Post1: Recursive Functions

Jackson Leisure

10/31/2017

A Recursive Take on Functions

Contents

Introduction

As we saw in lecture and lab, R allows users to implement their own functions, which allows for greater customization and improved control over data. Unfortunately, our interactions with functions have been relatively basic up to this point, focusing mostly on quite simple functions (such as `square <- function(x) {x * x}`) or functions which pull heavily on previously implemented formulas (such as Gaussian or Binomial functions from lab). Additionally, we learned about various loops, if/else statements, and nested functions. Using these topics, in addition to the concepts about how to document functions, will allow us to expand beyond the basics we learned in class and create more interesting functions.

Review of Functions

Here is a quick review on important concepts for functions in R.

As shown in lecture, this is the general form of a function in R:

```
# anatomy of a function
some_name <- function(arguments) {
  # body of the function
}
```

Just like the built in functions we have access to (`c()`, `plot()`, `mean()`, etc.), a user-defined function takes in inputs and performs some number of operations. Functions are defined by assigning 'function()' to the desired name and filling in the body of the function with the commands to be run when the function is called. Any arguments defined when creating this function will represent what happens to the inputs that are given when the function is called further in the code.

The idea behind user-defined functions is that they allow for commands which are repeated multiple times to be assigned a sort of 'code name', which will execute the same series of commands each time it is called. Additionally, functions improve readability of code, as (when they are well documented), functions will help those who use the code to understand exactly what is supposed to occur in each section without being bogged down by too many lines of code.

Abstraction & Modularization

This leads me to a brief mention of the terms "abstraction" and "modularization". Abstraction, as described on Wikipedia is, in essence, "the elimination of the irrelevant and the amplification of the essential." This, in layman's terms is "how to present only what is relevant" in our code. During most of our labs and homeworks, we leave all of our code out in the open (for grading purposes, obviously), but if we had the desire to generate beautiful graphs and apps in R without showing how we got them, we must learn to use abstraction. The first step of abstraction in R would be to change `echo = TRUE` to `echo = FALSE`. This will immediately hide the code chunks from the final product. However, this might not be satisfying enough, as a quick look at the file which generated the knitted document could be quite cluttered and lacking in readability.

This, in turn, leads to the next term I mentioned: modularization. Again, I provide a succinct definition from Wikipedia, stating that modularization "emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality." This definition can be interpreted simply as "use functions to simplify code." The use of functions allows sections of code that are repeated often to be condensed into a simplified object which can be used to reduce the number of lines (and the amount of confusion) present in code. As an example, it is much more difficult to understand the meaning of

```
x <- 8
result <- 1
for (i in 1:x) {
    result <- result * i
}
result
```

```
## [1] 40320
```

than it is to understand the meaning of

```
factorial(8)
```

```
## [1] 40320
```

Finally, these two concepts can be further combined, by creating multiple R files, each with various functions within them, and importing functions between them, allowing code to be written in a much more simplified and concise manner. This allows the user to create more intricate programs, since, as more files are created, the user can pull from more available functions, instead of wasting time rewriting the same or similar functions each time. Eventually, when enough related functions have been written, the user might think about collecting this data into a single

source, and exporting it as a package for others to make use of. The beauty of abstraction and modularization is what allows us to use packages such as ggplot2 or readr without knowing exactly what is occurring inside of each function; instead, we can just assume that they will work when provided with proper inputs and keep on coding!

Conditionals, Loops, & Nested Functions

Now that I have addressed why it is important to make use of abstraction and modularization, I will continue to address the benefit of functions and what useful structures we have learned.

On their own, simple functions are not exceedingly useful, and are merely rebrandings for a few lines of code. When paired with important structures (such as conditionals and loops), functions begin to really come to life. Not only can we directly evaluate, based on the input (like in the case of an exponent function), we can now choose a different outcome, based on information given

```
# returns logical representing whether x is even
isEven <- function(x) {
    if (x %% 2 == 0) {
        return(TRUE)
    } else {
        return(FALSE)
    }
}
isEven(42)
```

```
## [1] TRUE
```

```
isEven(13)
```

```
## [1] FALSE
```

or even perform a series of operations.

```
# returns the summation of the first n terms
sumToNum <- function(n) {
    summation = 0
    for (i in 1:n) {
        summation <- summation + i
    }
    return(summation)
}
sumToNum(6)
```

```
## [1] 21
```

From here, we are able to further add to the power of functions with the concept of nested functions. When first introduced to functions, it is typical to assume that these objects are independent of each other, and cannot be placed within one another (this can lead to some interesting scenarios, however); fortunately, this is not the case. Functions may be 'nested' within each other, allowing a single call of a user-defined function to lead to a series of additional calls of other user-defined functions. This fun example shows the power of nested functions; the first function contains a second 'helper' function within, which it uses to perform the desired operations.

```
# Sum of the squares of the first n Fibonacci numbers
sumSquareFibs <- function(n) {
    # returns the xth Fibonacci number
    fibNum <- function(x) {
        if (x <= 0) {
            return(0)
        }
        prev <- 0
        curr <- 1
        for (i in 1:x) {
            temp <- curr + prev
            prev <- curr
            curr <- temp
        }
        return(curr)
    }
    sum <- 0
    for (j in 1:n) {
        sum <- sum + (fibNum(j)*fibNum(j))
    }
    return(sum)
}

sumSquareFibs(1)
```

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

```
sumSquareFibs(2)
```

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

```
sumSquareFibs(3)
```

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

An important note: functions defined within other functions typically do not override a function of the same name, as shown in this example:

```
# foo!
foo <- function(x) {
    print("Original foo")
    return(x+1)
}
# bar?
bar <- function(y) {
    foo <- function(x) {
        print("New foo")
        return(x-1)
    }
    print("bar")
    return(foo(y))
}
foo(0)
```

```
## [1] "Original foo"
```

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

```
bar(0)
```

```
## [1] "bar"
## [1] "New foo"
```

```
## [1] -1
```

```
foo(0)
```

```
## [1] "Original foo"
```

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

Here, `foo()` produces the same result, even though we redefined the function within `bar()`. This is the result of the fact that the second time `foo()` was defined, it occurred within the local environment of `bar()`. This means that as soon as `bar()` is finished executing, all variables defined within are discarded (including the new definition of `foo()`), and the only variables available are those which were available when `bar()` was called. This is why `foo()` returns the original result, instead of remaining overridden when `bar()` has finished. Although the variables defined within a function are discarded once that function has finished executing, variables defined outside of the function's local environment are fair game, as is demonstrated in this example:

```
# foo!
foo <- function(x) {
    print("Original foo")
    return(x+1)
}
# bar?
bar <- function(y) {
    print("bar")
    return(foo(y))
}
foo(0)
```

```
## [1] "Original foo"
```

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

```
bar(0)
```

```
## [1] "bar"
## [1] "Original foo"
```

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

```
foo(0)
```

```
## [1] "Original foo"
```

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

If, however, you would like to define a function within another function and have it remain after the outer function's call finishes, you can use a special type of assignment, as is shown below:

```
# foo!
foo <- function(x) {
    print("Original foo")
    return(x+1)
}
# bar?
bar <- function(y) {
    foo <<- function(x) {
        print("New foo")
        return(x-1)
    }
    print("bar")
    return(foo(y))
}
foo(0)
```

```
## [1] "Original foo"
```

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

```
bar(0)
```

```
## [1] "bar"
## [1] "New foo"
```

```
## [1] -1
```

```
foo(0)
```

```
## [1] "New foo"
```

```
## [1] -1
```

This assignment technique changes the global definition of the variable, allowing it to persist throughout function calls.

Recursion

These topics covered so far may seem like they are leading nowhere, but this section is where they all come together. Recursion is a powerful technique for repeatedly breaking down a problem into smaller subproblems, in order to make it easier to solve. This is made possible via a compilation of the techniques discussed above, with a special emphasis on `if` statements and nested functions. Here is a familiar example, to get us started:

```
# recursively returns the xth Fibonacci number
fibRecursive <- function(x) {
    if (x == 0) {
        return(0)
    } else if (x == 1) {
        return(1)
    } else {
        return(fibRecursive(x-1) + fibRecursive(x-2))
    }
}
fibRecursive(3)
```

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

```
fibRecursive(5)
```

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

```
fibRecursive(8)
```

```
## [1] 21
```

In this function, we make use of a recursive definition in which we call the very function we are defining in the function. This sounds confusing, but it's really very elegant and simple. As an easier example, let's look at a cumulative summation function:

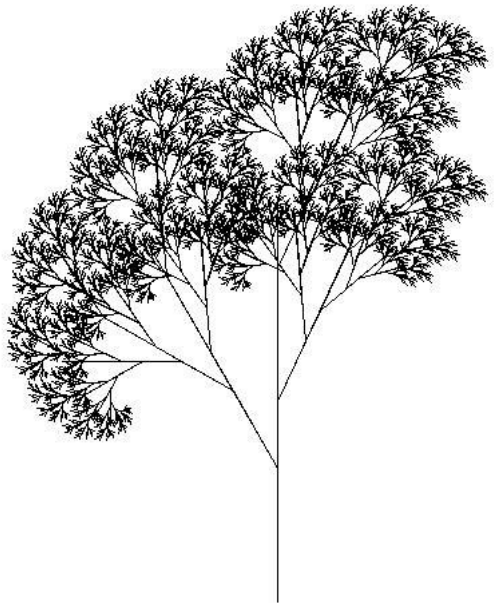
```r
# sums the first n positive integers
sumCum <- function(n) {
    if (n < 1) {
        return(0)
    }
    return(n + sumCum(n - 1))
}
sumCum(6)
```

```
## [1] 21
```

Here, we see that a call to `sumCum(6)` will return `6 + sumCum(5)`. When we call `sumCum(5)`, we return `5 + sumCum(4)`. This process continues until we call `sumCum(1)`, which returns `1 + sumCum(0)`. When we attempt to return `sumCum(0)`, our conditional will be `TRUE`, and we will have reached what is known as the base case. Entering this if statement, we end our recursive calls and return a tangible value. Now, we are able to collect the sum of all the previous calls, resulting in our answer of `21`.

To make a successful recursive function, one should include certain things: - At least one base case is mandatory, otherwise you might recurse infinitely and never return (oh no!) - At least one recursive step. This refers to the 'meat' of the function and what will occur at each call of the function - A recursive call at the end of each step, calling the function in such a way that it will reach the base case. If you do not move 'toward' your base case (e.g. if we had returned `n + sumCum(n)` at the end of the previous example), you will never reach our base case and will recurse ad infinitum. Generally, these are what constitute proper recursion. As you make more complex recursive functions, however, they make get complicated and you may become unsure of whether the recursion will execute as expected. When this occurs, it is sometimes more worthwhile to, instead of attempting to calculate exactly the result returned, take the 'recursive leap of faith' and assume that if your function appears to fulfill all the requirements, then the recursion will succeed. This often proves to be correct, as your intuition for creating recursive functions will make this 'leap of faith' easier as you practice.

To demonstrate the power of recursion, here are some examples of what recursion looks like in action:



When graphing, recursion can create mesmerizing trees

Artist's interpretation of recursion

References

- Professor Sanchez; 'Getting Started with Functions'
- Professor Sanchez; 'Basics of R Expressions and Conditionals'
- Professor Sanchez; 'Introduction to Loops'
- Professor Hilfinger; CS61A & CS61B Lectures/Assignments
- Wikipedia - Abstraction; 'https://en.wikipedia.org/wiki/Abstraction_(software_engineering)'
- Wikipedia - Modular Programming; 'https://en.wikipedia.org/wiki/Modular_programming'
- Wikipedia - Recursion (Computer Science); 'https://en.wikipedia.org/wiki/Recursion_(computer_science)'
- Hackernoon - What is Recursion; 'https://hackernoon.com/what-is-recursion-a1c26baa1d36'
- Jackson Leisure; 'A Recursive Take on Functions'