The very basics of programming

December 26, 2017

One of the very nice features of R when one comes from other statistical software like SAS or Stata is that it is very easy to write customized code and develop your own functions. That is a very good thing because the manipulation, analysis, and visualization of data is considerably easier when one can write small functions. In addition, because R is open-source, you can inspect what every function does, so it is useful to get a sense of the most basic elements of programming to be able to take advantage of that feature.

Defining functions

Think about functions as a way of packing operations in one reusable expression. For instance, consider a trivial example in which we want to calculate a mean. Instead of doing `sum` over `length` in every situation, we could just pack those operations together and give them a name:

```
my_mean <- function(x) sum(x)/length(x)
my_mean(c(0, 1))
```

```
## [1] 0.5
```

To define a function, we use the keyword `function` followed by parentheses, (optionally) the arguments that the function takes, and the expression that the function runs. We then assign this function to a name, in this case, `my_mean`.

See how typing `my_mean` allows you to see what you have just defined.

```
my_mean
```

```
## function(x) sum(x)/length(x)
```

The way we defined the function is perfectly valid but we could also be a bit more explicit by enclosing the statement in parenthesis and ensuring that it is returned.

```
my_mean <- function(x) {
  return(sum(x)/length(x))
}
```

Note the use of braces to enclose the contents of the function. Braces are optional when the function body consists of a single line but are necessary when the functional body contains multiple lines.

Closures

A closure can be thought of as a function with data attached. They can reduce bookkeeping and improve readability by keeping copies of data instead of passing them as parameters to the function each time. They are useful when a function is called multiple times but one or more parameters to the function does not change between calls. Closures are normally created by other functions. Here as an example.

```
setup_bootstrap <- function(dat) {

  n <- nrow(dat)

  function() {

    dat[sample.int(n = n, size = n, replace = TRUE),]
```

```
  }

}

fake_data <- data.frame(x = 1:5, y = 2:6)

bootstrap <- setup_bootstrap(fake_data)

mysample <- bootstrap()

# Get 30 bootstrap samples
more_samples <- lapply(1:30, function(unused) bootstrap())
```

`bootstrap` is a closure that is returned by `setup_bootstrap` (recall that functions in R are first-class, so we can return them like any other variable). Notice that we don't need to pass a copy of `fake_data` to `bootstrap` when we call it. That is because `bootstrap` already has an internal copy of the data set.

R uses something called lexical scoping. This means that if a function refers to a variable that is not defined within the function and is not passed as a parameter to the function, the next place R looks for the variable is where the function is defined (this is in contrast to dynamic scoping, which R's ancestor S uses, that instead looks where the funciton is called, not defined). `dat` and `n` are not passed to the anonymous (i.e., unnamed) function within `setup_bootstrap`, but that anonymous function can still see them. When `setup_boostrap` returns, `dat` and `n` would normally go out of scope and disappear, but the newly created function, `bootstrap`, keeps a copy because of lexical scoping. Note that changing `fake_data` will not update the copy within `bootstrap`, since the version within `bootstrap` is from the `dat` parameter of `setup_bootstrap`, not the global variable `fake_data`.

Technically, all user-defined funtions in R are closures. Keep this in mind when defining functions that return other functions. The resulting closure will keep the data defined within its lexical scope around "just in case," increasing the memory use of your program.

Conditionals

As you can imagine, conditionals execute an action depending on the value (`TRUE` or `FALSE`) of another statement. In R, they use the follow syntax:

```
if (x > 0) {
  print("It's positive")
} else if (x == 0) {
  print("It's neither positive nor negative")
} else {
  print("It's negative")
}
```

which we may wrap up in a function

```
my_sign <- function(x) {
  if (x > 0) {
    out <- "It's positive!"
  } else if (x == 0) {
    out <- "It's neither positive nor negative!"
  } else {
    out <- "It's negative!"
  }
```

```
  return(out)
}
```

and use it to illustrate what happens when you are not careful:[^1]

```
my_sign("A cow")
```

```
## [1] "It's positive!"
```

How to deal with the previous issue? By issuing an error:

```
my_sign <- function(x) {

  if (!is.numeric(x)) {
    stop("Input is not a number")
  }

  if (x > 0) {
    out <- "It's positive!"
  } else if (x == 0) {
    out <- "It's neither positive nor negative!"
  } else {
    out <- "It's negative!"
  }
  return(out)
}
```

Two things two notice here. First, that `!` is the negation operator (`TRUE == !FALSE`). Second, that `stop` interrupts the evaluation and produces an error, so the function never reaches the next conditional.

Vectorized operations

One common need is to apply an operation to each individual element of a vector or list. As in most programming languages, this can be done in R through looping constructs. However, the use of loops in R is considered by some to be bad practice, and loops can significantly slow down the execution of one's code in some cases, particularly when updating a data frame within a loop. This is not always true, such as when adding elements to a allocated list, but even in cases where loops do not increase run time, one should be hesitant to use loops since they are not consistent with R's status as a (somewhat) functional vector-based language. Another reason to avoid loops is that they can be difficult to translate into parallelized code.

What do we do then? The alternative in R is vectorized operations. There are a number of ways to do this. First, let's define a function that calculates a factorial of a number.

```
myfactorial <- function(x) {

  if(x == 0) {
    return(1)
  } else {
    return(x*myfactorial(x-1))
  }

}
```

Note that this function calls itself. This is known as recursion. In practice, some checks on the parameter should probably be added to assure that it's a nonnegative integer, and there are surely faster, safer implementations (a production-quality version would likely not use recursion), but let's not worry about that for now. Let's instead consider what we would need to do to apply the function to each element of a vector.

We cannot simply pass in the vector because the function expects a scalar, i.e., a vector of length 1. There are a few ways around this.

One option is to use the `apply` family of functions. These are functions that can apply a function to individual elements of a vector or list, or apply a function to each row or column of a matrix. In this case, we would use `sapply`.

```
results <- sapply(1:5, myfactorial)
```

The first parameter to `sapply` is the vector, and second parameter is the function which we want to apply the vectors elements. `sapply` will return a vector containing the resulting factorials. Note that if the return values are not all of the same type, then `sapply` will instead return a list. If you want a predictable return type, one can use `lapply` instead, which works the same way but always returns a list.

There are other functions in the apply family, such as `apply`, which applies a function to every row or column of a matrix, but they are used in a similar manner (see: `mapply`, `tapply`, `vapply`). Further, some R users prefer a group of functions defined in the `plyr` and `dplyr` packages which perform apply-like operations. These packages are briefly discussed in the next section.

A second option is to create a new function which handles all of these `apply` operations automatically. This can be done through the `Vectorize` function. We would define a new vectorized version like this:

```
myfactorial_vectorized <- Vectorize(myfactorial)
```

Alternatively, we could have done this in our original function definition.

```
myfactorial <- Vectorize(function(x) {

  if(x == 0) {
    return(1)
  } else {
    return(x*myfactorial(x-1))
  }

})
```

Using our new "vectorized" function is simple:

```
results <- myfactorial_vectorized(1:5)
```

Please note that `Vectorize` has one important parameter that we did not mention: `vectorize.args`. This should be set to a vector of strings which contain the variable names we wish to "vectorize" over. For example, suppose we have the following function:

```
weird_sum <- function(x, y, z) {

  x + y + sum(z)

}
```

Now assume we want to vectorize the function over `x` and `y` but not `z`, i.e., we want to add the first element of `x`, the first element of `y`, and the sum of `z`, and then the second element of `x`, the second element of `y`, and the sum of `z`, and so on. We would do it like this:

```
weird_sum <- Vectorize(function(x, y, z) {

  x + y + sum(z)

}, vectorize.args = c("x", "y"))
```

Actually, `weird_sum` will work as intended without the use of `Vectorize` but only because the `+` operator is itself vectorized. For the sake of illustration, let's ignore this pesky detail.

We did not need to specify `vectorize.args` in our `myfactorial` example because the default value is to vectorize over all arguments, which is what we wanted.

In some cases, both options are unnecessary because there are already built-in vectorized functions to accomplish our task. For example, the built-in `factorial` function in R is already vectorized, so there is no need to use `sapply`, `Vectorize`, etc. Similarly, there are a number of built-in functions for calculating sums and means over rows or columns of a matrix, e.g., `rowSums`, `colMeans`, which should be used in lieu of applying `sum` or `mean` to individual rows or columns via `apply`. The use of these kind of functions is simpler and often results in faster programs than the methods mentioned above, so one should use explicit vectorization only as needed.