R: Functional Programming

Signal Data Science

So far, you've been using `for` and `while` loops in R for iteration. There are, however, benefits to a functional programming approach.

In an iterative style, we *loop* through values and successively manipulate each value, whereas in a functional style we *apply* some function to every value independently. It's easiest to illustrate with an example.

Suppose that I have the following dataframe:

```
> df = data.frame(matrix(1:100, nrow=10))
> df
   X1 X2 X3 X4 X5 X6 X7 X8 X9 X10
1   1 11 21 31 41 51 61 71 81  91
2   2 12 22 32 42 52 62 72 82  92
3   3 13 23 33 43 53 63 73 83  93
4   4 14 24 34 44 54 64 74 84  94
5   5 15 25 35 45 55 65 75 85  95
6   6 16 26 36 46 56 66 76 86  96
7   7 17 27 37 47 57 67 77 87  97
8   8 18 28 38 48 58 68 78 88  98
9   9 19 29 39 49 59 69 79 89  99
10 10 20 30 40 50 60 70 80 90 100
```

Now, perhaps I would like to calculate the mean of every column.

One way to do this is to loop through the columns and use `mean()`:

```
> means = c()
> for (i in 1:ncol(df)) {
+   means = c(means, mean(df[[i]]))
+ }
> means
 [1]  5.5 15.5 25.5 35.5 45.5 55.5 65.5 75.5 85.5 95.5
```

However, I can do this in a somewhat more compact fashion by using R's `sapply()`:

```
> means = sapply(1:ncol(df), function(i) mean(df[[i]]))
> means
```

```
 [1]  5.5 15.5 25.5 35.5 45.5 55.5 65.5 75.5 85.5 95.5
```

In general, the family of *apply() functions in R all facilitate programming in a functional paradigm.

lapply()

We'll first learn about functional programming by using `lapply()`. The other `*apply()` functions are mainly extensions of `lapply()`, and we'll cover them later.

A picture is worth a thousand words:

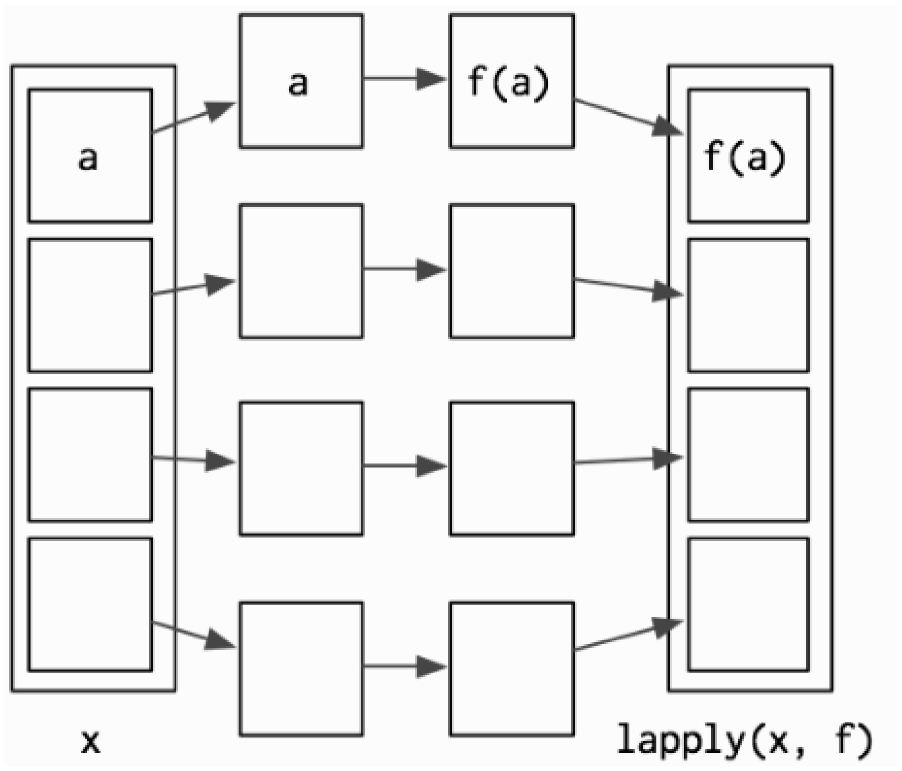

Figure 1: A visual illustration of `lapply()` from *Advanced R*.

Here's an example of using `lapply()` to double every number in a vector. Run the following code:

```
double = function(x) {
  2*x
}
lapply(1:10, double)
```

We first create a function `double(x)` and then we `lapply()` the `double()` function onto the vector `1:10`, with the result of each computation returned in a list. In general, when calling `lapply(values, func)`, each value of `values` is supplied as an unnamed first argument to `func()`.

- Why might we want to return the output of `lapply()` in a *list* by default instead of just `unlist()`ing the values automatically?[1]

We can write this more compactly using an *anonymous function*, which is an unnamed function defined for use in a local context only. Run the following code:

```
> lapply(1:10, function(x) 2*x)
 [1]  2  4  6  8 10 12 14 16 18 20
```

If we anticipate that we won't be using a function often enough to give it a name, we can define it within `sapply()` like we did with `function(x) 2*x`. (Recall that a function doesn't need an explicit `return()` statement – it returns the last expression evaluated by default – and that it only needs curly braces if the body of the function has multiple expressions.)

- Write a function using `lapply()` and `class()` to print out the class of each column in the built-in `mtcars` dataset. Run `unlist()` at the end so it prints in a more human-readable format. (*Hint:* Remember that data frames are built on top of lists.)
- Write a function using `lapply()` to standardize each column of `mtcars` by (1) subtracting off its mean and (2) dividing it by its standard deviation (given by `sd()`). Be sure to check that your function returns a data frame.
- Write a function using `lapply()` that standardizes every numeric column of an input data frame and leaves the others unchanged. Test your function on the dataframe defined by `df = data.frame(matrix(1:100, nrow=10)); df[1:5] = lapply(df[1:5], as.character)` (understand what this code is doing as well).
- Implement a basic version of `lapply(args, func)` without using any other `*apply()` functions. Don't worry about complicated functionality like passing in named arguments (discussed below). Make sure to improve runtime by preallocating the needed space.

[1]Functions in R don't have a return type, so we don't know in advance what they'll return. Although `double()` only returns numerics, that isn't always the case, so it's best to return results in a `list()`, which allows for multiple types in its entries.

Looping patterns

We'll pause for a moment to discuss, at a higher level, the operation of looping.

In general, there are three main ways to loop through a list-based data structure:

1. Looping through the elements: `for (col in df)`
2. Looping through the indices: `for (i in 1:length(df))`
3. Looping through the names: `for (n in names(df))`

(Remember that a data frame is just a list, so the first loop iterates through each column individually and the second loop iterates from 1 to the number of columns in `df`.)

- Write a function that takes a data frame as input and modifies each column to be equal to itself minus the *previous* column, with the first column remaining unchanged. Test your function on `df = data.frame(matrix(1:100, nrow=10))` – aside from 9 entries in the first column, every entry should be equal to 10.

The first form of iteration is the simplest, but you don't get the name or index of each item, just the item itself. The second and the third are more complex, but provide you with more information, so keep them in mind – they may be helpful for more complex problems.

`vapply()` and `sapply()`

`Lapply()` is the most basic of the `*apply()` functions, but there are more. Here's a brief description of two more functions to give you a sense for the overall landscape:

1. `lapply()` maps a function onto a list and *returns a list*. (Listed here for comparison purposes.)
2. `vapply()` is an extension of `lapply()` that maps a function onto a list and *returns an atomic vector*. It takes an additional argument specifying the *type* and *length* of each element of the return vector, throwing an error if they don't match.
3. `sapply()` is an extension of `lapply()` which will `unlist()` the results. If appropriate, it will also assign dimensions to the output, turning it into a matrix.

It's dangerous to use `sapply()` when writing functions you'll use elsewhere, because you won't know if your output is an unexpected type or has an unexpected length until your program exhibits strange behavior elsewhere. It's better to use `vapply()`, which throws an error when the output isn't of the specified type and length and enforces type consistency in various edge cases.

However, it's fine to use `sapply()` when working interactively in the console, where you'll be able to visually notice any strange behavior.

Passing in named arguments

If you have `sapply(df, func)` and want to pass in named arguments to every call of `func()`, you can do so by passing in named arguments into `sapply()` directly, *e.g.*, `sapply(df, func, param=TRUE)` will call `func(c, param=TRUE)` for every column `c` of `df`.

Suppose that we define the `multiply()` function as follows:

```
multiply = function(x, k=2) {
  k*x
}
```

Without being able to pass in named arguments as described above, if we wanted to call `multiply()` with `k=5`, we would have to do something ugly with anonymous functions:

```
> sapply(1:10, function(x) multiply(x, k=5))
 [1]  5 10 15 20 25 30 35 40 45 50
```

However, passing in the named arguments to `sapply()` directly is much easier:

```
> sapply(1:10, multiply)
 [1]  2  4  6  8 10 12 14 16 18 20
> sapply(1:10, multiply, k=3)
 [1]  3  6  9 12 15 18 21 24 27 30
> sapply(1:10, multiply, k=10)
 [1]  10  20  30  40  50  60  70  80  90 100
```

- Write a function using `sapply()` to find the mean of every vector in a list of numeric vectors, ignoring `NA` values. Test your function on the list `L = lapply(1:5, function(x) sample(c(1:4, NA)))`.

The same syntax works for `lapply()`. For `vapply()`, the named arguments go after the example return value.

Why use `*apply()` instead of loops?

- Write a function that takes a data frame as input and returns it with its column names modified, where the name of the `n`th column has `"_n"` appended to the end.

At times, the usage of loops is inevitable and the most natural way to program something. Don't get caught up in trying to code something functionally if a loop seems intuitive. In particular, these three use cases are more suitable for loops than for functional programming:

1. Modifying a data structure in place (changing it without making a new copy).
    - This is because you have to use the `<<-` operator to modify the object while situated in the scope of a function call.
2. Recursive functionality, which is self-dependent, contrary to the isolated nature of each function call when using `*apply()`.[2]
3. While loops, because you don't know in advance how many times the loop will iterate.

Moreover, people will sometimes say say that you should use the `*apply()` functions instead of loops because loops are slow. **This is not true.**

As we saw earlier with the n-dominoes problem, loops can be sped up significantly by *preallocating memory* for the data structures which you access. In general, loops can be made approximately as fast as writing equivalent code for a function to be used with `*apply()` if you follow these guidelines[3]:

1. Initialize new objects to full length before the loop, rather than increasing their size within the loop.
    - Every time you increase the size of an object within a loop, you actually *copy the whole structure over to a different part of memory* every single time.
2. Do not do things in a loop that can be done outside the loop.

Given that loops don't actually have performance issues in R, why should we use `*apply()` functions at all? For these two reasons:[4]

1. Using the `*apply()` functions can make it clearer what you're doing.
    - The notion of applying the same function to every element of a list is in general very intuitive. Code clarity is important, both for yourself and for others.
2. The `*apply()` functions have no unwanted side effects.
    - That is to say, their functionality is *isolated* from the rest of your code, so it's harder for you to make accidental modifications to variables you've defined elsewhere.[5]

[2]Recurrence relations can sometimes be "solved" in a sense and transformed into a nonrecursive form that's potentially amenable to functional programming, but this is difficult.

[3]From a 2008 issue of R News.

[4]See the answers to Is R's apply family more than syntactic sugar, including the comments on the first one.

[5]Two caveats are that this isn't true if you use `assign()` or the `<<-` operator, which are seldom used and only show up in very specific situations.

- Also, when calling, say, `sapply(args, func)`, each call of `func()` is completely independent of the other ones. This allows them to easily be dispatched to different processor cores.[6]

Supplemental exercises

- Go back to your old code for various R exercises. Find five functions which could be written more easily or clearly using functional programming (*e.g.*, using the `*apply()` functions instead of a `for` loop) and rewrite them. Check the difference in runtime.
- If you remember writing any R code in the past for exercises where you kept adding values to a vector or list on every iteration of a loop, rewrite the code here using a preallocated data structure (if possible). Check the difference in runtime using the `timeit` package.
- Return to these two exercises from the first day's assignments:

  > Calculate the sums $\sum_{i=10}^{100}\left(i^3+4i^2\right)$ and $\sum_{i=1}^{25}\left(\frac{2^i}{i}+\frac{3^i}{i^2}\right)$ using the `sum()` function.
  >
  > Create a vector of the values of $e^x\cos(x)$ at $x = 3, 3.1, 3.2, \ldots, 6$.

  Answer them again using the `*apply()` functions.
- Return to the exercise yesterday about expanding factors into binary indicator variables and rewrite your function using the `*apply()` functions. You can assume that your data frame contains only factors, because with a more general dataframe you can simply extract the column factors only and operate on those.

`apply()`

Now that we've covered the basics, we'll start to consider some more complex, lesser-used functions.

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:

[6] Multi-core processing packages for R implement parallelization by overwriting the built-in `*apply()` functions with their own versions. As such, liberal usage of `*apply()` in your code means that you'll be able to easily parallelize it without much rewriting.

```
> 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.

- 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()`.

`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.

- 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.
- Modify your previous function for applying weighted.mean() over a list of vectors so that the mean of vectors containing NAs ignores them.

Reduce()

Reduce(func, vec) 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).

- Implement your own version of sum() using Reduce() and addition. (*Hint:* "+" counts as a function.)
- Write my_union(L) and my_intersect(L) functions using Reduce() and set operations (see ?sets) that take lists of arbitrarily many vectors and calculates, respectively, the union or intersection of all of them.
- 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.
- 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`.

1. `Filter()` returns the elements in `vals` for which `func()` returns `TRUE` when evaluated on each of those elements.
2. `Find()` returns the first element in `vals` for which `func()` returns `TRUE` when evaluated on that element.
3. `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`.

- 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.[7]

Writing a simple spellcheck function

Spelling correction is one of the most natural and oldest natural language processing tasks. It may seem like a difficult task to you at the moment, but it's surprisingly easy to write a spellchecker that does fairly well. (Of course, companies like Google spend millions of dollars making their spellcheckers better and better, but we'll start with something simpler for now.)

- Read Peter Norvig's How to Write a Spelling Corrector, paying particular attention to the probabilistic reasoning (which is similar to the ideas behind a naive Bayes classifier). Recreate it in R and reproduce his results.
- **After** implementing your own spellchecker, read about this 2-line R implementation of Norvig's spellchecker.

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