Introduction to R: Functions

In R, a function is an object that accepts input, runs some code on the input and typically returns some output based on the supplied input. We've already seen many functions that are built into R and its packages. Built-in functions and packages can take you a long way in R, but it can be useful to define your own functions to perform specific tasks outside the scope of built-in functions.

Creating Functions

Create a new function in R using the following syntax:

```r
new_function <- function(arguments) { # Assign the function and declare arguments

    for (x in 1:arguments){           # Body of the function (code to execute)
    print ("Study")
    }
}
```

After defining a function and assigning it a name, you can call the function using that name just like you would use a built-in function:

```r
new_function(5)               # Run the function defined above on the input 5
```

```
## [1] "Study"
## [1] "Study"
## [1] "Study"
## [1] "Study"
## [1] "Study"
```

This particular function does not return anything: it simply prints values to the console. Functions in R return the last expression evaluated by default:

```r
add_10 <- function(number){

    number + 10               # This function returns the argument + 10
}

add_10(5)
```

```
## [1] 15
```

You can also use return() to explicitly break out of a function and return the specified value:

```r
add_20 <- function(number){

    return (number + 20)       # Exit and return a specified value

    number + 10                # The function exits before this line is executed
}

add_20(5)
```

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

Any time you want to break out of a function early and return a value, you need to use return(). You can use return() even when returning the last evaluated expression, but it is not necessary since the last evaluated expression will be returned by default:

```
# These two functions are essentially identical:

add_10 <- function(number){
    number + 10
}

add_10_version2 <- function(number){
    return(number + 10)
}
```

Whether to explicitly use return() in cases like this where it is not strictly necessary is largely a matter of personal preference. In complicated functions, explicit use of return() may make code easier to understand. In short functions like the one above, return may just make code look more cluttered.

Function Arguments

A function can have one or more named arguments. You can assign a default value to an argument when creating a function with the argument_name = argument_value syntax:

```
sum_3_items <- function(x,y,z,                # Create a new function
                        print_args = TRUE){   # One argument has a default

    if (print_args){
        print (x)
        print (y)
        print (z)
    }

    return(x+y+z)
}

sum1 <- sum_3_items(1,2,3)              # Here the arguments are printed
```

```
## [1] 1
## [1] 2
## [1] 3
```

```
sum2 <- sum_3_items(10,20,30,
                    print_args = FALSE) # Changing the default suppresses printing
```

When you call a function, the arguments you supply are matched based on the order in which you supply them unless you specify the name of an argument explicitly. For instance, in the code above, the arguments 1, 2 and 3 passed to sum_3_items() are assigned to x, y and z, respectively. Alternatively, we could have explicitly assigned the arguments when calling the function:

```
sum1 <-  sum_3_items(z=1, y=2, x=3)
```

```
## [1] 3
## [1] 2
## [1] 1
```

When you explicitly assign values to arguments by name, the order you pass the arguments to the function doesn't matter. You can also assign some variables using ordered matching and some explicitly. Explicitly assigning a variable removes it from the normal argument order.

```
# y is explicitly assigned so 3 and 1 are assigned to the remaining args x and z
sum1 <-  sum_3_items(3, 1, y=2)
```

```
## [1] 3
## [1] 2
## [1] 1
```

R also provides a special ellipsis (. . .) argument. The . . . argument collects all extra arguments passed to a function that are not matched. The . . . argument can be used in functions where the number of arguments is not known in advance. For instance, we could use . . . to add any number of numerical arguments together:

```
addition_function <- function(...){         # Accept any number of arguments

    total <- 0                      # Create a variable to store the sum

    for (value in list(...)){       # list(...) extracts the arguments to a list
        total <- total+value        # Add each argument in ... to the total
    }

    total
}

addition_function(2,4,6,8,10,12,14)     # Add several numbers
```

```
## [1] 56
```

We've already encountered several functions that use . . . such as c() which combines any number of arguments into a single vector. . . . can also be used to pass arguments from one function to another, offering an easy way to extend existing functions:

```
# This function is the same as c() but it also prints the number of arguments you supply

extend_c <- function(...) {
  print(length(list(...)))          # Print the number of arguments
  c(...)                            # Pass the arguments to c() and return the result
}

my_vector <- extend_c(1,3,4,5)
```

```
## [1] 4
```

```
print(my_vector)
```

```
## [1] 1 3 4 5
```

Function Documentation

When writing a function that you or someone else is going to use in the future it can be useful to supply some documentation that explains how the function works. Documentation can be added to a function by inserting a few lines of comments immediately below the function's assignment statement. Documentation typically includes a short description of the function, a summary of the function's arguments and a description of the function's return value:

```r
root_mean_squared_error <- function(predicted, targets){
    # Computes root mean squared error between two vectors
    #
    # Args:
    #    predicted: a numeric vector of predictions
    #    targets: a numeric vector of target values for each prediction
    #
    # Returns:
    #    The root mean squared error between predicted values and targets

    sqrt(mean((targets-predicted)^2))
}
```

*Note: root mean squared error (rmse) is a common evaluation metric in predictive modeling. Documentation should provide enough information that the user doesn't have to read the code in the body of the function to use the function. There are many other intricacies of functions that we haven't discussed here, but the basics described in this lesson are sufficient for our purposes. reference: http://hamelg.blogspot.com/