LAB100

Week 11: Writing Functions.

Your aim for this workshop is to learn how to create your own functions using the `function` command. Upon completing this workshop you should be able to:

- Understand how to create and use simple functions
- Replicate existing RStudio commands
- Create code for a mathematical function
- Generalise existing code by constructing a function

Remember that you need to change the working directory and open a new R script at the start of each workshop.

1 The `function` Command

Suppose that you need to use a command in RStudio for your current project to calculate something that you know how to evaluate mathematically. However, after searching through the help files and online resources you cannot find what you need. For example, say that you want to find the prime factors of 864. It is not difficult to evaluate this by hand to be $2^5 \times 3^3$, but after searching you find that RStudio does not have the command that performs this operation.

The command `function` allows you to define your own functions as long as you know how to evaluate them. Below is the general `function` structure:

```
<FN.NAME> <- function( <INPUT> ) {
  <COMMANDS>
  return( <OUTPUT> )
}
```

It begins with defining the name of your function (`<FN.NAME>`) that you will call when you are wanting to use it. This is followed by the command `function` and the `<INPUT>` arguments that you specify in order for your function to work are placed in rounded brackets. The body of your function, contained within curly brackets, will contain all of the `<COMMANDS>` that takes your input values and evaluates your expression. Finally, `return` ensure that only the `<OUTPUT>` values are exported from your command. Using your new function is just as simple as using any in-built RStudio command; type the function name and any input arguments in parenthesis:

```
<FN.NAME>( <INPUT> )
```

To illustrate, suppose that we want a function that computes the square of a number x. Let us name this function `square` that only takes the number `x` as an input. Then we can write this new command as follows:

```
square <- function( x ){
  output <- x*x
  return(output)
}
```

Test your command with a few examples and check that it obtains the same answers as using `^`:

```
square( 10 )                              10^2
square( -3 )                              (-3)^2
square( 0 )                               0^2
square( 0.5 )                             0.5^2
```

2 Replicating an Existing Command

So far in this course you have learnt how to use many RStudio commands, but do you know what operations the computer is actually performing in order to evaluate your code? All of these commands have been written by a programmer that use `function` to collate a set of code that performs a specific task. The best way to appreciate what a computer does when evaluating any command is to create our own version using `function`.

The `abs` command evaluates the absolute value of an number that is supplied. One way of understanding how this command operates is to state the mathematical function that it evaluates:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise.} \end{cases}$$

Let us recreate this function with a command called `myabsolute` that accepts a single real number that we define as `x`. The body of the function should begin with assessing whether the input value is less than zero. If true, then this number should be negated, else the number is positive or zero and should not be changed. Finally, the command should only return the single non-negative output value. The following set of code illustrates this sequence of operations within a single function:

```
myabsolute <- function (x){
    if(x<0){
        output <- -1*x
    }else{
        output <- x
    }
    return( output )
}
```

Check that `myabsolute` calculates the absolute value for various input values and that it returns the same result as `abs`.

```
myabsolute(2)                              abs(2)
myabsolute(-2)                             abs(-2)
myabsolute(0)                              abs(0)
```

3 Multiple Input Arguments

Suppose that you have a mathematical function that you need to evaluate a number of times under different conditions. Rather than copying and editing a set of code for every instance, you can create a single command that accepts multiple input values and evaluates this function wherever it is requested. This is particularly useful in simplifying your code such that it is more readable and therefore identifying mistakes become an easier task. For example, the general definition of a quadratic function is defined by:

$$g(x, a, b, c) = ax^2 + bx + c$$

This is a function that has four input values, the variable x and the constant co-efficients a, b and c. Creating a command in RStudio that evaluates this expression and accepts multiple arguments can be done with `function`. In the rounded brackets after `function`, simply type the name of the variable with each being separated with a comma. The code to create the above example would be:

```
g <- function(x, a, b, c){
    output <- a*x^2 + b*x + c
    return(output)
}
```

Once you have ran the above code in your console we can use this function to evaluate any quadratic function. For example, to evaluate $x^2 - 2x + 3$ when $x = 3$ we can execute:

```
g(x=3, a=1, b=-2, c=3)
```

In this example we have defined the input arguments explicitly. As long as you are using the names of the input arguments, the order is not important:

```
g(x=3, a=1, c=3, b=-2)
g(b=-2, c=3, a=1, x=3)
```

Alternatively, the following example would produce the same answer:

```
g(3, 1, -2, 3)
```

If the argument names are not specified, then the order of which you type the numbers into the command is important. In this case, RStudio allocates each number to the respective argument according to the order you specified when you created the function. Therefore, `3` is assigned to the first argument (`x`), `1` is assigned to the second argument (`a`), etc.

3.1 Vector inputs

Functions in R do not necessarily have to take scalar inputs. For instance, we may want to evaluate a quadratic function for a range of value of x. For instance, we can create a sequence of values between 0 and 10 and then compute

$$g(x) = x^2 + 3x - 2$$

at each value

```
x <- seq(0,10,by=0.1)
g(x, 1,3,-2)
```

this returns a vector of the same length as `x`. Potentially, we can also allow some of all the other inputs of `g` to be vectors as well. For instance

```
u <- g(x=c(1,2),a=c(1,3),b=1,c=c(-1,1))
```

will return a vector where the first element is $g(1)$ for $g(x) = x^2 + x - 1$ and the second element is $g(2)$ where $g(x) = 3x^2 + x + 1$. Note that, if one of the inputs is a scalar, it is assumed to apply to all the elements.

If a function can only take a scalar input, it can often be adapted to allow a vector input. For instance, consider again the `myabsolute` function defined in Section 2. Currently if we attempt to supply a vector to this function we will get a warning message and not necessarily the expected answer.

```
myabsolute(c(-2,2))
```

To obtain the absolute value of a number we need to multiply by (-1) if it is negative and 1 otherwise. This can be achieved by using logical statements:

```
myabsolute <- function (x){
  return( (-1)^(x<0) * x )
}
```

```
myabsolute(c(-2,2))
```

4 Functions for recursion relations

Functions provide a convenient way of generalising code. For instance, recall the `for` loop code to generate the Fibonacci sequence considered in Week 9:

```
MAX <- 5
x <- rep( 1, MAX )
for( n in 3:MAX){
   x[n] <- x[n-1] + x[n-2]
}
x
```

The Fibonacci sequence is a special case of a second order recurrence relation defined through

$$a_n = a_{n-1} + a_{n-2}$$

for $n > 2$ with initial conditions $a_1 = a_2 = 1$
More generally we could define a sequence

$$a_n = \alpha a_{n-1} + \beta a_{n-2}$$

for $n > 2$ with initial conditions $a_1 = u, a_2 = v$
We can write a function to find the first N terms of such a sequence:

```
recursion <- function(A,B,u,v,N) {
  x <- c(u,v,rep( 1, N -2))
  for( n in 3:N){
    x[n] <- A*x[n-1] + B*x[n-2]
  }
  return(x)
}
```

here `A` and `B` represent α and β respectively, while `u` and `v` represent a_1 and a_2. For instance,

```
fib <- recursion(1,1,1,1,100)
```

produces a vector `fib` consisting of the first 100 terms of the Fibonacci sequence.

Quiz 1: Reading a `function` command

The following code creates a function called `f` that accepts a single input value `x`.

```
f <- function(x){
    a <- x+4
    b <- a^2
    c <- b-32
    return(c)
}
```

What does this function evaluate?

(**A**) $f(x) = (x+4)^2 + 32$

(**B**) $f(x) = x^2 - 8x - 16$

(**C**) $f(x) = (x^2 + 4) - 32$

(**D**) $f(x) = x^2 - 32$

(**E**) $f(x) = (x+4)^2 - 32$

Quiz 2: Vectorised function

Consider the following function

```
f <- function(x,y){
if (x <y) {
       out <- y
    }else {
       out <- x
    }
    return(out)
}
```

Which of the following performs the same operation as `f` to each pair of components of vectors `x,y`.

(A) `g <- function(x,y) return( max(x,y) )`

(B) `g <- function(x,y) return( x*(x >=y) + y*(x<y) )`

(C) `g <- function(x,y) return( y*(x >=y) + x*(x<y) )`

(D) `g <- function(x,y) return( x*(max(x,y)==x) + y*(max(x,y)==y) )`

(E) `g <- function(x,y) return( x*(max(x,y)==x) + y*(max(x,y)!=x) )`

Quiz 3: Recursion relationship I

A sequence a_n is defined by the recursion relationship

$$a_n = \frac{3}{2}a_{n-1} - a_{n-2}$$

where $a_1 = 1, a_2 = 1/2$
Use the `recursion` function defined in Section 3 to determine the number of a_n satisfying

$$|a_n| < \frac{1}{2}$$

and $n \leq 100$.

Quiz 4: Recursion relationship II

A sequence b_n is defined by the inhomogeneous recursion relationship

$$b_n = \frac{3}{2}b_{n-1} - b_{n-2} + k$$

where $b_1 = 1, b_2 = 1/2$
Adapt the `recursion` function to compute the first N values of an inhomogeneous recursion relationship and hence find the value of k for which $b_{100} = 0$ (or very close to 0).

(**A**) $k = 0.3319$

(**B**) $k = -0.1323$

(**C**) $k = 1.006$

(**D**) $k = -1.006$

(**E**) $k = 0.4561$

Quiz 5: Recursion relationship III

Let $\beta_m = \frac{m}{50}$ for $m = 1, \dots, 50.$ For each m define a sequence via the recursion relationship

$$d_{n,m} = \beta_m d_{n-1,m} - \beta_m d_{n-2,m} - \beta_m$$

and let $d_{1,m} = 1, d_{2,m} = 1/2.$
For which m does the sequence $(d_{n,m})_{n=1}^{n=100}$ contain exactly 5 terms satisfying $d_{n,m} < -0.5.$
[**Hint**: Construct a `for` loop and call your modified recursion function from within it.]
