Functional Programming in R

Note: This is a long assignment, so feel free to take a break in the middle. However, the content is **absolutely essential**, so be sure to understand everything here.

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 the `mean()` function:

```
> 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()` function:

```
> 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.

Using `sapply()` with anonymous functions

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:

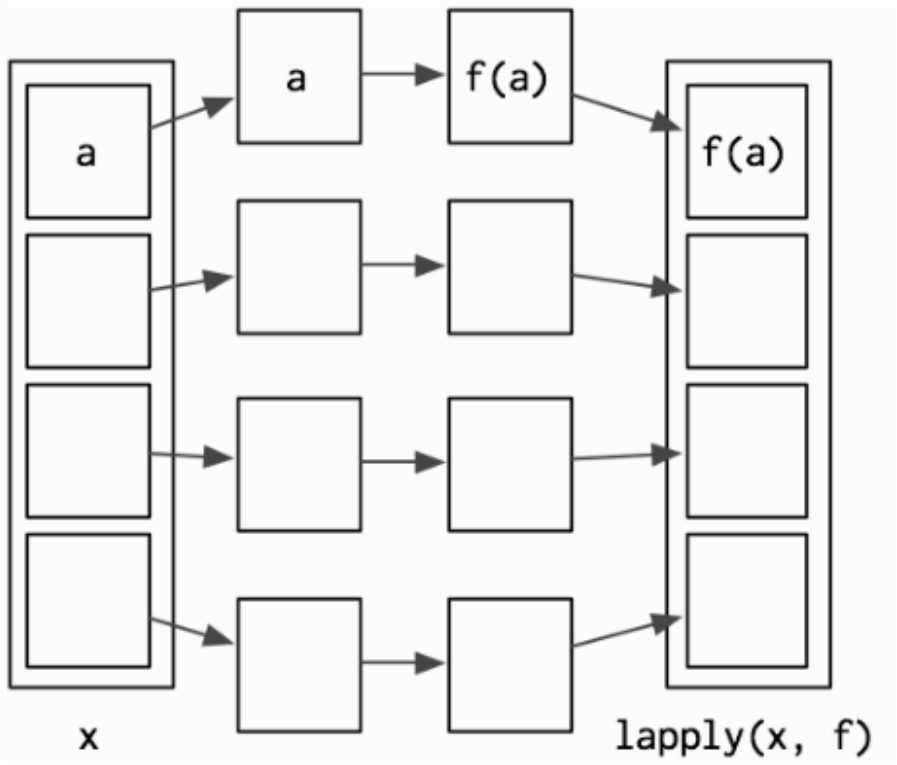

Figure 1: lapply picture

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)` which returns 2 times its argument, 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()`.

Exercise. Why might we want to by default return the output of `lapply()` in a *list* 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 I only need curly braces for a multiline function definition.)

Exercise. 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.)

Exercise. 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.

Exercise. 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).

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:

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

[1] Functions in R don't have a return type, so we don't know in advance what they'll return. Although the `double()` function 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.

Exercise. The `lapply()` equivalent of the first is `lapply(df, function(x) {})`. Write down equivalents for the second and third forms.

The first form is the simplest, but you don't get the name or index of each item. 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.

Other `*apply()` functions

`sapply()` is in fact not the most basic of the `*apply()` functions; `lapply()` is. Here's a brief description of each one to give you a sense for the overall landscape:

- `lapply()` maps a function onto a list and *returns a list*.
- `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* of the vector.
- `sapply()` tries to *guess the best output type*. Operating on the columns of a data frame, `sapply()` will bind the results together into a new data frame. When appropriate, `sapply()` will `unlist()` a list of results into a vector.su
- `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])`.

Remark. 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.

More `sapply()` problems

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`.

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

Exercise. Let `trims = seq(0, 0.5, 0.1)` and `x = rnorm(100)`. Rewrite the expression `lapply(trims, function(trim) mean(x, trim=trim))` to not need an anonymous function.

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

Sometimes, people will 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[2]:

- 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.
- Do not do things in a loop that can be done outside the loop.
- Do not avoid loops simply for the sake of avoiding loops.
  - Sometimes, 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.

Given that loops don't actually have performance issues in R, why should we use `*apply()` functions at all? For these three reasons:[3]

- 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.
- 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.

[2] From a 2008 issue of R News.

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

  - 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.

- 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.
  - It's precisely because the `*apply()` functions have no side effects that they're often used for parallelization purposes. Otherwise, it can be very difficult to ensure correct program behavior.