Functions and Loops

Ian Crandell

April 22nd, 2015

Outline

Review of Logic

```
if(condition){
        do if condition == TRUE
    }else{
        do if condition == FALSE
        }
```

- if() statements can compel R to do different things depending on whether or not some condition it met.
- condition can be either TRUE or FALSE. This is called a *boolean*. See code for examples.

What is a function?

- Most generally, a function is a procedure which transforms inputs into outputs.
- The inputs to a function in R are called the *arguments*. R functions can have zero, one, or many arguments.
- The output of the function is called its *returned value*. The returned value can be a number, matrix, text string, plot, or countless other objects.

Why use functions?

- Functions let you compactly execute a large block of code quickly and easily, saving time and space.
- This results in more compact and more readable code.
- Functions allow you to break complicated code into easily comprehensible parts.
- Functions are vital for fulfilling the Fundamental Law of Coding, which is...

The Fundamental Law of Coding

It is always better to do the same task using fewer lines of code.

Some Terminology

- An *object* is anything which can exist in R's memory
- Each object has a *class*, or classes, which describe what kind of object it is. For example, 4 is an object of class *numeric*. Other classes include *matrix*, *character*, *list*, or *histogram*.
- Some functions do different things depending on the class of the object given as its argument. These are called *methods*. A function will only work for a specific class if it has a method for that class.
- Example: the sum() function has methods for objects of class numeric, vector, matrix, and others, but not for objects of class character. These methods determine what exactly the function will do to the object its given.

Example: Summation

$$add = function(x, y)\{$$
$$z = x + y$$
$$return(z)$$
$$\}$$
$$add(3, 4)$$
$$7$$

- Here, x and y are the arguments of the function
- When the function is called, R will execute the code contained within the function.
- The value in return() is what the function will produce.
- particular numbers can be substituted for x and y and R will place those numbers in the function code wherever those variables occur.

Example: Functions with no Arguments

$$
\begin{aligned}
&say.hi = function()\{\\
&\qquad print(''E\ \ kassan!'')\\
&\quad\}
\end{aligned}
$$

- You can easily define functions that take no arguments by leaving the arguments section blank. You still need the parentheses, though.

Please see the script for more examples.

Multiple Outputs

$$
\begin{aligned}
&describe = function(vector)\{\\
&\qquad x1 = length(vector)\\
&\qquad x2 = mean(vector)\\
&\qquad x3 = median(vector)\\
&\qquad out = list(length = x1,\\
&\qquad mean = x2, median = x3)\\
&\qquad return(out)\\
&\quad \}
\end{aligned}
$$

- In general, R functions can only return one object.
- However, you can return multiple pieces of information by combining them into a list object.
- Here, length, mean, and median are the names of the objects inside of the list. We can extract them using the *$* operator.

Nested Functions

- It's possible to call functions from within other functions. This is called *nesting*.
- We've already see nesting in action. We called R' print() function from inside our say.hi() function. It's also possible to nest user defined functions. See script for examples.

Example: Multiplication Function

- In R, the * operator will multiply two numbers.
- However, it doesn't do classic matrix multiplication, rather, if A and B are matrices, A*B will just multiply the respective components of A and B. Proper matrix multiplication must be done with A %*% B.
- We will write a function which will examine its input to determine if it's a matrix or not. If it is, it will do matrix multiplication. If it isn't, it will do standard multiplication.
- This will give us a function with methods for matrices and for numbers.

Example: Pseudocode

```
$multiply = function(x, y)\{$
        $if$ (x and y are conformable matrices)$\{$
        $return(x\% * \%y)$
        $\}else\{$
        $return(x \times y)$
        $\}$
      $\}$
```

For loops

```
for(i in set){
Do code for each i in set
}
```

- For loops allow you to execute code multiple times easily.
- Each time the loop runs is called an *iteration* of the loop.
- i is called the *indexing variable* and will take on each value in *set*.
- The loop will iterate as many times as there are elements in set. At the first iteration, i will be the first value in set, the second value at the second iteration, etc.

Please see the script for more examples.

While loops

```
while(condition){
Do code as long as condition is true
}
```

- While loops continue to perform the encircled code for as long as the condition remains true
- This is useful for times when you can't specify for how long you want your code to run, which happens a lot with certain stochastic processes
- Be careful not to have a loop which will run forever.

Please see the script for more examples.

The Apply Family and apply()

apply(data, margin, function, ...)

- Apply family functions are used to apply a given function to multiple objects.
- The first function, apply(), will apply a function to either the rows (margin $= 1$) or columns (margin $= 2$) of a data table or matrix.
- The ... indicates that other arguments for function can be given to apply to be passed to the function.

tapply()

tapply(x, index, function, ...)

- tapply() is used apply a function across a vector x which is grouped according to the values in the indexing variable, or variables.

lapply(), vapply(), sapply()

lapply(x, function, ...)
sapply(x, function, ...)
vapply(x, function, returned type, ...)

- All these apply functions apply function to every element in x. The only differ in the value they return.
- lapply will return a list, sapply will try to figure out a good type to return, vapply will return what you specify.
- The primary usage of these functions is to replace loops. This has some advantages in terms of speed and usage in some specialty applications. We won't focus on these too much.