Functional Programming in R, Part II

We'll cover slightly more complex functions in this lesson.

`apply()`

Calling `apply(mat, dims, func)` will preserve the dimensions specified in `dims` and collapse the rest of the dimensions to single values using `func()` for every combination of the values taken on by the dimensions of `dism`.

For example, we can take row means of a matrix like so:

```
> m = matrix(1:9, nrow=3)
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> apply(m, 1, mean)
[1] 4 5 6
> rowMeans(m)
[1] 4 5 6
```

Since we passed in a dimension of `1` to `apply()`, for every value of the 1st dimension (*i.e.,* for every row number) all the data corresponding to that value (*i.e.,* each row) was passed in to `mean()`. As such, we end up taking the row means of the matrix.

Exercise. What will happen when we call `apply(m, c(1, 2), mean)`? Predict an answer before running the code.

`apply()` is mostly useful for running functions over every row of a data frame.

`outer()`

For *creating* matrices and arrays, we have `outer(A, B, func)`, which iterates over *every combination of values in `A` and `B`* and applies `func()` to both values. The `func` argument defaults to normal multiplication, so the functionality of `outer()` can be easily demonstrated in the creation of a times table:

```
> outer(1:3, 1:4)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    2    4    6    8
[3,]    3    6    9   12
```

Some operations become very easy with `outer()`, so let's return to past assignments and see how we can speed things up.

Exercise. Using `outer()`, write answers for these old questions:

- Make a data frame where the nth column is a logical vector with `TRUE` in position m if F_m divides F_n and `FALSE` otherwise.
- Write a function `min_matrix(n, m)` with `n` rows and `m` columns where the value in row i, column j is equal to $\min(i, j)$.

`Map()`

We'll begin with a discussion of `mapply()`, upon which `Map()` is built.

`mapply()` applies a function (which accepts multiple parameters) over multiple vectors of arguments, calling the function on the first element of each list, then the second elements, and so on and so forth. Precisely, it accepts as input a function `func` and `N` equivalently-sized lists of arguments `args1`, .., `argsN`, each of length `k`. It returns as output a list containing `func(args1[1], ..., argsN[1])`, `func(arg1[2], ..., argsN[2])`, ..., `func(args1[k], ..., argsN[k])`.

Intuitively, you can think of `mapply()` as walking down multiple parallel vectors of arguments, applying the function to each row in turn and returning the results. Alternatively, you can also think of `lapply()` as being a stunted version of `Map()` which can only iterate over one vector of arguments instead of arbitrarily many.

`Map()` is a wrapper for `mapply()` that calls it with the parameter `simplify=FALSE`. This is usually good, because the `simplify=TRUE` default can result in odd, unexpected behavior.

Exercise. Using `Map()`, write a function that takes two lists of equal size, `values` and `weights`, and applies `weighted.mean()` to calculate the mean of each vector in `values` weighted by the corresponding weights in `weights`. Test your function on the inputs `values = lapply(1:10, function(x) rnorm(10))`; `weights = lapply(1:10, function(x) rnorm(10))`. Return the output as a vector.

Exercise. Modify your previous function for applying `weighted.mean()` over a list of vectors so that the mean of vectors containing `NA`s ignores them.

`Reduce()`

The `Reduce(func, vec)` function calls `func()` on the first two elements of `vec`, and then calls `func()` on the output and the third element of `vec`, and so on and so forth. That is, `Reduce(f, 1:4)` is equivalent to `f(f(f(1, 2), 3), 4)`.

Exercise. Implement your own version of `sum()` using `Reduce()` and addition. (*Hint:* `"+"` counts as a function.)

Exercise. Write `my_union(x, y)` and `my_intersect(x, y)` functions using `Reduce()` and set operations (see `?set`) that take lists of arbitrarily many vectors and calculates, respectively, the union or intersection of all of them.

Exercise. There are functions which, when passed into `Reduce()`, give a different overall result depending on whether `Reduce()` starts with the two leftmost or the two rightmost elements of the vector it's operating on. Write a function that runs `Reduce()` in both directions and, if the two results are the same, returns the result, and returns `NA` otherwise.

Exercise. Implement your own version of `Reduce()` with all the basic functionality.

`Filter()`, `Find()`, and `Position()`

All three of these functions accept a function `func()` as their first argument and a vector or list `vals` as their second argument, with the restriction that `func()` must return only `TRUE` or `FALSE` when applied to the entries of `vals`.

- `Filter()` returns the elements in `vals` for which `func()` returns `TRUE` when evaluated on each of those elements.
- `Find()` returns the first element in `vals` for which `func()` returns `TRUE` when evaluated on that element.
- `Position()` returns the position of the first element in `vals` for which `func()` returns `TRUE` when evaluated on that element.

Both `Find()` and `Position()` search from the left by default, but they can search starting from the right with the parameter `right=TRUE`.

Advanced R., 11.4.3.4. Implement `Any()`, a function that takes a list and a predicate function (a function returning either `TRUE` or `FALSE`), and returns `TRUE` if the predicate function returns `TRUE` for any of the inputs. Implement `All()` similarly.[1]

[1] *Hint:* The logical operators `"|"` and `"&"` can be passed into `Reduce()`.