18.4: Writing Functions

In this section I want to talk about functions again. Functions were introduced in Section 3.5, but you've learned a lot about R since then, so we can talk about them in more detail. In particular, I want to show you how to create your own. To stick with the same basic framework that I used to describe loops and conditionals, here's the syntax that you use to create a function:

```
FNAME <- function ( ARG1, ARG2, ETC ) {
       STATEMENT1
       STATEMENT2
       ETC
       return( VALUE )
     }
```

What this does is create a function with the name FNAME, which has arguments ARG1, ARG2 and so forth. Whenever the function is called, R executes the statements in the curly braces, and then outputs the contents of VALUE to the user. Note, however, that R does not execute the commands inside the function in the workspace. Instead, what it does is create a temporary local environment: all the internal statements in the body of the function are executed there, so they remain invisible to the user. Only the final results in the VALUE are returned to the workspace.

To give a simple example of this, let's create a function called `quadruple()` which multiplies its inputs by four. In keeping with the approach taken in the rest of the chapter, I'll use a script to do this:

```
## --- functionexample.R
quadruple <- function(x) {
  y <- x*4
  return(y)
}
```

When we run this script, as follows

```
source( "./rbook-master/scripts/functionexample.R" )
```

nothing appears to have happened, but there is a new object created in the workspace called `quadruple`. Not surprisingly, if we ask R to tell us what kind of object it is, it tells us that it is a function:

```
class( quadruple )
```

```
## [1] "function"
```

And now that we've created the `quadruple()` function, we can call it just like any other function And if I want to store the output as a variable, I can do this:

```
my.var <- quadruple(10)
print(my.var)
```

```
## [1] 40
```

An important thing to recognise here is that the two internal variables that the `quadruple()` function makes use of, `x` and `y`, stay internal. That is, if we inspect the contents of the workspace,

```
library(lsr)
```

```
## Warning: package 'lsr' was built under R version 3.5.2
```

```
who()
```

```
##    -- Name --           -- Class --   -- Size --
##    balance              numeric       1
##    day                  character     1
##    i                    integer       1
##    interest             numeric       1
##    itng.table           table         3 x 4
##    month                numeric       1
##    monthly.multiplier   numeric       1
##    msg                  character     1
##    my.var               numeric       1
##    payments             numeric       1
##    quadruple            function
##    speaker              character     10
##    today                Date          1
##    total.paid           numeric       1
##    utterance            character     10
##    w                    character     1
##    W                    character     1
##    w.length             integer       1
##    words                character     7
##    x                    numeric       1
```

we see everything in our workspace from this chapter including the `quadruple()` function itself, as well as the `my.var` variable that we just created.

Now that we know how to create our own functions in R, it's probably a good idea to talk a little more about some of the other properties of functions that I've been glossing over. To start with, let's take this opportunity to type the name of the function at the command line without the parentheses:

```
quadruple
```

```
## function (x)
## {
##     y <- x * 4
##     return(y)
## }
```

As you can see, when you type the name of a function at the command line, R prints out the underlying source code that we used to define the function in the first place. In the case of the `quadruple()` function, this is quite helpful to us – we can read this code and actually see what the function does. For other functions, this is less helpful, as we saw back in Section 3.5 when we tried typing `citation` rather than `citation()` .

18.4.1 Function arguments revisited

Okay, now that we are starting to get a sense for how functions are constructed, let's have a look at two, slightly more complicated functions that I've created. The source code for these functions is contained within the `functionexample2.R` and `functionexample3.R` scripts. Let's start by looking at the first one:

```
## --- functionexample2.R
pow <- function( x, y = 1) {
  out <- x^y  # raise x to the power y
  return( out )
}
```

and if we type `source("functionexample2.R")` to load the `pow()` function into our workspace, then we can make use of it. As you can see from looking at the code for this function, it has two arguments `x` and `y`, and all it does is raise `x` to the power of `y`. For instance, this command

```
pow(x=3, y=2)
```

```
## [1] 9
```

calculates the value of 3^2. The interesting thing about this function isn't what it does, since R already has has perfectly good mechanisms for calculating powers. Rather, notice that when I defined the function, I specified `y=1` when listing the arguments? That's the default value for `y`. So if we enter a command without specifying a value for `y`, then the function assumes that we want `y=1`:

```
pow( x=3 )
```

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

However, since I didn't specify any default value for `x` when I defined the `pow()` function, we always need to input a value for `x`. If we don't R will spit out an error message.

So now you know how to specify default values for an argument. The other thing I should point out while I'm on this topic is the use of the `...` argument. The `...` argument is a special construct in R which is only used within functions. It is used as a way of matching against multiple user inputs: in other words, `...` is used as a mechanism to allow the user to enter as many inputs as they like. I won't talk at all about the low-level details of how this works at all, but I will show you a simple example of a function that makes use of it. To that end, consider the following script:

```
## --- functionexample3.R
doubleMax <- function( ... ) {
  max.val <- max( ... )   # find the largest value in ...
  out <- 2 * max.val      # double it
  return( out )
}
```

When we type `source("functionexample3.R")`, R creates the `doubleMax()` function. You can type in as many inputs as you like. The `doubleMax()` function identifies the largest value in the inputs, by passing all the user inputs to the `max()` function, and then doubles it. For example:

```
doubleMax( 1,2,5 )
```

```
## [1] 10
```

18.4.2 There's more to functions than this

There's a lot of other details to functions that I've hidden in my description in this chapter. Experienced programmers will wonder exactly how the "scoping rules" work in R,[137] or want to know how to use a function to create variables in other environments[138],

or if function objects can be assigned as elements of a list[139] and probably hundreds of other things besides. However, I don't want to have this discussion get too cluttered with details, so I think it's best – at least for the purposes of the current book – to stop here.
