Week 8 Module 3: Write Your Own Functions

CSCI E-5a: Programming in R

Let's clear the global computing environment:

```r
rm( list = ls() )
```

Run this code to load in the R objects for this module:

```r
# load( "Module 3 R Objects.Rdata" )

ls()
```

```
## character(0)
```

Module Overview and Learning Objectives

Hello! And welcome to Module 3: Write Your Own Functions.

In this module, we're going to learn how to write our own functions.

- In Section 1, we'll learn the basics of how to create our own user-defined functions.
- In Section 2, we'll explore the `return()` statement.
- In Section 3, we'll see how to define default values for our parameters.
- In Section 4, we'll learn about global and local environments.
- In Section 5, we'll reflect on writing generalizable code.

Once you've completed this module, you should be able to:

- Define your own functions, including specifying default values for parameters.
- Explain the difference between the global and local environments.

There are two new built-in R functions in this module:

- `function()`
- `return()`

All right! Let's get started by learning how to write our own functions.

Section 1: Creating User-Defined Functions

Main Idea: *We can write our own functions*

In this section, we'll learn the basics of how to create our own user-defined functions.

Creating user-defined functions is one of the most important techniques in all of programming, and every programming language supports this feature, so it can reasonably called "universal".

We create our own functions by constructing a function object and then associating it with a variable name by using the assignment operation:

- First, we specify the name of our user-defined function.
- Next, we have an assignment operator.
- Finally, on the right-hand side of the assignment operator we have a function object, which has a number of components:
  - The first component of the function object is the keyword `function`.
  - Next is an open parenthesis.
  - Then we have the sequence of the names of the input arguments.
  - Then there is a closing parenthesis.
  - Then there is an open curly brace, which contains the *body* of the function.
  - Finally an closing curly brace ends the body of the function.

Here's an example of a user-defined function:

```
double.the.input <-

    function( x ) {

        double.x <- 2 * x

        return( double.x )

    }
```

Let's see how this works:

- First we have the name that we want for our function, which in this case is "double.the.input".
- Next we have an assignment operation.
- Next we have the keyword `function`.
- Next an opening parenthesis, followed by "x", which is the name of the input argument, followed by the closing parenthesis.
- Finally, we have the opening curly brace, then the body of the loop, and then the closing curly brace that ends the loop body.

Once we've defined the `double.the.input()` function, we can use it just as though it were a built-in function:

```
double.the.input( x = 3.5 )
```

```
## [1] 7
```

What happens when R executes this statement?

- First, R associates the value 3.5 with the variable `x`.
- Next, it enters the function body:
  - The expression `2 * x` is evaluated, which give $2 \times 3.5 = 7$, and this value is assigned to the variable `double.x`.
  - The function then returns the value of `double.x`, so that the function returns the value 7.

The important thing to remember is that once R enters the body of the function, it's just executing statements like it usually does.

A function thus is a way for us to bundle a sequence of statements together.

The function can contain all the operations that we've seen so far: variable creating and assignment, `for` loops and `if` statements, vectorized operations, even other function definitions.

By the way, we could have written our function in a more compressed manner:

```
double.the.input <-
    function( x ) {

        return( 2 * x )

    }
```

So that's how to create our own user-defined functions.

Now let's learn how to return values from a function.

Section 2: The `return()` Statement

> **Main Idea:** *We can return values from a function*

In this section, we'll explore the `return()` statement.

One important aspect of the function definition is the `return()` statement.

Usually, when we call a function, we want it to calculate a value and return it to us.

The `return()` statement indicates that R should return the value, and exit from the function.

If you don't have a `return()` statement in your code, then R will automatically return the last value that was calculated. So we could written just this:

```
double.the.input <-
    function( x ) {

        2 * x

    }
```

Let's see this at work:

```
double.the.input( x = 3.5 )
```

```
## [1] 7
```

Personally, I think the best practice is to always use the `return` statement to explicitly indicate when the function is returning a value.

So that's how to return values from a function.

Now let's see how to define default values for input arguments.

Section 3: Setting Default Values

> **Main Idea:** *We can define default values for input arguments*

In this section, we'll see how to define default values for our parameters.

We set default values by using a single equals sign and the default value.

Let's create a function called `multiplier()` that takes an input value x and an optional multiplier value k, and returns the product of x times k.

If no value for k is specified, the function will use a default value of 2:

```
multiplier <-
    function(x, k = 2) {

        x.times.k <- x * k

        return( x.times.k )
    }
```

If we don't specify a value for k, then the function will use the default value of $k = 2$:

```
multiplier( x = 7 )
```

```
## [1] 14
```

If we don't want the default value we can always explicitly override it:

```
multiplier( x = 3, k = 4)
```

```
## [1] 12
```

So that's how we can define default values for input parameters.

Now let's learn about local and global computing environments.

Section 4: Local and Global Environments

Main Idea: *Functions have their own local environments*

In this section, we'll learn about global and local environments.

Let's explore a little puzzle.

Let's assign the value 3 to the variable `x`:

```r
x <- 3
```

Next, we'll define a strange little function:

- This function takes one input argument, denoted x.
- The function body executes two actions:
  - First, it assigns the value 5 to x.
  - Then it prints out the value of x, which is 5.

I admit, this is a strange function:

```r
strange.function <- function( x ) {

    x <-5

    cat( "x =", x, "\n")

}
```

Next, we run our `strange.function()`:

```r
strange.function( x = 3 )
```

```
## x = 5
```

Remember how we defined `strange.function()`: it took an input argument, automatically assigned the value 5 to it, and then printed out the value. So this looks good.

What happens if we now evaluate the variable `x`?

```r
cat( "x =", x)
```

```
## x = 3
```

What's going on here? I thought when we ran `strange.function()` it assigned the value 5 to `x`. Why is it still 3?

- When we first assigned the value 3 to the variable `x`, we did this in what is called the *global environment.*
- The global environment is the top-level collection of variable assignments in R.

- Since the variable `x` is in the global environment, it is called a "global variable".
- When R calls a function, it creates a special new private environment just for that function, and the input arguments are part of that private environment.
- So, when we call `strange.function()`, it creates a private environment in which the name `x` is associated with the value 3.
- Then the first line of the function body assigns the value 5 to the object `x`. This happens within the private environment of the function call, and the global environment remains unchanged.
- The function then prints out the `cat()` statement and exits.
- When the function exits, the private environment is deleted, and R reverts to using the global environment to look up the values for variables.
- So, when we evaluate `x` after calling `strange.function(x)`, it is evaluated using the global environment, and `x` has the value 3.

The main point here is that the function has its own private environment, and any variable associations made within that private environment are lost when the function exits.

Thus, the variables in a function are *local* variables, in the sense that they only apply to the body of the function.

It *is* possible to destructively modify variables in the global environment from within a function, but we have to use a special assignment operator denoted `<<-`. Do you see the difference? This assignment operator has two less-than signs instead of just one.

Now let's use this special assignment operation to destructively modify the variable `x` in the global environment:

```
destructive.function <- function( x ) {
    x <<- 5
}
```

Let's try it out:

```
x <- 3

destructive.function( x )

x
```

```
## [1] 5
```

Most programmers agree that it's good to avoid making these sorts of global assignments from within a function.

It can be very difficult to reason about a program if global variables are being modified by function calls.

So that's how local and global environments operate.

Now let's think about copying and pasting code.

Section 5: Copy and Paste

Main Idea: *Write generalizable code*

In this section, we'll reflect on writing generalizable code.

I often point out to you that you can copy and paste your code.

If you've written your code properly, defining variables at the beginning and then always using them for the rest of the computation, then it's easy to copy and paste this and modify it.

To do this, you have to be able to design the computation in a general way, instead of just for your specific situation.

But that's exactly what writing a function is – generalizing a computation so that it can take many different possible inputs.

So if you've been writing code that's been easy to copy and paste you should have a lot of practice already in designing general code.

If you look back in much of our previous work you'll see that I always try to define the input variables for a computation first, and then everything else uses them, so my code examples are intended as examples of how to write this sort of generalized procedure.

Being able to generalize a computation is essential for creating user-defined functions, so that was the point of encouraging you to write your code so that it was easy to copy and paste.

So that's why I emphasized copying and pasting code.

Now let's review what we've learned in this module.

Module Review

In this module, we learned how to write our own functions.

- In Section 1, we learned the basics of how to create our own user-defined functions.
- In Section 2, we explored the `return()` statement.
- In Section 3, we saw how to define default values for our parameters.
- In Section 4, we learned about global and local environments.
- In Section 5, we reflected on writing generalizable code.

Now that you've completed this module, you should be able to:

- Define your own functions, including specifying default values for parameters.
- Explain the difference between the global and local environments.

There were two new built-in R functions in this module:

- `function()`
- `return()`

All right! That's it for Module 3: Write Your Own Functions.

Now let's move on to Module 4: Examples.