Lecture 7
R programming structure

GEOG 489 SPRING 2020

R Programming Structure

1) Global vs. local environment of a function

Global environment: higher level

Local environment: lower level

If an object is created within a function, that object will be *local* to that function.

```
g <- function(x)
{
    a=x^2
    return(a)
}
g(10)
print(a) # a was LOCAL to the function g, so no longer exists.
```

R Programming Structure

1) Global vs. local environment of a function

```
w <- 10   # Global variable
g <- function(x)
{
    w <- 20   # local variable
    return(x+w)
}
g(5) # Notice that the local variable takes precedence.
```

R Programming Structure

1) Global vs. local environment of a function

```
# Functions cannot modify variables at a level higher than it (usually).
w <- 10
g <- function(x)
{
    w <- 20
    return(x+w)
}
g(5)
w    # Notice that w, even though it was defined as 20 within the function, still is equal to 10, because the function g was unable to change the global environment.
```

R Programming Structure

1) Global vs. local environment of a function

Writing upstairs (write a variable to a higher level environment; use the superassignment operator <<-)

```
two <- function(u)
{
     u <<- 2*u
     z <- 2*z
 return(z)
}
x <- 1
z <- 3
u # Has not been assigned yet.
two(u=x)
```

R Programming Structure

1) Global vs. local environment of a function

Writing to Nonlocals with assign()

```
# Assign gives you finer control over writing variables up a level.
two <- function(u)
{
    assign("u",2*u, pos=.GlobalEnv)
}
x <- 1
two(x)
```

R Programming Structure

2) Recursion

A recursive function calls itself. This can be a very powerful solution to various problems.

The basic notion of a recursive function is:

For a problem you are trying to solve of type X:

1) Break the original problem of type X into one or more smaller problems of type X.
2) Within f(), call f() on each of the smaller problems.
3) Within f(), consolidate the results of step 2 to solve the original problem.

R Programming Structure

2) Recursion

Here is a basic example to solve a sorting problem, "Quicksort":

```
# Input: a vector of numbers.
# Output: the vector of numbers sorted from smallest to largest.

x <- c(5,4,12,13,3,8,88)
```

R Programming Structure

```
qs <- function(x)
{
      # If x is a one (or zero) element, return it.
      # Notice that this is not a trivial statement, this
      # is a termination condition.
      if(length(x) <= 1) return(x)

      pivot <- x[1]  # The first element of the vector.
      therest <- x[-1] # Every other element.

      sv1 <- therest[therest < pivot]    # Every element less than the pivot.
      sv2 <- therest[therest >= pivot]   # Every element greather than the pivot.

      sv1 <- qs(sv1) # Recursive, send all the less-than elements back to the function.
      sv2 <- qs(sv2) # Recursive, send all the greater-than or equal to elements back to the function.
      # Notice that if the recursion ends, it will return a single element "up the chain".

      return(c(sv1,pivot,sv2))
}
```

Quiz 3

Write a **recursive function** to calculate the factorial of a positive integer number. Factorial of a positive integer number is defined as the product of all the integers from 1 to that number.

Hint: For example, the factorial of 5 (denoted as 5!) will be 5! = 1*2*3*4*5 = 120. This problem of finding factorial of 5 can be broken down into a sub-problem of multiplying the factorial of 4 with 5 (namely 5! = 5*4!). More generally, n! = n*(n-1)!

The R file needs to be named:

LastName_FirstName_Quiz3.R

Please submit the quiz R file on Compass by the end of this class.