Lists, Vectors, and their Interesting Defferences

Nicholas Weis

October 31, 2017

```
install.packages('purrr', repos='https://cran.cnr.berkeley.edu/')
```

```
## Installing package into 'C:/Users/Nick/Documents/R/win-library/3.4'
## (as 'lib' is unspecified)
```

```
## package 'purrr' successfully unpacked and MD5 sums checked
##
## The downloaded binary packages are in
##  C:\Users\Nick\AppData\Local\Temp\Rtmpg9NOPg\downloaded_packages
```

```
library(purrr)
```

```
## Warning: package 'purrr' was built under R version 3.4.2
```

One of the topics that has most interested me in the early weeks of the semester is the existence of both vectors and lists, twin data structures that are inherently similar but that function in very different ways. Whereas vectors are limited in the variety of data types they can contain, lists are more freeform and unshackled by the regulations that define its other half. In this blog post, my goal is to show interesting examples of the applications of both vectors and lists in R, before looking at some useful functions that can apply to both of these important data structures.

1.) Vectors

A lot of code in R deals with manipulation of multiple vectors at a time, including comparing and contrasting their contents. A valuable function I've researched is setdiff(), which takes in two vectors as arguments and returns the elements of the first that are not in the second. For example:

```
x <- c(1,2,3,4)
y <- c(2,3,4,5)
setdiff(x, y)
```

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

```
setdiff(y, x)
```

```
## [1] 5
```

Running the previous block of code creates the two vectors x and y, with the following calls to the setdiff() function revealing the differences in the contents of each set. One can imagine how this functionality could be useful when dealing with exponentially large datasets, where the naked eye cannot possibly spot the distinct differences between them. In this example, I formed the vectors myself from scratch. However, in more practical cases, we could be dealing with messier and incompatible datasets that would influence the data in ways we were not otherwise aware of. That is why this function, as well as the following others, are extremely handy to keep in a statistician's toolset.

```
union(x, y)
```

```
## [1] 1 2 3 4 5
```

```
intersect(x, y)
```

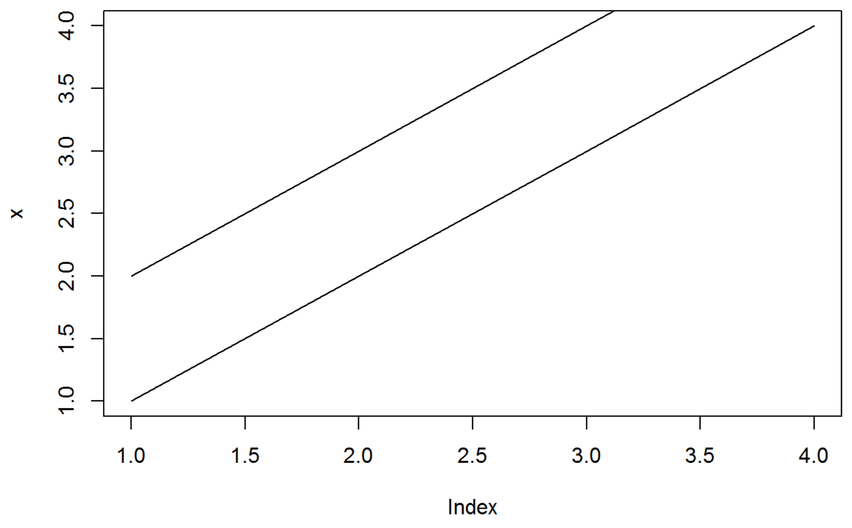
```
## [1] 2 3 4
```

```
setequal(x, y)
```

```
## [1] FALSE
```

setdiff() is only one of a family of functions that highlight the contents of multiple vectors in relation to each other. All of these functions behave similarly to concepts you've most likely seen in mathematical applications, where sets of numbers (or other data types in this case) are mixed or stripped apart based on the nature of another set.

```
plot(x ,type = "l")
lines(y)
```



Here I have included a visualization of vectors x and y, where the index composes the x-axis and the value at each index makes up the y-axis. With a little manual scanning, it's easy to see that the values at each index never exactly match, but that some values certainly match those located at other indexes. Pinpointing how vectors differ and synchronize in these ways is the purpose of these functions, a tool that has obvious implications for the ease of data management.

References: 1- https://stackoverflow.com/questions/1837968/how-to-tell-what-is-in-one-vector-and-not-another 2- http://stat.ethz.ch/R-manual/R-patched/library/base/html/sets.html

2.) Lists

For lists, I would like to examine the mapping functions available in R and their purpose. These functions have a use that mirrors the repeated application to elements of vectors. Each mapping function takes in a list as an argument, as well as a separate function that is applied to each element. It's also definitely worth noting that these functions, as well as many others that applications in relation to lists in R, are included in the purrr package. This must be installed before any of these can be used. Take a look at the following code:

```
z = list(1:3, 2, c(1, 3, 5))
map_int(z, length)
```

```
## [1] 3 1 3
```

```
map_dbl(z, mean)
```

```
## [1] 2 2 3
```

```
map_dbl(z, median)
```

```
## [1] 2 2 3
```

What this code did was calculate the length, mean, and median, respectively, of each element in the list. The results are then outputted in the order of the indexes of the list. The powers of this tool are exponential- similar to the previous discussion of vectors, imagine the simplicity of calculating these mathematical necessities over a list with hundreds, thousands, or possibly even millions of elements.

In addition, there are options included that are built around the hierarchal structure of lists (such as when lists contain other lists as elements). For example, we can remove a level of hierarchy with the function flatten().

```
x2 = list(list(1, 2), list(3, 4))
y2 = flatten(x2)
y2
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
```

This function takes the lists that were included as elements and strips away each of their individual components. Each of these pieces then moves up a level of the hierarchy, in this case resulting in a list with twice the number of surface-level elements.

One more detail about lists that I would like to touch upon here is the speed differences when using different coding strategies in R. It's typical in

R for users to use the wide variety of functions at their disposal to simplify their code. However, the lack of implementation for some data structures in R (like hashsets, my personal favorite from my days as a computer science major) makes the layered application of many merging and vectorized functions unfeasible when aiming for tighter speeds. It's something to think about in the future when dealing with larger and larger datasets that may be affected by the phenomenon.

References: 1- http://r4ds.had.co.nz/lists.html 2- https://www.refsmmat.com/posts/2016-09-12-r-lists.html

3.) Functions for use with both vectors and lists

There is another family of functions that is similar to those described earlier for lists. The leading function is lapply(), which applies a designated function to each element in the provided list. This may sound like a carbon copy of the mapping functions; however, this new set can also be used on vectors. This functionality is accomplished by converting each element into a list object, something that is necessary given the widespread scope of this function. This can be seen in the following code:

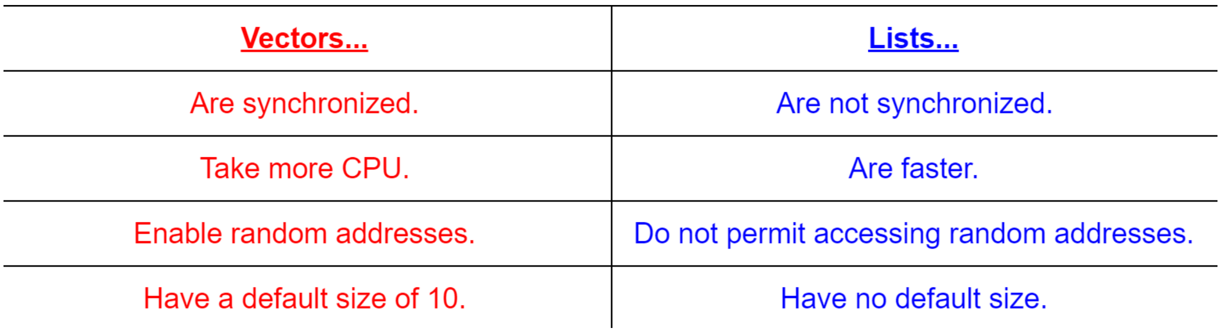
```r
z2 = c(1,2,3,4,5)
lapply(z2, median)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5
```

sapply() will always return a vector or matrix as long as that output is appropriate. replicate() is commonly used alongside sapply() when a computation must be repeatedly evaluated.

Now that we've reviewed this efficient bit of coding, what is one of the most practical uses for lapply()? First and foremost, lists are commonly used when storing objects of multiple dimensions, especially when computing different values based on the contents of each layer in a matrix. This is a fantastic way to manipulate and store data, because we can repeatedly use these functions to treat the data in a multitude of ways, storing the changes at each and every step if we choose. The possibilities are truly endless here.

As a final aside for this section, I would like to point out some of the most important distinctions between lists and vectors. These two structures are so inherently similar that it may be very easy to confuse the pair. Most of these inclusions are related to technical specifications and don't have much to do with standard use, but they are still useful to be aware of.



Vectors...	**Lists...**
Are synchronized.	Are not synchronized.
Take more CPU.	Are faster.
Enable random addresses.	Do not permit accessing random addresses.
Have a default size of 10.	Have no default size.

series2

References: 1 - http://astrostatistics.psu.edu/su07/R/html/base/html/lapply.html 2 - https://www.r-bloggers.com/how-to-use-lists-in-r/ 3 - https://www.youtube.com/watch?v=s-dd4-AP0SA

4.) Take-Home Message

This post has been dedicated to highlighting the functionality of lists and vectors that exists outside their standard documentation. These structures can be manipulated in a variety of ways, only a small snippet of which were discussed here. However, I hope those that read this post will be inspired to look up further methods for playing around with vectors and lists, as well as learning the proper times to use each. They have their pros and cons, al of which are key for data sciientists to be educated on.