Creating your own R functions

An important feature in R is the ability for you to build your own functions. This extensibility allows you to automate repetitive tasks or run a series of commands with different input for each repetition. The basic form for an R function is:

```
myFuctionName = function( ) {
      >MY CODE GOES HERE<
}
```

Once you run the function declaration, it is available for your use during your current R session. This behavior is slightly different than the behavior of functions in R packages. Once an R package is loaded, you have immediate access to all of the package function. Your functions are only available once you run the function declaration, and they are only available in your current session. RStudio enables you to override this behavior. You can save your R session when you close RStudio and it will resume that session, with its entire environment [including your functions] available, when you restart RStudio.

Here is a function that creates a vector of integers from 1 to n:

```
integerVector = function(n) {
      v = c(1:n)
      v
}
```

The name of the new function is **`integerVector()`**. This function has one numeric parameter **`n`** that is used in the function to set the upper limit of the integer sequence in the output vector. The first line of code builds the vector using the R function `c()`. The final line outputs the resulting vector. This final line is necessary if you want to output the results of the code inside of your function for use outside the function.

That last sentence is an important concept to understand. All of the data objects that are inside your function are not accessible outside your function. The data object **`v`** in **`integerVector()`** only exists while **`integerVector()`** executes. If you want to use the result of the computation in your function, you must output that data. The last line of **`integerVector()`** does this.

Once you execute this declaration, you can use `integerVector()` in your R session. The function call `integerVector(10)` will result in the output

```
1 2 3 4 5 6 7 8 9 10
```

You can save the output of `integerVector()` by simply assigning its output to an R object like this:

```
x = integerVector(10)
```

Now the object x contains the vector `1 2 3 4 5 6 7 8 9 10`

It is important to remember to use unique names for your functions. If you accidentally choose a function name that matches the name of existing R function, your function will override the existing function and only your function will be active in your current session. While you may want to compute a result differently than the existing R function, it is still better to use your own unique

function name. This naming scheme will ensure that you have access to both functions, not simply your custom function.

Here is a more complex function:

```
select10 = function(n) {
      v = c()
      for(i in 1:n) {
            v = rbind(v, sample(1:10, size = 1))
      }
      v
}
```

The function `select10()` above creates a list of n random numbers in the range of 1 to 10. The function creates an empty list v. It then repeats the random selection of a number between 1 and 10 and adds it to the list. Once the list is complete, `select10()` outputs the final list.

You can modify `select10()` so that it creates a list of **`size`** numbers between **`mini`** and **`maxi`**. [We will avoid using `min()` and `max()` since they are standard R function names]. The new function might look like this:

```
selectAny = function(size, mini, maxi) {
      v = c()
      for(i in 1:size) {
            v = rbind(v, sample(mini:maxi, size = 1))
      }
      v
}
```

`selectAny()` does the same computational steps as **`select10()`** but the function arguments of **`selectAny()`** provide an interface to designate the size of the list and the minimum and maximum values. Both of these functions use a **`for()`** loop to repeat an operation and you can control the number or repeated operations with a function argument, as shown in **`selectAny()`**.

Functions can also be used to simulate tasks. Here is an example:

```
# data from a pair of 6-sided dice roll function
dice6Data = function(x) {
  d1 = c() # a numeric vector
  d2 = c() # a numeric vector
  ds = c() # a numeric vector
  # roll the dice x times
  for (i in 1:x) {
```

```
    d1 = rbind(d1, sample(1:6, size = 1))
    d2 = rbind(d2, sample(1:6, size = 1))
  }
  ds = d1 + d2
  v =cbind(d1, d2, ds)
  v = as.data.frame(v)
  names(v) = c("die1", "die2", "sum")
  v # output the results
}
```

`dice6Data()` simulates the roll of a pair of 6-sided dice. It outputs a data frame that contains the value of each die and the sum of the two dice. The input argument identifies how many times you want to roll the dice. If you use 10 as your input argument, **`dice6Data()`** will return a data frame with 10 rows, one row in the data frame for each dice roll. **`dice6Data()`** can help you simulate the outcomes of a large number of dice rolls.

Because functions help you automate tasks, you can develop functions to compute anything that is not already included in an R package. An example of this would be the mode statistical metric. In descriptive statistics, the mode is the most frequent value in a set of values. This statistic is normally used to describe discrete [integer] sets of values. R does not have a standard function to calculate mode. The **`mode()`** function built into R returns the data type of a data object, not its statistical mode. You can remedy this by writing your own function. Here is an example:

```
# statistical mode function - R mode() does something else
statMode = function(v) {
  result = as.numeric(names( which(table(v) == max(table(v)))))
  return(result)
}
```

When given a data object containing a vector of integers, **`statMode()`** will return the most frequent value or values [if there is a tie]. Notice that `statMode()` uses the command to return the result of the function's computation. This command **`return()`** is another way to output the results of your function.

So, in summary, user defined functions in R allow you to repeat operations by repetitively calling your function, help you simulate a process for analysis, and to automate a functionality not normally included in R or any of its packages.