Functions: Basics and Lambda Expressions

Stat 133, Fall 2024

Lectures 7 & 8, 09/18/2024

1. Introduction to Functions

Functions are a way of modularizing our code, thus making it easier to reuse as needed (instead of the alternative: copy-pasting the same block of code multiple times). A function has 3 building blocks: (1) input e.g., arguments or parameters, (2) computation or processing, and (3) an output. The general syntax of a function in R is:

```
function_name = function(arg1, arg2) { # code goes here }
```

This concise format is often preferred when functions are simple or short, and the body of the function can fit easily on a single line.

1.1. Example

Let's create a simple function that adds two numbers:

```
add_numbers = function(x, y) { return(x + y) }
```

To use this function:

```
add_numbers(3, 5) # returns 8
```

2. Arguments and Defaults

Functions in R can have default values for arguments. This allows a function to be called with fewer arguments than it is defined with, using the default values for missing parameters.

2.1. Example with Defaults

Consider the following function:

```
greet = function(name = "World") { return(paste("Hello,", name)) }
```

Calling the function with or without an argument:

- `greet()` returns `"Hello, World"`
- `greet("Bob")` returns `"Hello, Alice"`

2.1. Continued

Default arguments provide flexibility but can sometimes lead to confusion if not handled carefully, especially when the order of arguments matters.

3. Returning Values from Functions

In R, a function returns the value of the last expression evaluated. However, to be consistent with other programming languages, it's good practice to use the `return()` function (explicitly) to indicate what the output of the function should be.

3.1. Example

Here is a function that returns the square of a number:

```
square = function(x) { return(x**2) }
```

3.2. Implicit Return

Alternatively, without using `return()`:

```
square = function(x) { x**2 }
```

Both versions of the function behave identically. However, using `return()` can make the function's intent clearer.

4. Anonymous Functions and Lambda Expressions

Lambda expressions (aka anonymous functions), are functions that are defined without a name. They are typically used for short-term operations where defining a fully-named function would be cumbersome or unnecessary. Lambda expressions allow us to pass quick, inline logic into functions without the need for a formal function definition.
The general syntax of a lambda expression in R is:

```
function(arg1, arg2, ...)  { # code or expression goes here }
```

Lambda expressions are especially useful when working with functions like `apply()`, `lapply()`, `sapply()`, and other higher-order functions that take other functions as arguments.

4.1. Example: Using Lambda Expressions in `lapply()`

Consider the following list of numbers:

```
lst = list(1, 2, 3, 4, 5)
```

We want to compute the square of each number. Instead of defining a named function for this, we can directly use a lambda expression inside `lapply()`:

```
lapply(lst, function(x) { x**2 })
```

This results in a list where each element is the square of the corresponding element from the original list:

```
{1, 4, 9, 16, 25}
```

4.2. More Complex Example: Filtering with Lambda Expressions

Lambda expressions are also useful for filtering data. Suppose we want to filter out numbers less than 3 from a vector. We can do this easily with the `Filter()` function:

```
Filter(function(x) { x >= 3 }, c(1, 2, 3, 4, 5)) # returns {3, 4, 5}
```

Here, the anonymous function `function(x) { x >= 3 }` is passed directly into `Filter()`, specifying that only elements greater than or equal to 3 should be kept.

4.3. Lambda Expressions and Closures

Closures, or functions that capture the environment in which they were created, can also be constructed using lambda expressions. Here's a more complex example that shows how lambda expressions can work with closures:

```
make_adder = function(n) { return(function(x) { x + n }) }
```

The `make_adder` function returns a lambda expression that adds `n` to any input `x`. For instance:

```
add_three = make_adder(3)
add_three(5) # returns 8
```

In this case, `add_three` is a function that adds 3 to its input, and it remembers the value of `n` that was passed to `make_adder()`. This behavior is made possible by lexical scoping in R.

4.4. Function Composition

Lambda expressions can also be used in function composition, where the output of one function is used as the input of another. For example:

```
composer = function(f, g) { return(function(x) { f(g(x)) }) }

double = function(x) { x**2 }
square = function(x) { x**2 }
double_then_square = composer(square, double)
double_then_square(3) # returns 36
```

Here, the lambda expression `function(x) { f(g(x)) }` allows us to compose two functions, `square` and `double`, into one that first doubles a number and then squares the result.

5. Scope and Environments

R uses lexical scoping to determine where a variable's value is looked up. The value of a variable is determined by the environment in which the function was defined, not the environment in which it was called.

5.1. Example: Lexical Scoping

Consider the following code:

```
x = 10
f = function() { x = 5; return(x) }
f() # returns 5
x # returns 10
```

Here, the value of `x` inside the function is 5, but the value of `x` outside the function remains 10. This is because the function `f()` has its own environment where the variable `x` is redefined.

6. Summary

Functions allow us to modularize certain parts of code that repeats and define customs functions that, well, do certain tasks (which may not be pre-defined in R) Lambda expressions (I like to use them whenever possible) provide a powerful and flexible way to work with functions in R and are useful when a quick, temporary function is needed, thus allow us to write even more concise and clean code. Scope and environments is also a critical concept to understand, and can be helpful when writing code (and especially debugging).