Introduction to R
Lecture 3

EEB C119/C219 (Winter 2012)

Christopher C. Strelioff

Overview

- Today
  - Functions
    - Introduction
    - Default arguments
    - Function **return**
    - Variable scope
  - Discrete logistic function
  - How do you use functions? Some live demos.
- Previous lectures
  - Variables and assignment
  - Vectore, Matrices
  - Plots
  - Workspace
  - Intro to scripts, **source** command
  - **for** loops, If else
  - Program flow & pseudocode

Function basics

Why use functions?

- Functions allow for the effective reuse of code
- Functions are the basic building block for building large simulations
- We have already seen useful (built-in) functions, ex:

```
> x <- 1:10
> (sumOfx <- sum(x))
[1] 55
```

  - `sum` is a function, the vector `x` is the argument we pass to the function
- We can define our own functions

Functions

Defining a function

- Elements of a function definition:
  - Give the function a name (be careful not to use reserved words like `if`, `for`, etc.)
  - Decide what arguments will be passed to function (`r`, `K`, etc.)
  - Decide what calculations (expressions) will be done by the function?
    - Ex: Calculate a vector of `n` values
  - Decide what to return from the function
    - Return the `n` vector? Or, just plot the results?

Functions

Defining a function

- Example format:

```
functionName <- function(arg1, arg2, ... )  {
  expression 1 ...
  expression 2 ...

  return(value)
}
```

- The explicit `return` expression is optional
- By default, result of last expression in function is returned
- I suggest always being explicit about return value

Functions

Example - divide numbers

```
> divideNumbers <- function(x,y)  {
+   # divide x by y
+   result <- x/y
+
+   # return result
+   return(result)
+ }
> # After definition, use function
> divideNumbers(1,5)

[1] 0.2

> # or
> divideNumbers(5,1)

[1] 5
```

- Note – order of arguments is important!

Functions

Example - default values for arguments

```
> divideNumbers <- function(x=1,y=4)  {
+   # divide x by y
+   result <- x/y
+
+   # return result
+   return(result)
+ }
> # No arguments, assume defaults
> divideNumbers()

[1] 0.25

> # Use as before
> divideNumbers(1,10)

[1] 0.1

> # Change ordering
> divideNumbers(y=10,x=1)

[1] 0.1
```

Functions

Variations on **return**

- A function can have more than one return
  - Useful to control program flow inside function
- A function can return more than one variable, vector, etc.
  - Use one function to calculate and return a bunch of things
- Let's do some examples . . .

Functions

Example - default values, plus multiple **returns**

```
> myOperations <- function(x=1,y=4,op="+")  {
+   # select the appropriate action
+   if (op == "+") {
+     return(x+y)
+   } else if (op == "-") {
+     return(x-y)
+   } else if (op == "*") {
+     return(x*y)
+   } else {
+     cat("\nUnrecognized operation!\n")
+   }
+
+ }#end function
> myOperations()

[1] 5

> myOperations(x=5,op="*")

[1] 20

> myOperations(x=5,op="/")

Unrecognized operation!
```

Functions

Example - **return** more than one result

```
> allOperations <- function(x,y)  {
+   # do all operations, return 'named' vector
+   result <- c("add"=x+y,
+               "subtract"=x-y,
+               "multiply"=x*y)
+
+   return(result)
+
+ }#end function
> temp <- allOperations(1,5); temp

     add subtract multiply
       6       -4        5

> names(temp)

[1] "add"      "subtract" "multiply"

> allOperations(5,1)

     add subtract multiply
       6        4        5
```

Functions

Variable **scope**

- Variables used inside function have their own workspace (memory)
    - Variables assigned inside function **can't** be seen outside function
    - However, variables assigned outside functions **can** be seen inside (**global** variables)
    - If both local (inside function) and global (outside function) variables with same name, local 'wins' inside function
- This behavior can result in some very confusing behavior and errors – careful!

Functions

Example - variable scope

```
> test <- function(x) {
+   # add one to x
+   y <- x+1
+   return(y)
+ }
> test(1)

[1] 2

> x

Error in try(x) : object 'x' not found

> y

Error in try(y) : object 'y' not found

> y <- 10
> test(1);y

[1] 2

[1] 10
```

Functions

Another example - variable scope

```
> test2 <- function(x) {
+   # add z to x, notice that z in not assigned
+   y <- x+z
+   return(y)
+ }

> test2(1)

Error in test2(1) : object 'z' not found
```

- Assign z as a global variable (this means outside of function)

```
> z <- 10
> test2(1)  # now, test2() works!

[1] 11
```

Demo 1

Repeat contents of last two slides in **rstudio**

- Ordering of commands in previous examples is important
- Also, show how to access functions you've written
    - This is a 'hands-on', interactive way to use a function
- Basic description of this method:
    - Write functions in text file, just like a script
    - **source** the script – this executes the definitions
    - Look at the workspace
    - Try the examples from the command line, in the order presented
- **script:** `Lecture03_Ex01.R`

Logistic model

A general function

```
> logisticModel <- function(n0, rd, K, timesteps)  {
+   # iterate model for desired number of timesteps
+   N <- rep(0,timesteps+1) # preallocate vector N (faster)
+   N[1] <- n0              # initialize first time point
+
+   # use for loop to iterate
+   for (t in 1:timesteps) {
+     N[t+1] <- N[t]*(1 + rd*(1 - N[t]/K))
+   }
+
+   # return vector
+   return(N)
+ }
> # After definition, use function
> (data <- logisticModel(10,1.2,500,5))

[1]  10.00000  21.76000  46.73561  97.57621 191.81699 333.69235

> (data2 <- logisticModel(10,2.83,500,5))

[1]  10.0000  37.7340 136.4622 417.2501 612.6752 221.9467
```

Demo 2

Using the logisticModel function

- Ex1 - Write function in text file and **source**, as before
  - **script:** `Lecture03_Ex02.R`
- Ex2 - Use function in **second** script
  - Use second script to call function many times
  - Plot dynamics for many `n0`, while holding other parameters fixed
  - **scripts:** `Lecture03_Ex02.R` and `Lecture03_Ex02a.R`