# R session 6: Avoiding loops – the split-apply-combine tools of `dplyr`

*ESS 211*

## Contents



## 1 Introduction and other resources

The `dplyr` package has rapidly become one of `R`'s most popular, and for good reason. The package has a variety of functions for data wrangling and crunching, but the main thing we hope you get out of it is how to use `group_by()`, which allows you to apply a function to sub-groups of a data.frame, rather than just over the entirety of rows or columns as you've done with `apply()`. You may already be familir with this concept if you've come across `tapply()` (don't worry if you haven't); `dplyr` offers what is essentially a more convenient, more versatile, and faster version of `tapply()`.

We're trying to present the essentials here, but there are many more details and features that we don't have time to get into. For more practice and examples, we recommend:

1) The dplyr and tidyr vignettes (Google them, or run `vignette("introduction", package="dplyr")` or `vignette("tidy-data", package="tidyr")` at the command line.
2) Rstudio's data wrangling cheatsheet (link here; also, the pdf is on Coursework)
3) DataCamp's dplyr chapter in the Intermediate R course

# 2 The problem - applying a function to subsections data

Load the following packages and review the `airquality` dataset, which contains daily values of ozone, radiation, wind speed, and temperature during five months.

```
> library(tidyr)
> library(dplyr)
> data(airquality) # load the dataset into the workspace
> head(airquality) # look at its first few rows to understand its structure
```

```
  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
```

Say we're interested in each variables average over time. We could get each variable's grand mean over all months and days with `apply(airquality,2,mean)`, but what if we want averages for each individual month? The `apply()` function has no way of saying "take the mean of each column, but just for the rows where Month==5, and then for the rows where Month==6, . . . ", which is what we want.

## 2.1 A loop as one solution

The straightforward solution is to loop over all the unique months in the data.frame. At each loop iteration, we pick out all the rows that belong to a month, compute the averages, and bind the new result onto the previous ones.

```
> # Sloppy and slow, but easy to write (which can be a virtue!)
> monthly.avgs = c() # initialize as an empty object
> months = unique(airquality$Month)
> for (month in months) {
+   month.data = airquality[airquality$Month==month,] # split
+   avgs = apply(month.data,2,mean,na.rm=T) # apply
+   monthly.avgs = rbind(monthly.avgs, avgs) # combine
+ }
> monthly.avgs
```

```
        Ozone  Solar.R      Wind     Temp Month  Day
avgs 23.61538 181.2963 11.622581 65.54839     5 16.0
avgs 29.44444 190.1667 10.266667 79.10000     6 15.5
avgs 59.11538 216.4839  8.941935 83.90323     7 16.0
avgs 59.96154 171.8571  8.793548 83.96774     8 16.0
avgs 31.44828 167.4333 10.180000 76.90000     9 15.5
```

More rigorously, we would initialize `monthly.avgs` as a matrix with the exact dimensions that we know it needs, and fill up the appropriate indices at each step. This is much more efficient if you have many more loop iterations, and/or if each loop iteration involves a costly step, such as a call to `rbind()`.

```
> # Takes a bit more time and care to write, but will run much faster
> months = unique(airquality$Month)
> n_months = length(months)
> n_vars = ncol(airquality)
> monthly.avgs = matrix(nrow=n_months, ncol=n_vars)
> t1=Sys.time()
> for (i in 1:n_months) {
+   month.data = airquality[airquality$Month==months[i],] # split
+   avgs = apply(month.data,2,mean,na.rm=T) # apply
+   monthly.avgs[i,] = avgs # combine
+ }
> monthly.avgs
```

```
         [,1]     [,2]      [,3]     [,4] [,5] [,6]
[1,] 23.61538 181.2963 11.622581 65.54839    5 16.0
[2,] 29.44444 190.1667 10.266667 79.10000    6 15.5
[3,] 59.11538 216.4839  8.941935 83.90323    7 16.0
[4,] 59.96154 171.8571  8.793548 83.96774    8 16.0
[5,] 31.44828 167.4333 10.180000 76.90000    9 15.5
```

### 2.2 Downsides to this approach

You won't notice any difference between these two methods in this case, but try changing `months` to a vector with 1,000 elements, and time each method using `system.time()`. By what order of magnitude is the `rbind()` way slower? What would that imply if you had to loop over 10,000 elements, or a million?

That said,though, the ugly first method is totally fine for rapid prototyping of smaller tasks, and you should feel free to use it until speed becomes an issue. "Premature optimization is the root of all evil." - Donald Knuth

So we have two variants of a loop to perform this task. The first is a little cumbersome to write, and agonizingly slow for big loops. The other is much faster (but still not fast enough for bigger jobs!), but even more of a pain to write. We can do better. Three basic things happen in either method:

1) We **split** the data. At each iteration of the loop, we took out just one group of the original data.frame - the group corresponding to a given month.
2) We **apply** a function (`mean()` in this case) to each of those groups.
3) We **combine** the results of each group operation together into one data structure.

The beauty of `dplyr` is that it gives you a way to split, apply, and combine in a way that's much easier to write *and* that will run much faster than any loop.

## 3 The `summarise()` and `mutate()` functions

### 3.1 The basics

Before getting to how `dplyr` works on multiple groups, you need to know its two basic tools for applying an arbitrary function to any single group.

1) A `mutate()` operation takes a column and transforms each of its elements, resulting in a new column with exactly the same number of elements as before. Examples: squaring all elements in a column, or converting units from Celsius to Fahrenheit.

2) A `summarise()` operation collapses a column into a single value, such as taking the mean or standard deviation.

First we'll create a mini version of `airquality` in which only the first day of each month has been kept. The point in doing that is to be able to see the whole data.frame - and the results of any operations on it - in one glance. This is sometimes called making a "toy" dataset, and it's an enormously helpful way to develop and debug your code.

```
> air_day1 = airquality[airquality$Day==1,]
> print(air_day1)
```

```
    Ozone Solar.R Wind Temp Month Day
1      41     190  7.4   67     5   1
32     NA     286  8.6   78     6   1
62    135     269  4.1   84     7   1
93     39      83  6.9   81     8   1
124    96     167  6.9   91     9   1
```

And here are some basic examples of using existing columns to create new ones. The first argument of either `mutate()` or `summarise()` is the data.frame you're dealing with. All subsequent arguments, separated by commas, are the new columns you're creating.

```
> mutate(air_day1, Ozone_squared=Ozone^2, Temp_C=(Temp-32)*5/9)
```

```
  Ozone Solar.R Wind Temp Month Day Ozone_squared   Temp_C
1    41     190  7.4   67     5   1          1681 19.44444
2    NA     286  8.6   78     6   1            NA 25.55556
3   135     269  4.1   84     7   1         18225 28.88889
4    39      83  6.9   81     8   1          1521 27.22222
5    96     167  6.9   91     9   1          9216 32.77778
```

```
> summarise(air_day1, Ozone_mean=mean(Ozone), Temp_sd=sd(Temp)) # can you remove the NA's?
```

```
  Ozone_mean Temp_sd
1         NA 8.81476
```

Note how you simply refer to the columns by name, without quotes or dollar signs. Also note how `mutate()` left the original columns intact and simply added the new ones, whereas `summarise()` only produced the newly requested columns. Think about why that makes sense!

## 3.2 Using intermediate results

One great feature of these functions is that a new column can be created from other new columns, even in the same line of code. For example, the saturation pressure function from the first assignment, `e0()`, requires that its temperature argument be in Celsius, so we can do the following:

```
> e0 = function(temp) 0.6108*exp(17.27*temp/(temp+237.3))
> mutate(air_day1, Temp_C=(Temp-32)*5/9, sat_pressure=e0(Temp_C))
```

```
  Ozone Solar.R Wind Temp Month Day   Temp_C sat_pressure
1    41     190  7.4   67     5   1 19.44444     2.259066
2    NA     286  8.6   78     6   1 25.55556     3.274130
3   135     269  4.1   84     7   1 28.88889     3.980029
4    39      83  6.9   81     8   1 27.22222     3.612087
5    96     167  6.9   91     9   1 32.77778     4.967786
```

```
> mutate(air_day1, sat_pressure=e0(Temp_C), Temp_C=(Temp-32)*5/9) # but order does matter!
```

```
Error in e0(Temp_C): object 'Temp_C' not found
```

The new column `Temp_C` was added and became immediately available for use as an argument to any subsequent columns, even within the same `mutate()` call. Using traditional data.frame column creation, this would have required two steps like so:

```
> air_day1$Temp_C = (air_day1$Temp-32)*5/9
> air_day1$sat_pressure = e0(air_day1$Temp_C)
```

### 3.3 Allowable sizes of columns produced by `mutate()` and `summarise()`.

If you use `mutate()` with a function that returns a single value, e.g. `mean()`, that's ok. It will just recycle that single value to fill the column. There are no other options, though; an operation within `mutate()` must produce either a single value, or a vector with as many elements as the column.

```
> mutate(air_day1, Temp_mean=mean(Temp)) # ok
```

```
  Ozone Solar.R Wind Temp Month Day   Temp_C sat_pressure Temp_mean
1    41     190  7.4   67     5   1 19.44444     2.259066      80.2
2    NA     286  8.6   78     6   1 25.55556     3.274130      80.2
3   135     269  4.1   84     7   1 28.88889     3.980029      80.2
4    39      83  6.9   81     8   1 27.22222     3.612087      80.2
5    96     167  6.9   91     9   1 32.77778     4.967786      80.2
```

```
> mutate(air_day1, gt80=which(Temp>80)) # error
```

```
Error in eval(expr, envir, enclos): wrong result size (3), expected 5 or 1
```

With `summarise()`, these is no ambiguity; the result must be a single value.

**Exercise:** With `mutate()`, add a column of temperature anomalies (differences from the mean) to `air_day1`. In the same line, use this column to produce two other columns: the "standardized anomalies" (anomalies divided by the standard deviation), and the root mean squared error. Then, with `summarise()`, find how many of these standardized anomalies are greater in absolute value than 1.

### 3.4 transform *all* columns - `summarise_each()` and `mutate_each()`

So far we've seen how to create specific new columns by explicitly naming them. Only knowing what we know so far, taking the mean of every relevant column in `air_day1` (everything except `Day` and `Month`) would require

```
> summarise(air_day1, Ozone=mean(Ozone), Temp=mean(Temp),
+           Solar.R=mean(Solar.R), Wind=mean(Solar.R))
```

```
  Ozone Temp Solar.R Wind
1    NA 80.2     199 39.8
```

This is awkward, and infeasible if there's a large number of columns. This is where `summarise_each()` and `mutate_each()` come in. They let you specify the function you wish to apply, and the columns you wish to apply it to.

```
> summarise_each(air_day1, "mean") # all columns
  Ozone Solar.R Wind Temp Month Day   Temp_C sat_pressure
1    NA     199 6.78 80.2     7   1 26.77778     3.618619
> summarise_each(air_day1, "mean", Temp, Wind) # summarise() on the Temp and Wind columns
  Temp Wind
1 80.2 6.78
> summarise_each(air_day1, "mean", -c(Month,Day)) # all columns EXCEPT Month and Day
  Ozone Solar.R Wind Temp   Temp_C sat_pressure
1    NA     199 6.78 80.2 26.77778     3.618619
```

Note how we got an NA in Ozone, because there was an NA in the original column, and `mean()` returns NA if any elements are NA. We can handle that (and more) by defining our own function instead of simply naming an existing one. The syntax for this is as follows:

```
> mutate_each(air_day1, funs(./2)) # divide every column by 2
```

```
  Ozone Solar.R Wind Temp Month Day    Temp_C sat_pressure
1  20.5    95.0 3.70 33.5   2.5 0.5  9.722222     1.129533
2    NA   143.0 4.30 39.0   3.0 0.5 12.777778     1.637065
3  67.5   134.5 2.05 42.0   3.5 0.5 14.444444     1.990014
4  19.5    41.5 3.45 40.5   4.0 0.5 13.611111     1.806043
5  48.0    83.5 3.45 45.5   4.5 0.5 16.388889     2.483893
```

The `funs()` wrapper lets `mutate_each()` know that we're defining a function to apply to all columns, and the period is a placeholder argument, which gets filled with each column in turn. Think of the period as standing for `Ozone` when `Ozone` is getting divided by 2, `Temp` when `Temp` is getting divided by 2, etc. This is the best way to include optional arguments such as `na.rm=T`.

```
> summarise_each(air_day1, funs(mean(., na.rm=T)), -c(Month,Day))
```

```
  Ozone Solar.R Wind Temp   Temp_C sat_pressure
1 77.75     199 6.78 80.2 26.77778     3.618619
```

This line says to apply `mean(., na.rm=T)`, where `.` is the current column, to all columns except `Month` and `Day`. It is essentially shorthand for:

```
> summarise(air_day1, Ozone=mean(Ozone, na.rm=T), Solar.R=mean(Solar.R, na.rm=T),
+           Wind=mean(Wind, na.rm=T), Temp=mean(Temp, na.rm=T) )
```

# 4 Grouped operations with `group_by()`

Going back to our example of getting monthly averages, we know that if we had the data split up into a group for each month, like this,

```
  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

  Ozone Solar.R Wind Temp Month Day
1    NA     286  8.6   78     6   1
2    NA     287  9.7   74     6   2
3    NA     242 16.1   67     6   3

  Ozone Solar.R Wind Temp Month Day
1   135     269  4.1   84     7   1
2    49     248  9.2   85     7   2
3    32     236  9.2   81     7   3
```

we could then use `summarise()` on each group. This is where `group_by()` comes in. It creates these groups for us, so that when we use `summarise()` or `mutate()`, we're not doing so on the entire data.frame, but rather on each individual group of the data.frame. Here's how it works:

```
>  # group airquality according to unique values of its Month column
> air_grouped = group_by(airquality, Month)
>
> # summarise Ozone within each of these groups
> summarise(air_grouped, Ozone_mean = mean(Ozone,na.rm=T)) # take care of NA's
Source: local data frame [5 x 2]

  Month Ozone_mean
1     5   23.61538
2     6   29.44444
3     7   59.11538
4     8   59.96154
5     9   31.44828
```

The first line grouped the `airquality` data.frame by its `Month` column, which means it invisibly divided `airquality` into multiple sub-data.frames, each one corresponding to a unique value of `Month`, exactly as seen above. Even though `air_grouped` and `airquality` have exactly the same dimensions and exactly the same data (check this), there's an important difference. `air_grouped` is like a version of `airqaulity` that has made a mental note to itself, say9ing that any subsequent operations should apply to each of the monthly sub-data.frames, not the data.frame as a whole.

**Exercise:** Find the month with the biggest temperature range. Find how many days in each month had exactly the same temperature.

# 5 Subsetting rows and columns with `filter()` and `select()`

These functions allow you subset your data by rows and columns, respectively. In `filter()`, you use logical conditions to specify which rows you want to keep. Multiple conditions are separated by commas, so you can think of each comma as a logical `and`.

```
> filter(airquality, Temp==67) # keep rows where Temp equals 67
```

```
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    23      13 12.0   67     5  28
3    NA     242 16.1   67     6   3
4    18     224 13.8   67     9  17
```

```
> filter(airquality, Temp==67, Month==5) # keep rows where Temp==67 AND Month==5
```

```
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    23      13 12.0   67     5  28
```

```
> # equivalent "manual" syntax
> airquality[airquality$Temp==67,]
> airquality[airquality$Temp==67 & airquality$Month==5,]
```

In `select()`, you name the columns you want to keep, or use a minus sign to indicate the columns to omit.

```
> select(air_day1, Ozone, Solar.R) # keep columns Ozone and Solar.R
```

```
    Ozone Solar.R
1      41     190
32     NA     286
62    135     269
93     39      83
124    96     167
```

```
> select(air_day1, -Wind, -Temp) # keep all columns except Wind and Temp
```

```
    Ozone Solar.R Month Day   Temp_C sat_pressure
1      41     190     5   1 19.44444     2.259066
32     NA     286     6   1 25.55556     3.274130
62    135     269     7   1 28.88889     3.980029
93     39      83     8   1 27.22222     3.612087
124    96     167     9   1 32.77778     4.967786
```

For one-off operations, they don't seem to offer a huge value-add over other syntax options. Their real advantage comes into play when chaining them together in conjuntion with other functions such as `group_by()` and `summarise()`, as seen in the next section.

# 6 Chaining operations together with the %>% ("then") operator

You already know how to pass the results of function calls to other functions (as a simple example, `mean(rnorm(10))` generates 10 random numbers, and those 10 numbers then become the argument to `mean()`). In the `airquality` data, if you want to take just the JJA averages of each month, you would

- filter out months not belonging to 6, 7, or 8

- pass this result to `group_by()` and group on the `Month` column
- pass this result to `summarise_each()` with the appropriate arguments

Read the expression in the following code from the inside out to make sure you see how it's accomplishing the three steps.

```
> # 1) Keep only rows where Month equals either 6, 7, or 8
> # 2) group the result by month
> # 3) take the by-month means of each columns
> summarise_each(group_by(filter(airquality,Month %in% 6:8), Month), funs(mean(.,na.rm=T)))
```

```
Source: local data frame [3 x 6]

  Month    Ozone  Solar.R      Wind     Temp  Day
1     6 29.44444 190.1667 10.266667 79.10000 15.5
2     7 59.11538 216.4839  8.941935 83.90323 16.0
3     8 59.96154 171.8571  8.793548 83.96774 16.0
```

This is a valid way of chaining these operations together, but it's very confusing to read. The following is much more readable:

```
> df1 = filter(airquality, Month %in% 6:8)
> df2 = group_by(df1, Month)
> df3 = summarise_each(df2, funs(mean(.,na.rm=T)))
> print(df3)
```

```
Source: local data frame [3 x 6]

  Month    Ozone  Solar.R      Wind     Temp  Day
1     6 29.44444 190.1667 10.266667 79.10000 15.5
2     7 59.11538 216.4839  8.941935 83.90323 16.0
3     8 59.96154 171.8571  8.793548 83.96774 16.0
```

This is much more intelligble, but also leaves these nondescript, intermediate variables clogging up your workspace. This can cause problems when you later forget what they are, accidentally overwrite them, and wrongly re-use them (not that this has ever, ever happened to any of your instructors).

The `%>%` ("then") operator solves both problems. It lets you chain functions together without nesting them (for readability), but also eliminates the need for temporary variables.

```
> result = filter(airquality, Month %in% c(6,7,8)) %>% # keep months c(6,7,8), then ...
+          group_by(Month) %>% # group by Month, then ...
+          summarise_each(funs(mean(.,na.rm=T))) # summarise each column
> print(result)
```

```
Source: local data frame [3 x 6]

  Month    Ozone  Solar.R      Wind     Temp  Day
1     6 29.44444 190.1667 10.266667 79.10000 15.5
2     7 59.11538 216.4839  8.941935 83.90323 16.0
3     8 59.96154 171.8571  8.793548 83.96774 16.0
```

Note how the `group_by()` and `summarise_each()` functions no longer need their first argument, the data.frame bieng worked on. That's because the `%>%` automatically implies that the result of the previous step *is* that data.frame. Splitting up each operation into separate lines wasn't necessary, but it can help readability a lot.

# 7 Reshaping between "wide" and "long" data.frames with `tidyr`

So now we have some great tools for applying functions to specific groups within data.frames, with clean code and no loops, but what if you the variables that define your groups are spread out over multiple columns? For example, what if you want to group by `Month`, because someone instead gave data that look like this:

```
Source: local data frame [4 x 6]

  variable       May       Jun        Jul        Aug       Sep
1    Ozone  23.61538  29.44444  59.115385  59.961538  31.44828
2  Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
3     Wind  11.62258  10.26667   8.941935   8.793548  10.18000
4     Temp  65.54839  79.10000  83.903226  83.967742  76.90000
```

This is called "wide" format, because a variable that could be in just one column is instead spread out over several, thus making the data.frame wider. You can't group by month with this data.frame, since `group_by()` needs the name of the column on which to group, and there's no month column. For that, you'd rather the data.frame be in "long" format, as we've already seen it:

```
Source: local data frame [5 x 5]

  Month    Ozone  Solar.R      Wind     Temp
1   May 23.61538 181.2963 11.622581 65.54839
2   Jun 29.44444 190.1667 10.266667 79.10000
3   Jul 59.11538 216.4839  8.941935 83.90323
4   Aug 59.96154 171.8571  8.793548 83.96774
5   Sep 31.44828 167.4333 10.180000 76.90000
```

## 7.1 `gather()` - gather multiple columns into one column

We'll start with our monthly avgs data.frame, `avgs`, and rename the months for clarity.

```
> avgs = airquality %>% group_by(Month) %>% summarise_each(funs(mean(.,na.rm=T)), -Day)
```

There's nothing wrong with this format per se, but an alternative representation is to instead have a `variable` column, whose values are `Ozone`, `Solar.R`, `Wind`, and `Temp`. The remaining column would then contain the values for each month-variable combination. Here's how `gather()` is used to do that.

```
> long = gather(avgs, variable, value, Ozone, Solar.R, Wind, Temp)
> print(long)
```

```
Source: local data frame [20 x 3]

   Month variable     value
1      5    Ozone 23.615385
```

```
2       6     Ozone  29.444444
3       7     Ozone  59.115385
4       8     Ozone  59.961538
5       9     Ozone  31.448276
6       5   Solar.R 181.296296
7       6   Solar.R 190.166667
8       7   Solar.R 216.483871
9       8   Solar.R 171.857143
10      9   Solar.R 167.433333
11      5      Wind  11.622581
12      6      Wind  10.266667
13      7      Wind   8.941935
14      8      Wind   8.793548
15      9      Wind  10.180000
16      5      Temp  65.548387
17      6      Temp  79.100000
18      7      Temp  83.903226
19      8      Temp  83.967742
20      9      Temp  76.900000
```

This line of code gathered all the values in the columns `Ozone`, `Solar.R`, `Wind`, and `Temp`, and put those values in a new column named `value` (we could have named it anything we like). It also created another new column, `variable`, which contains the names of the gathered columns (if this column didn't exist, you wouldn't have any way of knowing which values paired with which variables). The arguments are:

1) the data.frame being reshaped
2) the name of the new column containing the names of the gathered columns
3) the name of the new column containing the corresponding values
4) the names of the columns being gathered, separated by commas

Since we can use a minus sign to select columns according to those we *don't* want, a more compact way of writing this line of code is `long = gather(avgs, variable, value, -Month)`.

### 7.1.1 Good or bad?

Some would say this is the "proper" or "tidy" shape for these data, since any value in these data is completely specified by its combination of `Month` and `variable`, and each row of the data.frame now correpsonds to a unique month-variable combination. However, there are probably better ways to pass the time than arguing over it, since whether you'd prefer the data in this or the previous format will depend on the application. The important thing is having the tools to reshape your data in a way the works for you.

## 7.2 `spread()` - spread one column into multiple columns

To see how `spread()` works, we'll create a separate column for each month. Its three arguments are:

1) the data.frame being reshaped)
2) the name of the column being spread into separate columns
3) the name of the column with the values with the corresponding values

In our case, arguments (2) and (3) are `Month` and `value`.

```
> wide = spread(long, Month, value)
> print(wide)
```

```
Source: local data frame [4 x 6]

  variable         5         6          7          8        9
1    Ozone  23.61538  29.44444  59.115385  59.961538  31.44828
2  Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
3     Wind  11.62258  10.26667   8.941935   8.793548  10.18000
4     Temp  65.54839  79.10000  83.903226  83.967742  76.90000
```

You can then rename the month columns with `names(avgs)[-1] = c('May','Jun','Jul','Aug','Sep')`. This might be preferable since having integers as column names can create ambiguity. For example, do you mean column 5 as in the fifth column, or the column named 5? They're not the same.

**Exercise:** Use `gather()` to convert the above result back to its original shape (i.e. with a `Month` column).

**Exercise:** Explain the result of `spread(avgs, Month, Ozone)`. How does that explain why we gathered the variables in `avgs` before spreading the months?