built in function var

var(data)
```

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

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

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

```
## [1] 1.001174
```

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

```
## Time difference of 0.001759529 secs
```

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

```
## [1] 1.001174
```

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

```
## Time difference of 0.009550095 secs
```

It is important to be able to write user defined functions in order to make extensions to the language. However, as this example shows, 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 is to use builtin functions when available.

0.3 Scope:

tTe *scope* of a variable tells us which version of the variable is being used. Variables can be local or global. When you first enter R, you are working in the global environment, and the variables which you see there using the *ls()* command are referred to as global variables. A variable defined within a function is local to that function.

0.3.1 Some examples:

- Example 1:

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
```

```
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.

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

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

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

```
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
```

```
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.

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

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

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

```
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"
```

```
ls()
```

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

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

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

```
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
```

0.4 Advanced topic: the argument ...

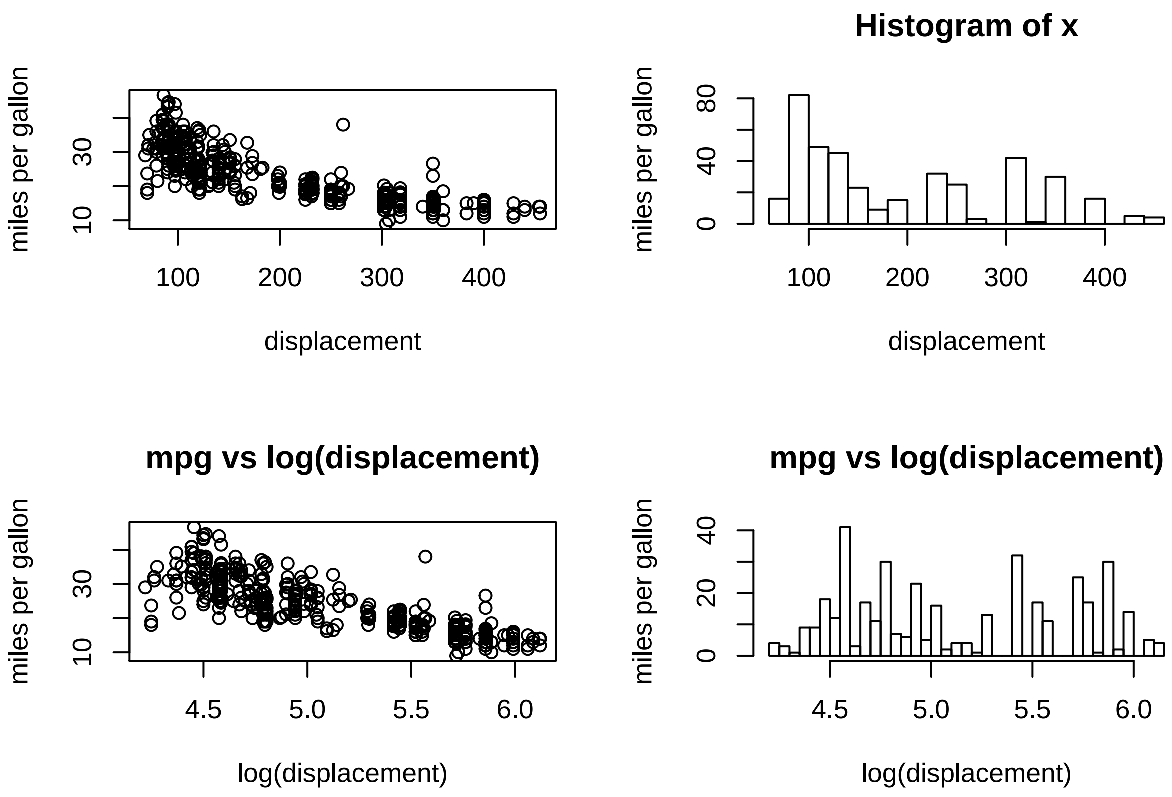
Usually a function is called with a fixed number of arguments. In more advanced applications, you may want to pass in arguments only on some occasions, and pass them directly through to other functions within the function you have written. The argument ... is a placeholder for any number of named argments,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

```r
data=mydata=read.csv("http://www-bcf.usc.edu/~gareth/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
```



0.5 Recursion

A *recursive* function calls itself.

Recursive programs often look very elegant as compared non-recursive functions which carry out the same task, and the recursive function often mimics a mathematical construct very closely.

0.5.1 Some examples:

The prototypic example of the use of recursion is in the calculation of factorials.

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 $n!$ recursion.

```
fact0=function(x){
   if(x==1){return(1)
   } else
    {return(x*fact0(x-1))}
}
fact0(6)
```

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

```
#better to force the input value to be an integer
#and to include some comment as to what the function is doing
#also, include the consistency condition 0!=1.

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)}
   if(x==0|x==1){return(1)}
   return(x*fact1(x-1))
   }


fact1(6)
```

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

- Example:
- The following calculates the 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)}
   if(x==0|x==1){return(1)}
  temp=1  #initialize
  for (i in 2:x)temp=temp*i #iterate
  return(temp)
}

fact2(6)
```

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

- The following calculates the factorial using a *while* loop.

```
fact3=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)}
   if(x==0|x==1){return(1)}

   temp=1
   while(x>1){
     temp=temp*x
     x=x-1
     }
   return(temp)
   }
```

```r
fact3(6)
```

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