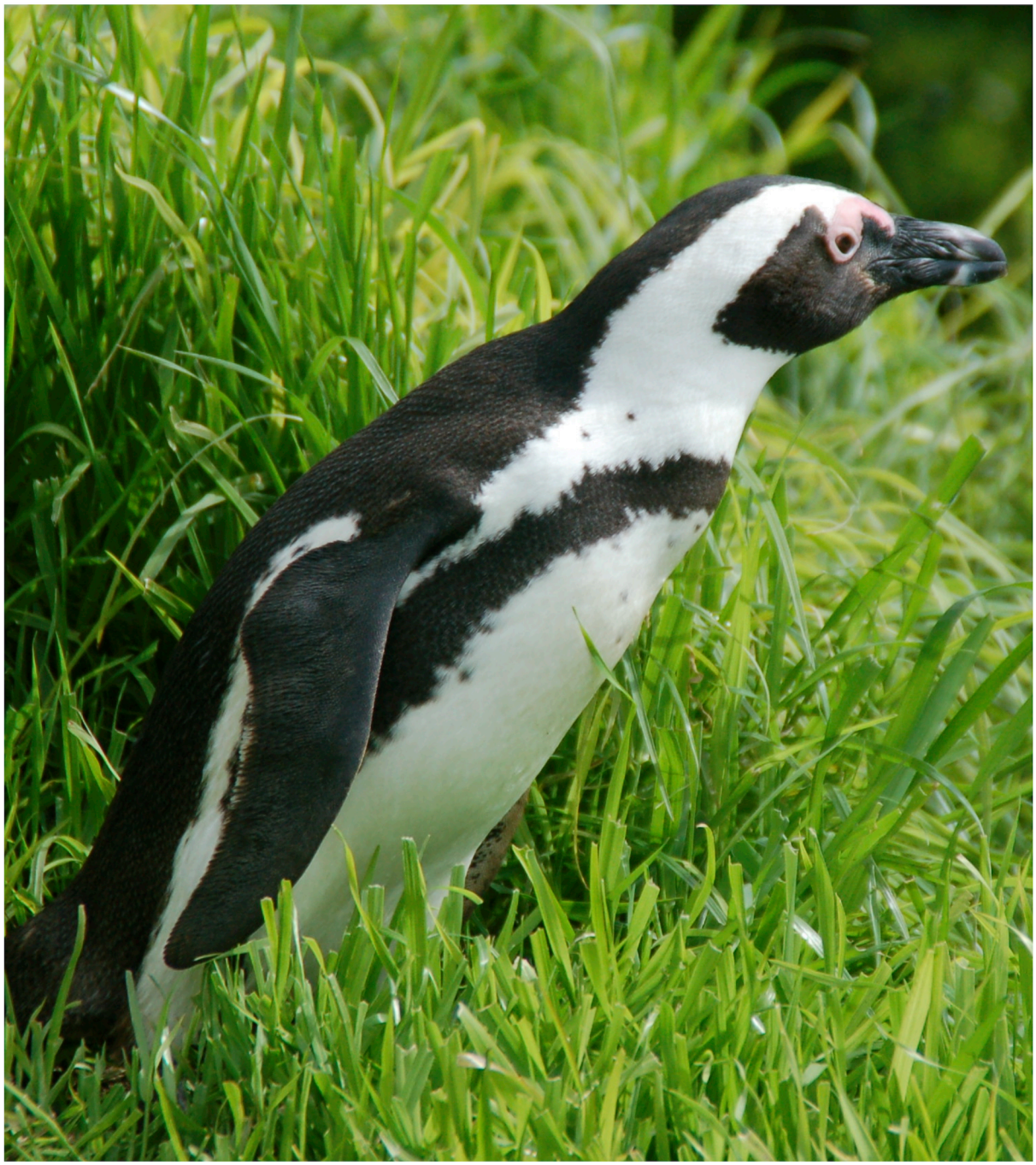
R Programming for Data Science



Roger D. Peng

R Programming for Data Science

Roger D. Peng

This book is for sale at http://leanpub.com/rprogramming

This version was published on 2015-04-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Preface

I started using R in 1998 when I was a college undergraduate working on my senior thesis. The version was 0.63. I was an applied mathematics major with a statistics concentration and I was working with Dr. Nicolas Hengartner on an analysis of word frequencies in classic texts (Shakespeare, Milton, etc.). The idea was to see if we could identify the authorship of each of the texts based on how frequently they used certain words. We downloaded the data from Project Gutenberg and used some basic linear discriminant analysis for the modeling. The work was eventually published[1] and was my first ever peer-reviewed publication. I guess you could argue it was my first real "data science" experience.

Back then, no one was using R. Most of my classes were taught with Minitab, SPSS, Stata, or Microsoft Excel. The cool people on the cutting edge of statistical methodology used S-PLUS. I was working on my thesis late one night and I had a problem. I didn't have a copy of any of those software packages because they were expensive and I was a student. I didn't feel like trekking over to the computer lab to use the software because it was late at night.

But I had the Internet! After a couple of Yahoo! searches I found a web page for something called R, which I figured was just a play on the name of the S-PLUS package. From what I could tell, R was a "clone" of S-PLUS that was free. I had already written some S-PLUS code for my thesis so I figured I would try to download R and see if I could just run the S-PLUS code.

It didn't work. At least not at first. It turns out that R is not exactly a clone of S-PLUS and quite a few modifications needed to be made before the code would run in R. In particular, R was missing a lot of statistical functionality that had existed in S-PLUS for a long time already. Luckily, R's programming language was pretty much there and I was able to more or less re-implement the features that were missing in R.

After college, I enrolled in a PhD program in statistics at the University of California, Los Angeles. At the time the department was brand new and they didn't have a lot of policies or rules (or classes, for that matter!). So you could kind of do what you wanted, which was good for some students and not so good for others. The Chair of the department, Jan de Leeuw, was a big fan of XLisp-Stat and so all of the department's classes were taught using XLisp-Stat. I diligently bought my copy of Luke Tierney's book[2] and learned to really love XLisp-Stat. It had a number of features that R didn't have at all, most notably dynamic graphics.

But ultimately, there were only so many parentheses that I could type, and still all of the research-level statistics was being done in S-PLUS. The department didn't really have a lot of copies of S-PLUS lying around so I turned back to R. When I looked around at my fellow students, I realized that I was basically the only one who had any experience using R. Since there was a budding interest in R

[1] http://amstat.tandfonline.com/doi/abs/10.1198/000313002100#.VQGiSELpagE

[2] http://www.amazon.com/LISP-STAT-Object-Oriented-Environment-Statistical-Probability/dp/0471509167/

around the department, I decided to start a "brown bag" series where every week for about an hour I would talk about something you could do in R (which wasn't much, really). People seemed to like it, if only because there wasn't really anyone to turn to if you wanted to learn about R.

By the time I left grad school in 2003, the department had essentially switched over from XLisp-Stat to R for all its work (although there were a few hold outs). Jan discusses the rationale for the transition in a paper[3] in the *Journal of Statistical Software*.

In the next step of my career, I went to the Department of Biostatistics[4] at the Johns Hopkins Bloomberg School of Public Health, where I have been for the past 12 years. When I got to Johns Hopkins people already seemed into R. Most people had abandoned S-PLUS a while ago and were committed to using R for their research. Of all the available statistical packages, R had the most powerful and expressive programming language, which was perfect for someone developing *new* statistical methods.

However, we didn't really have a class that taught students how to use R. This was a problem because most of our grad students were coming into the program having never heard of R. Most likely in their undergradute programs, they used some other software package. So along with Rafael Irizarry, Brian Caffo, Ingo Ruczinski, and Karl Broman, I started a new class to teach our graduate students R and a number of other skills they'd need in grad school.

The class was basically a weekly seminar where one of us talked about a computing topic of interest. I gave some of the R lectures in that class and when I asked people who had heard of R before, almost no one raised their hand. And no one had actually used it before. The main selling point at the time was "It's just like S-PLUS but it's free!" A lot of people had experience with SAS or Stata or SPSS. A number of people had used something like Java or C/C++ before and so I often used that a reference frame. No one had ever used a functional-style of programming language like Scheme or Lisp.

To this day, I still teach the class, known a Biostatistics 140.776 ("Statistical Computing"). However, the nature of the class has changed quite a bit over the past 10 years. The population of students (mostly first-year graduate students) has shifted to the point where many of them have been introduced to R as undergraduates. This trend mirrors the overall trend with statistics where we are seeing more and more students do undergraduate majors in statistics (as opposed to, say, mathematics). Eventually, by 2008–2009, when I'd asked how many people had heard of or used R before, everyone raised their hand. However, even at that late date, I still felt the need to convince people that R was a "real" language that could be used for real tasks.

R has grown a lot in recent years, and is being used in so many places now, that I think it's essentially impossible for a person to keep track of everything that is going on. That's fine, but it makes "introducing" people to R an interesting experience. Nowadays in class, students are often teaching me something new about R that I've never seen or heard of before (they are quite good at Googling around for themselves). I feel no need to "bring people over" to R. In fact it's quite the opposite–people might start asking questions if I *weren't* teaching R.

[3] http://www.jstatsoft.org/v13/i07

[4] http://www.biostat.jhsph.edu

This book comes from my experience teaching R in a variety of settings and through different stages of its (and my) development. Much of the material has been taking from by Statistical Computing class as well as the R Programming[5] class I teach through Coursera.

I'm looking forward to teaching R to people as long as people will let me, and I'm interested in seeing how the next generation of students will approach it (and how my approach to them will change). Overall, it's been just an amazing experience to see the widespread adoption of R over the past decade. I'm sure the next decade will be just as amazing.

[5] https://www.coursera.org/course/rprog

Loop Functions

Looping on the Command Line

Writing `for` and `while` loops is useful when programming but not particularly easy when working interactively on the command line. Multi-line expressions with curly braces are just not that easy to sort through when working on the command line. R has some functions which implement looping in a compact form to make your life easier.

- `lapply()`: Loop over a list and evaluate a function on each element
- `sapply()`: Same as `lapply` but try to simplify the result
- `apply()`: Apply a function over the margins of an array
- `tapply()`: Apply a function over subsets of a vector
- `mapply()`: Multivariate version of `lapply`

An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.

`lapply()`

Watch a video of this section[6]

The `lapply()` function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a *function* to each element of the list (a function that you specify)
3. and returns a list (the `l` is for "list").

This function takes three arguments: (1) a list `X`; (2) a function (or the name of a function) `FUN`; (3) other arguments via its `...` argument. If `X` is not a list, it will be coerced to a list using `as.list()`.

The body of the `lapply()` function can be seen here.

[6] https://youtu.be/E1_NlFb0E4g

```
> lapply
function (X, FUN, ...)
{
    FUN <- match.fun(FUN)
    if (!is.vector(X) || is.object(X))
        X <- as.list(X)
    .Internal(lapply(X, FUN))
}
<bytecode: 0x7fa339937fc0>
<environment: namespace:base>
```

Note that the actual looping is done internally in C code for efficiency reasons.

It's important to remember that `lapply()` always returns a list, regardless of the class of the input.

Here's an example of applying the `mean()` function to all elements of a list. If the original list has names, the the names will be preserved in the output.

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
[1] 3

$b
[1] 0.1322028
```

Notice that here we are passing the `mean()` function as an argument to the `lapply()` function. Functions in R can be used this way and can be passed back and forth as arguments just like any other object. When you pass a function to another function, you do not need to include the open and closed parentheses `()` like you do when you are *calling* a function.

Here is another example of using `lapply()`.

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.248845

$c
[1] 0.9935285
```

```
$d
[1] 5.051388
```

You can use `lapply()` to evaluate a function multiple times each with a different argument. Below, is an example where I call the `runif()` function (to generate uniformly distributed random variables) four times, each time generating a different number of random numbers.

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.02778712

[[2]]
[1] 0.5273108 0.8803191

[[3]]
[1] 0.37306337 0.04795913 0.13862825

[[4]]
[1] 0.3214921 0.1548316 0.1322282 0.2213059
```

When you pass a function to `lapply()`, `lapply()` takes elements of the list and passes them as the *first argument* of the function you are applying. In the above example, the first argument of `runif()` is `n`, and so the elements of the sequence `1:4` all got passed to the `n` argument of `runif()`.

Functions that you pass to `lapply()` may have other arguments. For example, the `runif()` function has a `min` and `max` argument too. In the example above I used the default values for `min` and `max`. How would you be able to specify different values for that in the context of `lapply()`?

Here is where the `...` argument to `lapply()` comes into play. Any arguments that you place in the `...` argument will get passed down to the function being applied to the elements of the list.

Here, the `min = 0` and `max = 10` arguments are passed down to `runif()` every time it gets called.

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 2.263808

[[2]]
[1] 1.314165 9.815635

[[3]]
[1] 3.270137 5.069395 6.814425

[[4]]
[1] 0.9916910 1.1890256 0.5043966 9.2925392
```

So now, instead of the random numbers being between 0 and 1 (the default), the are all between 0 and 10.

The `lapply()` function and its friends make heavy use of *anonymous* functions. Anonymous functions are like members of Project Mayhem[7]—they have no names. These are functions are generated "on the fly" as you are using `lapply()`. Once the call to `lapply()` is finished, the function disappears and does not appear in the workspace.

Here I am creating a list that contains two matrices.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Suppose I wanted to extract the first column of each matrix in the list. I could write an anonymous function for extracting the first column of each matrix.

[7] http://en.wikipedia.org/wiki/Fight_Club

```
> lapply(x, function(elt) { elt[,1] })
$a
[1] 1 2

$b
[1] 1 2 3
```

Notice that I put the `function()` definition right in the call to `lapply()`. This is perfectly legal and acceptable. You can put an arbitrarily complicated function definition inside `lapply()`, but if it's going to be more complicated, it's probably a better idea to define the function separately.

For example, I could have done the following.

```
> f <- function(elt) {
+         elt[, 1]
+ }
> lapply(x, f)
$a
[1] 1 2

$b
[1] 1 2 3
```

Now the function is no longer anonymous; it's name is `f`. Whether you use an anonymous function or you define a function first depends on your context. If you think the function `f` is something you're going to need a lot in other parts of your code, you might want to define it separately. But if you're just going to use it for this call to `lapply()`, then it's probably simpler to use an anonymous function.

sapply()

The `sapply()` function behaves similarly to `lapply()`; the only real difference is in the return value. `sapply()` will try to simplify the result of `lapply()` if possible. Essentially, `sapply()` calls `lapply()` on its input and then applies the following algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

Here's the result of calling `lapply()`.

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] -0.251483

$c
[1] 1.481246

$d
[1] 4.968715
```

Notice that `lapply()` returns a list (as usual), but that each element of the list has length 1.

Here's the result of calling `sapply()` on the same list.

```
> sapply(x, mean)
        a         b         c         d
 2.500000 -0.251483  1.481246  4.968715
```

Because the result of `lapply()` was a list where each element had length 1, `sapply()` collapsed the output into a numeric vector, which is often more useful than a list.

split()

Watch a video of this section[8]

The `split()` function takes a vector or other objects and splits it into groups determined by a factor or list of factors.

The arguments to `split()` are

```
> str(split)
function (x, f, drop = FALSE, ...)
```

where

- `x` is a vector (or list) or data frame

[8] https://youtu.be/TjwE5b0fOcs

- `f` is a factor (or coerced to one) or a list of factors
- `drop` indicates whether empty factors levels should be dropped

The combination of `split()` and a function like `lapply()` or `sapply()` is a common paradigm in R. The basic idea is that you can take a data structure, split it into subsets defined by another variable, and apply a function over those subsets. The results of applying tha function over the subsets are then collated and returned as an object. This sequence of operations is sometimes referred to as "map-reduce" in other contexts.

Here we simulate some data and split it according to a factor variable.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$`1`
 [1]  0.3981302 -0.4075286  1.3242586 -0.7012317 -0.5806143 -1.0010722
 [7] -0.6681786  0.9451850  0.4337021  1.0051592

$`2`
 [1] 0.34822440 0.94893818 0.64667919 0.03527777 0.59644846 0.41531800
 [7] 0.07689704 0.52804888 0.96233331 0.70874005

$`3`
 [1]  1.13444766  1.76559900  1.95513668  0.94943430  0.69418458
 [6]  1.89367370 -0.04729815  2.97133739  0.61636789  2.65414530
```

A common idiom is `split` followed by an `lapply`.

```
> lapply(split(x, f), mean)
$`1`
[1] 0.07478098

$`2`
[1] 0.5266905

$`3`
[1] 1.458703
```

Splitting a Data Frame

```
> library(datasets)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

We can split the `airquality` data frame by the `Month` variable so that we have separate sub-data frames for each month.

```
> s <- split(airquality, airquality$Month)
> str(s)
List of 5
 $ 5:'data.frame':        31 obs. of  6 variables:
  ..$ Ozone  : int [1:31] 41 36 12 18 NA 28 23 19 8 NA ...
  ..$ Solar.R: int [1:31] 190 118 149 313 NA NA 299 99 19 194 ...
  ..$ Wind   : num [1:31] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
  ..$ Temp   : int [1:31] 67 72 74 62 56 66 65 59 61 69 ...
  ..$ Month  : int [1:31] 5 5 5 5 5 5 5 5 5 5 ...
  ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
 $ 6:'data.frame':        30 obs. of  6 variables:
  ..$ Ozone  : int [1:30] NA NA NA NA NA NA 29 NA 71 39 ...
  ..$ Solar.R: int [1:30] 286 287 242 186 220 264 127 273 291 323 ...
  ..$ Wind   : num [1:30] 8.6 9.7 16.1 9.2 8.6 14.3 9.7 6.9 13.8 11.5 ...
  ..$ Temp   : int [1:30] 78 74 67 84 85 79 82 87 90 87 ...
  ..$ Month  : int [1:30] 6 6 6 6 6 6 6 6 6 6 ...
  ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
 $ 7:'data.frame':        31 obs. of  6 variables:
  ..$ Ozone  : int [1:31] 135 49 32 NA 64 40 77 97 97 85 ...
  ..$ Solar.R: int [1:31] 269 248 236 101 175 314 276 267 272 175 ...
  ..$ Wind   : num [1:31] 4.1 9.2 9.2 10.9 4.6 10.9 5.1 6.3 5.7 7.4 ...
  ..$ Temp   : int [1:31] 84 85 81 84 83 83 88 92 92 89 ...
  ..$ Month  : int [1:31] 7 7 7 7 7 7 7 7 7 7 ...
  ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
 $ 8:'data.frame':        31 obs. of  6 variables:
  ..$ Ozone  : int [1:31] 39 9 16 78 35 66 122 89 110 NA ...
  ..$ Solar.R: int [1:31] 83 24 77 NA NA NA 255 229 207 222 ...
  ..$ Wind   : num [1:31] 6.9 13.8 7.4 6.9 7.4 4.6 4 10.3 8 8.6 ...
  ..$ Temp   : int [1:31] 81 81 82 86 85 87 89 90 90 92 ...
```

```
  ..$ Month  : int [1:31] 8 8 8 8 8 8 8 8 8 8 ...
  ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
 $ 9:'data.frame':        30 obs. of  6 variables:
  ..$ Ozone  : int [1:30] 96 78 73 91 47 32 20 23 21 24 ...
  ..$ Solar.R: int [1:30] 167 197 183 189 95 92 252 220 230 259 ...
  ..$ Wind   : num [1:30] 6.9 5.1 2.8 4.6 7.4 15.5 10.9 10.3 10.9 9.7 ...
  ..$ Temp   : int [1:30] 91 92 93 93 87 84 80 78 75 73 ...
  ..$ Month  : int [1:30] 9 9 9 9 9 9 9 9 9 9 ...
  ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

Then we can take the column means for `Ozone`, `Solar.R`, and `Wind` for each sub-data frame.

```
> lapply(s, function(x) {
+         colMeans(x[, c("Ozone", "Solar.R", "Wind")])
+ })
$`5`
   Ozone  Solar.R     Wind
      NA       NA 11.62258

$`6`
    Ozone   Solar.R      Wind
       NA 190.16667  10.26667

$`7`
     Ozone    Solar.R       Wind
        NA 216.483871   8.941935

$`8`
   Ozone  Solar.R     Wind
      NA       NA 8.793548

$`9`
   Ozone  Solar.R     Wind
      NA 167.4333  10.1800
```

Using `sapply()` might be better here for a more readable output.

```
> sapply(s, function(x) {
+         colMeans(x[, c("Ozone", "Solar.R", "Wind")])
+ })
               5         6          7        8        9
Ozone         NA        NA         NA       NA       NA
Solar.R       NA 190.16667 216.483871       NA 167.4333
Wind    11.62258  10.26667   8.941935 8.793548  10.1800
```

Unfortunately, there are `NA`s in the data so we cannot simply take the means of those variables. However, we can tell the `colMeans` function to remove the `NA`s before computing the mean.

```
> sapply(s, function(x) {
+         colMeans(x[, c("Ozone", "Solar.R", "Wind")],
+                  na.rm = TRUE)
+ })
                5         6          7          8        9
Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

Occasionally, we may want to split an R object according to levels defined in more than one variable. We can do this by creating an interaction of the variables with the `interaction()` function.

```
> x <- rnorm(10)
> f1 <- gl(2, 5)
> f2 <- gl(5, 2)
> f1
 [1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> ## Create interaction of two factors
> interaction(f1, f2)
 [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

With multiple factors and many levels, creating an interaction can result in many levels that are empty.

```
> str(split(x, list(f1, f2)))
List of 10
 $ 1.1: num [1:2] 1.512 0.083
 $ 2.1: num(0)
 $ 1.2: num [1:2] 0.567 -1.025
 $ 2.2: num(0)
 $ 1.3: num 0.323
 $ 2.3: num 1.04
 $ 1.4: num(0)
 $ 2.4: num [1:2] 0.0991 -0.4541
 $ 1.5: num(0)
 $ 2.5: num [1:2] -0.6558 -0.0359
```

Notice that there are 4 categories with no data. But we can drop empty levels when we call the `split()` function.

```
> str(split(x, list(f1, f2), drop = TRUE))
List of 6
 $ 1.1: num [1:2] 1.512 0.083
 $ 1.2: num [1:2] 0.567 -1.025
 $ 1.3: num 0.323
 $ 2.3: num 1.04
 $ 2.4: num [1:2] 0.0991 -0.4541
 $ 2.5: num [1:2] -0.6558 -0.0359
```

tapply

Watch a video of this section[9]

`tapply()` is used to apply a function over subsets of a vector. It can be thought of as a combination of `split()` and `sapply()` for vectors only. I've been told that the "t" in `tapply()` refers to "table", but that is unconfirmed.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

The arguments to `tapply()` are as follows:

- `X` is a vector

[9] https://youtu.be/6YEPWjbk3GA

- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- ... contains other arguments to be passed `FUN`
- `simplify`, should we simplify the result?

Given a vector of numbers, one simple operation is to take group means.

```
> ## Simulate some data
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> ## Define some groups with a factor variable
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
        1         2         3
0.1896235 0.5336667 0.9568236
```

We can also take the group means without simplifying the result, which will give us a list. For functions that return a single value, usually, this is not what we want, but it can be done.

```
> tapply(x, f, mean, simplify = FALSE)
$`1`
[1] 0.1896235

$`2`
[1] 0.5336667

$`3`
[1] 0.9568236
```

We can also apply functions that return more than a single value. In this case, `tapply()` will not simplify the result and will return a list. Here's an example of finding the range of each sub-group.

```
> tapply(x, f, range)
$`1`
[1] -1.869789  1.497041

$`2`
[1] 0.09515213 0.86723879

$`3`
[1] -0.5690822  2.3644349
```

apply()

Watch a video of this section[10]

The `apply()` function is used to a evaluate a function (often an anonymous one) over the margins of an array. It is most often used to apply a function to the rows or columns of a matrix (which is just a 2-dimensional array). However, it can be used with general arrays, for example, to take the average of an array of matrices. Using `apply()` is not really faster than writing a loop, but it works in one line and is highly compact.

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

The arguments to `apply()` are

- `X` is an array
- `MARGIN` is an integer vector indicating which margins should be "retained".
- `FUN` is a function to be applied
- `...` is for other arguments to be passed to `FUN`

Here I create a 20 by 10 matrix of Normal random numbers. I then compute the mean of each column.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)  ## Take the mean of each column
 [1]  0.02218266 -0.15932850  0.09021391  0.14723035 -0.22431309
 [6] -0.49657847  0.30095015  0.07703985 -0.20818099  0.06809774
```

I can also compute the sum of each row.

[10] https://youtu.be/F54ixFPq_xQ

```
> apply(x, 1, sum)   ## Take the mean of each row
 [1] -0.48483448  5.33222301 -3.33862932 -1.39998450  2.37859098
 [6]  0.01082604 -6.29457190 -0.26287700  0.71133578 -3.38125293
[11] -4.67522818  3.01900232 -2.39466347 -2.16004389  5.33063755
[16] -2.92024635  3.52026401 -1.84880901 -4.10213912  5.30667310
```

Note that in both calls to `apply()`, the return value was a vector of numbers.

You've probably noticed that the second argument is either a `1` or a `2`, depending on whether we want row statistics or column statistics. What exactly *is* the second argument to `apply()`?

The `MARGIN` argument essentially indicates to `apply()` which dimension of the array you want to preserve or retain. So when taking the mean of each column, I specify

```
> apply(x, 2, mean)
```

because I want to collapse the first dimension (the rows) by taking the mean and I want to preserve the number of columns. Similarly, when I want the row sums, I run

```
> apply(x, 1, mean)
```

because I want to collapse the columns (the second dimension) and preserve the number of rows (the first dimension).

Col/Row Sums and Means

For the special case of column/row sums and column/row means of matrices, we have some useful shortcuts.

- `rowSums = apply(x, 1, sum)`
- `rowMeans = apply(x, 1, mean)`
- `colSums = apply(x, 2, sum)`
- `colMeans = apply(x, 2, mean)`

The shortcut functions are heavily optimized and hence are *much* faster, but you probably won't notice unless you're using a large matrix. Another nice aspect of these functions is that they are a bit more descriptive. It's arguably more clear to write `colMeans(x)` in your code than `apply(x, 2, mean)`.

Other Ways to Apply

You can do more than take sums and means with the `apply()` function. For example, you can compute quantiles of the rows of a matrix using the `quantile()` function.

```
> x <- matrix(rnorm(200), 20, 10)
> ## Get row quantiles
> apply(x, 1, quantile, probs = c(0.25, 0.75))
          [,1]       [,2]      [,3]       [,4]       [,5]        [,6]
25% -1.0884151 -0.6693040 0.2908481 -0.4602083 -1.0432010 -1.12773555
75%  0.1843547  0.8210295 1.3667301  0.4424153  0.3571219  0.03653687
          [,7]       [,8]       [,9]     [,10]      [,11]      [,12]
25% -1.4571706 -0.2406991 -0.3226845 -0.329898 -0.8677524 -0.2023664
75% -0.1705336  0.6504486  1.1460854  1.247092  0.4138139  0.9145331
         [,13]      [,14]      [,15]        [,16]      [,17]      [,18]
25% -0.9796050 -1.3551031 -0.1823252 -1.260911898 -0.9954289 -0.3767354
75%  0.5448777 -0.5396766  0.7795571  0.002908451  0.4323192  0.7542638
         [,19]      [,20]
25% -0.8557544 -0.7000363
75%  0.5440158  0.5432995
```

Notice that I had to pass the `probs = c(0.25, 0.75)` argument to `quantile()` via the `...` argument to `apply()`.

For a higher dimensional example, I can create an array of 2×2 matrices and the compute the average of the matrices in the array.

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
          [,1]       [,2]
[1,] 0.1681387 -0.1039673
[2,] 0.3519741 -0.4029737
```

In the call to `apply()` here, I indicated via the `MARGIN` argument that I wanted to preserve the first and second dimensions and to collapse the third dimension by taking the mean.

There is a faster way to do this specific operation via the `colMeans()` function.

```
> rowMeans(a, dims = 2)    ## Faster
          [,1]       [,2]
[1,] 0.1681387 -0.1039673
[2,] 0.3519741 -0.4029737
```

In this situation, I might argue that the use of `rowMeans()` is less readable, but it is substantially faster with large arrays.

mapply()

Watch a video of this section[11]

The `mapply()` function is a multivariate apply of sorts which applies a function in parallel over a set of arguments. Recall that `lapply()` and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what `mapply()` is for.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

The arguments to `mapply()` are

- `FUN` is a function to apply
- `...` contains R objects to apply over
- `MoreArgs` is a list of other arguments to `FUN`.
- `SIMPLIFY` indicates whether the result should be simplified

The `mapply()` function has a different argument order from `lapply()` because the function to apply comes first rather than the object to iterate over. The R objects over which we apply the function are given in the `...` argument because we can apply over an arbitrary number of R objects.

For example, the following is tedious to type

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

With `mapply()`, instead we can do

```
>  mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

This passes the sequence `1:4` to the first argument of `rep()` and the sequence `4:1` to the second argument.

Here's another example for simulating randon Normal variables.

[11] https://youtu.be/z8jC_h7S0VE

```
> noise <- function(n, mean, sd) {
+       rnorm(n, mean, sd)
+ }
> ## Simulate 5 randon numbers
> noise(5, 1, 2)
[1] -0.5196913  3.2979182 -0.6849525  1.7828267  2.7827545
>
> ## This only simulates 1 set of numbers, not 5
> noise(1:5, 1:5, 2)
[1] -1.670517  2.796247  2.776826  5.351488  3.422804
```

Here we can use `mapply()` to pass the sequence `1:5` separately to the `noise()` function so that we can get 5 sets of random numbers, each with a different length and mean.

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 0.8260273

[[2]]
[1] 4.764568 2.336980

[[3]]
[1] 4.6463819 2.5582108 0.9412167

[[4]]
[1]  3.978149  1.550018 -1.192223  6.338245

[[5]]
[1] 2.826182 1.347834 6.990564 4.976276 3.800743
```

The above call to `mapply()` is the same as

```
> list(noise(1, 1, 2), noise(2, 2, 2),
+      noise(3, 3, 2), noise(4, 4, 2),
+      noise(5, 5, 2))
[[1]]
[1] 0.644104

[[2]]
[1] 1.148037 3.993318

[[3]]
```

```
[1]  4.4553214 -0.4532612  3.7067970

[[4]]
[1]  5.4536273  5.3365220 -0.8486346  3.5292851

[[5]]
[1] 8.959267 6.593589 1.581448 1.672663 5.982219
```

Vectorizing a Function

The `mapply()` function can be use to automatically "vectorize" a function. What this means is that it can be used to take a function that typically only takes single arguments and create a new function that can take vector arguments. This is often needed when you want to plot functions.

Here's an example of a function that computes the sum of squares given some data, a mean parameter and a standard deviation. The formula is $\sum_{i=1}^n (x_i-\mu)2/\sigma^2$.

```
> sumsq <- function(mu, sigma, x) {
+         sum(((x - mu) / sigma)^2)
+ }
```

This function takes a mean `mu`, a standard deviation `sigma`, and some data in a vector `x`.

In many statistical applications, we want to minimize the sum of squares to find the optimal `mu` and `sigma`. Before we do that, we may want to evaluate or plot the function for many different values of `mu` or `sigma`. However, passing a vector of `mu`s or `sigma`s won't work with this function because it's not vectorized.

```
> x <- rnorm(100)       ## Generate some data
> sumsq(1:10, 1:10, x)  ## This is not what we want
[1] 110.2594
```

Note that the call to `sumsq()` only produced one value instead of 10 values.

However, we can do what we want to do by using `mapply()`.

```
> mapply(sumsq, 1:10, 1:10, MoreArgs = list(x = x))
 [1] 196.2289 121.4765 108.3981 104.0788 102.1975 101.2393 100.6998
 [8] 100.3745 100.1685 100.0332
```

There's even a function in R called `Vectorize()` that automatically can create a vectorized version of your function. So we could create a `vsumsq()` function that is fully vectorized as follows.

```
> vsumsq <- Vectorize(sumsq, c("mu", "sigma"))
> vsumsq(1:10, 1:10, x)
 [1] 196.2289 121.4765 108.3981 104.0788 102.1975 101.2393 100.6998
 [8] 100.3745 100.1685 100.0332
```

Pretty cool, right?

Summary

- The loop functions in R are very powerful because they allow you to conduct a series of operations on data using a compact form
- The operation of a loop function involves iterating over an R object (e.g. a list or vector or matrix), applying a function to each element of the object, and the collating the results and returning the collated results.
- Loop functions make heavy use of anonymous functions, which exist for the life of the loop function but are not stored anywhere
- The `split()` function can be used to divide an R object in to subsets determined by another variable which can subsequently be looped over using loop functions.