R Programming Basics - User defined functions

Contents

1 Outline

- TOPIC 1: How to create functions in R
- TOPIC 2: Scope of variables
- TOPIC 3: Mechanics of arguments and returned values
- TOPIC 4: Recursive functions

2 TOPIC 1: How to create functions in R

General Syntax

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

UDF: User-defined functions (as opposed to built-in functions in R or R packages)

Learn how to create UDFs

2.1 Examples

- Example 1: write a function to compute the sum of squares of two scalar arguments passed to the function

```
SquareNorm1 = function(x,y){
    val = x^2+y^2
    return(val)
}
```

Call the function

```
# x=3 y=4
  out = SquareNorm1(3,4)
  out
```

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

Make the function more general so that as an argument we can take a vector of any length. There are different ways to implement this function. In the first version, we will use a for loop to calculate the relevant sum.

```
# x is now a vector of any length

SquareNormAny = function(x){

    temp=0 # will store the sum of squares of each coordinate of vector x
    for (i in 1:length(x)){
      #i is the index that iterates through each element of vector x

      temp=temp+x[i]^2
    }
    return(temp)
}

data1=c(3,4)
SquareNormAny(data1) #Same answer!
```

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

```
data2=c(4,1,-2,5) #Let's try a longer vector
SquareNormAny(data2)
```

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

```
SquareNormAny1 = function(x){
# another way to do the same thing
    temp=0
    for (xi in x){
      #print(c(xi,xi^2)) #We can use the print function to print variable values within a loop
      temp=temp+xi^2
      }
    return(temp)
}

SquareNormAny1(data1)
```

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

```
SquareNormAny1(data2)
```

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

Let's try another way of implementing the function using the sum function.

```
SquareNormAny2 = function(x){
    temp = sum(x^2) #Exercise 1: Can you "dissect" this line to figure out what operations it performs?
    return(temp)
}
```

```
SquareNormAny2(data1)
```

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

```
SquareNormAny2(data2)
```

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

Let's return to our first version of the function. Let's add checks of arguments.

```
SumSquares = 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)
}

SumSquares(data1)
```

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

```
SumSquares(c("a","b"))
```

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

```
## NULL
```

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

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

```
## NULL
```

Exercise 2:

Write your own function, named mysqrt, with a single argument, which calculates the square root of the argument if it is a positive number, and writes an error message if it is a negative number.

```
#mysqrt =

#mysqrt(-2)
#mysqrt(3)
```

- Example 2 : create a function for finding numbers in a sequence evenly divisible by a divisor

Recall our example of evenly dividing by 3 from Module 1. We will now use the modulo operator to create a function.

- input: lower, upper, divisor
- output: return a vector of all the numbers that are evenly divisible by the divisor between lower and upper

solution 1

first implementation of a function to find evenly divided numbers in a given interval using a for loop

```
even_div = function(lower,upper,divisor){


    output = NULL

#uses a for loop

  for (i in lower:upper){

  if (i %% divisor ==0){
    output = c(output,i)
  }

  }

  return(output)
}

even_div(1,20,3)
```

```
## [1]  3  6  9 12 15 18
```

solution 2: while loop

```
even_div2 = function(lower,upper, divisor){
  results = c()
#uses a while loop, as opposed to a for loop
  i = lower
  while (i <= upper){
    if (i %% divisor == 0){
      results = c(results,i)
    }
    i = i + 1
  }
  return(results)
}

even_div2(1,20,3)
```

```
## [1]  3  6  9 12 15 18
```

solution 3: logical indexing

```
even_div3 = function(interval=1,divisor){
  return(interval[interval %% divisor == 0])}

# we use interval[condition]

# we do NOT pass the ends of the interval but the vector for all values in the interval
even_div3(1:20,3)
```

```
## [1]  3  6  9 12 15 18
```

```
test=even_div3(1:20,3)
test
```

```
## [1]  3  6  9 12 15 18
```

2.2 Building complex functions from simpler ones

Functions can call other functions.

This is called nesting.

- Example: compute the sample variance of a numerical vector

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.

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

myvar=function(x){
  n=length(x)
# subtract the mean from the values in x     this will center the vector 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)
  }
```

```
# try it
data <-   c(0.1,0.8,-0.2,0.5,0.6,-1.4,2.1)
myvar(data)
```

```
## [1] 1.129524
```

check: use built-in function var:

```
var(data)
```

```
## [1] 1.129524
```

2.3 Efficient programming/optimization

Compare speeds of execution of a user-defined function (UDF) versus a built-in function

```
set.seed(1)
v=rnorm(50000) # vector of 50000 observations from the standard normal


start1=Sys.time()   # clock time now before doing a task
var(v) #sample variance of v using builtin function    this will take some time to run
```

```
## [1] 1.011307
```

```
end1=Sys.time()     # clock time after execution
tm1=end1-start1
tm1 #elapsed time  for the built-in function var
```

```
## Time difference of 0.002010822 secs
```

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

```
## [1] 1.011307
```

```
end2=Sys.time()
tm2=end2-start2
tm2 #elapsed time  for my own function
```

```
## Time difference of 0.008890867 secs
```

```
c(tm1,tm2)
```

```
## Time differences in secs
## [1] 0.002010822 0.008890867
```

If you are interested in this topic, the microbenchmark function in the microbenchmark package is a nice (and more accurate) way to compare functions. It runs functions multiple times and gives summary statistics on the execution time.

```
#install.packages("microbenchmark")
library(microbenchmark)

microbenchmark(var(v),myvar(v))
```

3 TOPIC 2: Scope of variables

Definition. Local/Global

-The scope of a variable is basically the domain of the program where the value of the variable is 'visible'.

-Variables that are defined in the global environment are accessible inside functions (variables go in)

-Variables that are defined inside functions are local to the function and not accessible outside the function (variables do not go out).

-Global variables are those variables which exists throughout the execution of a program. It can be changed and accessed from any part of the program.

-Local variables are those variables which exist only within a certain part of a program (e.g. a function) and is released when the function call ends.

-When we pass arguments to a function, the function creates local copies of the corresponding variables, so that modifying these copies will have no effect on the variables passed one level up

Each environment (global or local) has a name

You can find out the environment you are working in like this:

```
#
environment()
```

```
## <environment: R_GlobalEnv>
```

Each function creates its own environment and we can print it inside the function. Please note

```
# I define three nested functions ftop, fmiddle (called by ftop) and fbottom (called by fmiddle)
# and I print the function name, the environment and the list of variables in the environment. This cod

fbottom<- function(argument_fbottom){
  print("Inside fbottom")
  print(environment())
  print(ls())
}

fmiddle <- function(argument_fmiddle){
  fbottom(2)
  print("Inside fmiddle")
  print(environment())
  print(ls())
}


ftop    <- function(argument_ftop){
  fmiddle(7)
  print("Inside ftop")
  print(environment())
  print(ls())
}

ftop(3)
```

```
## [1] "Inside fbottom"
## <environment: 0x7fe8138d9510>
## [1] "argument_fbottom"
## [1] "Inside fmiddle"
## <environment: 0x7fe8138d9660>
## [1] "argument_fmiddle"
## [1] "Inside ftop"
## <environment: 0x7fe8138d97b0>
## [1] "argument_ftop"
```

- Example 1:

  Global: z Local: a,b,x,y

```
ls()
```

```
##  [1] "data"           "data1"          "data2"          "end1"
##  [5] "end2"           "even_div"       "even_div2"      "even_div3"
##  [9] "fbottom"        "fmiddle"        "ftop"           "mymean"
## [13] "myvar"          "out"            "SquareNorm1"    "SquareNormAny"
## [17] "SquareNormAny1" "SquareNormAny2" "start1"         "start2"
## [21] "SumSquares"     "test"           "tm1"            "tm2"
## [25] "v"
```

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

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

```
z=10
ls()
```

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

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

#z=10


ls()
```

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

```
test=function(x){ # name of argument in DEFINITION is dummy
                  # is a local copy of the argument passed
  a=1 # this variable is local to the function
  b=2 # this variable is also local
  y=a+b*x
  return(y)
}


test(z)  # name of argument PASSED is critical
```

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

```
#x       # NOTE that this is undefined !
ls()
```

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

- Example 2:

  Global: z Local: a

```
rm(list=ls())  #clear everything in the global environment
z=10
ls()
```

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

```
test=function(){  # name of argument is dummy
                  # is a local copy of the argument passed
  a=1
  print(c("a exists within the function but z is global and not listed : ",ls()))
  return(1+z)
  }
test()  # name of argument is important
```

```
## [1] "a exists within the function but z is global and not listed : "
## [2] "a"
```

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

```
z       # this is unchanged !
```

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

```
ls()    # NOTE how a is not listed (scope local to function)
```

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

- Example 3:

Global: a,b,z Local: x,y

```
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 4:

Local versions stay local

'Global goes in, local does not come out'

```
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   # original, global version of *a* , not the local version
```

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

```
b   # original, global version of *b* , not the local version
```

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

Link: variable scope and environment in R

4 TOPIC 3: Mechanics of arguments and returned values

4.0.1 Setting default values / no need to pass all arguments

Below is a function that accepts scalars n and p and returns $\frac{n!}{p!}$

```
factnp      <-   function(n,p=1){

# we can give default values to some or all the arguments


# default values are flexible (they simplify the calls)
# default values can be cryptic
# LOOK at the R documentation to identify all arguments

anp =  factorial(n)/factorial(p)
return( anp  )
}

factnp(6,2)
```

```
## [1] 360
```

```
#Let's check the default argument p=1:
for(i in 1: 8){print(factnp(i) ==  factnp(i,1))}
```

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

4.0.2 (Advanced topic:) the argument ...

plot2=function(x,y,nclassin=20,...)

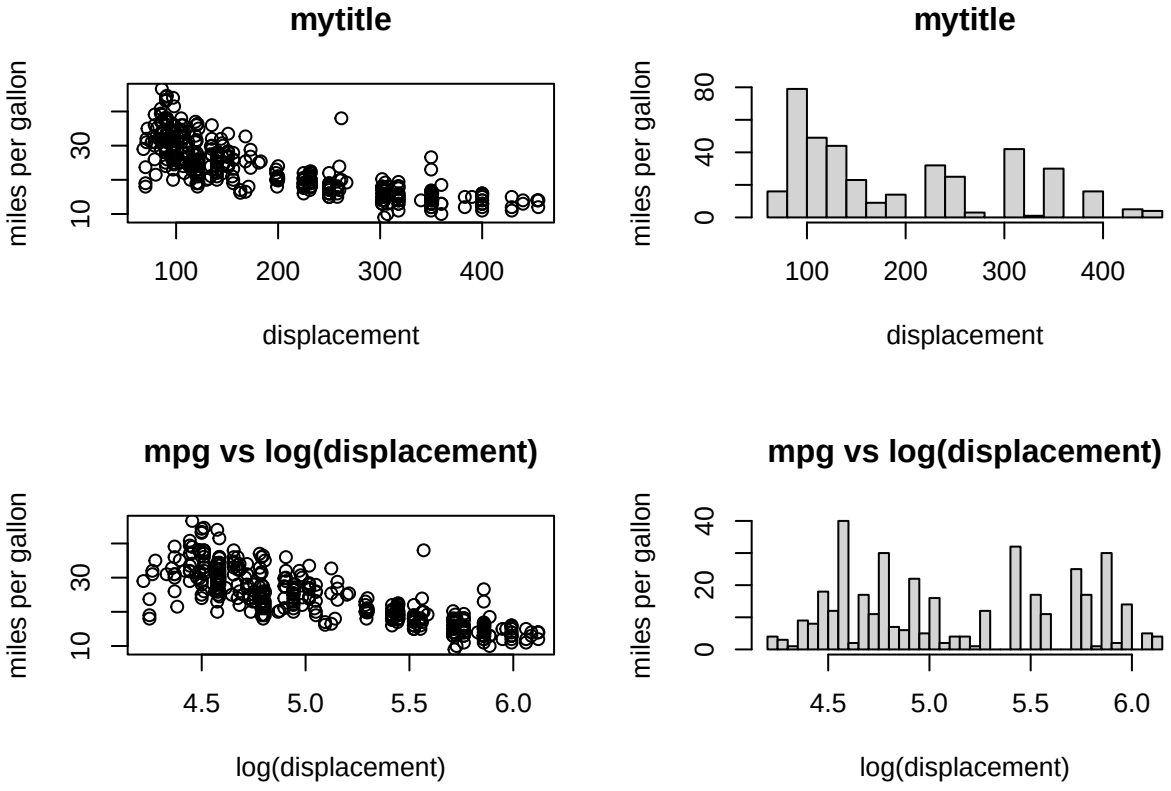
DELAYED ARGUMENT:

```
library("ISLR")
data(Auto)
mydata=Auto

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



4.0.3 Returning several values

You do not need to pass an argument to a function. You can return multiple values rather than one. The latter is usually performed using a list. Let's look at an example.

```
multi_return <- function() {


my_list <- list("color" = "red", "size" = 20, "shape" = "round")


return(my_list)
}

lout <- multi_return()

lout$size
```

```
## [1] 20
```

4.1 Exercise 3:

Write a function that *accepts* a vector of length evenly divisible by two (optional challenge: add a check for the argument) and *returns both* a matrix with two columns where the entries of the vector are entered by row AND the sum of the vector.

5 TOPIC 4: Recursive functions

Definition of recursive functions: functions that call themselves

5.1 The factorial function

Recall: $n! = 1 \times 2 \times 3 \times \ldots \times n$, where $0! = 1$. We can calculate the number of ways to select p objects from n (combination: n choose p) using the formula below:

$$\binom{n}{p} = \frac{n!}{p!(n-p)!}$$

- Implementation of the factorial function calculation via a FOR loop:

```
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)}   # 0! = 1  1! = 1
  temp=1  #initialize the variable that stores the product
  for (i in 2:x)temp=temp*i #iterate
  return(temp)
}

fact1(6)
```

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

```
factorial(6) # built in R
```

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

- Implementation of the factorial function calculation via a WHILE 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
   while(x>1){
     temp=temp*x
```

```
    x=x-1
    }
  return(temp)
  }

fact2(6)
```

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

- Implementation of the factorial function via a recursive function:

n! = n * (n-1)!

n x ((n-1) x (n-2) .....)

```
factr=function(x){


  # any recursive function is composed of two parts
  # part 1 : the treatment of terminal values

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

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

```
factr1=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*factr1(x-1))
   }


factr1(6)
```

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

```
factorial(6)
```

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

One can ALWAYS replace recursion by loops, but recursion is very elegant !

5.2 Exercise 4:

$S(n) = f(1) + f(2) + f(3) \ldots + f(n)$

For example, if I take $f(x) = \sin(x)$

$S(6) = sin(1) + sin(2) + sin(3) + sin(4) + sin(5) + sin(6)$

Why can this calculation be considered recursively?

Because the S function can be calculated by calling itself!

Indeed, try to express S as a function of S:

$S(6) = sin(6) + (sin(5) + sin(4) + sin(3) + sin(2) + sin(1)) = sin(6) + S(5)$

Write a recursive UDF that uses this idea to calculate S(n) for any value of n

```
#myS= #non-recursive function


#myS_rec =  #recursive function

# call your functions:
#n=9
#myS(n)
#myS_rec(n)
```