Quick Howto 2

Jeff Newmiller

14 May 2016

The `ave` *function*

If you have an existing vector of data

```
X <- c( 1, 2, 3, 4, 5, 6, 7, 8 )
```

and another vector that identifies which *groups* the values in the first vector are in

```
G <- c( 1, 1, 1, 1, 2, 2, 2, 1 )
```

We could find the mean for group 1

```
X[ 1 == G ]
```

```
## [1] 1 2 3 4 8
```

```
mean( X[ 1 == G ] )
```

```
## [1] 3.6
```

and for group 2

```
X[ 2 == G ]
```

```
## [1] 5 6 7
```

```
mean( X[ 2 == G ] )
```

```
## [1] 6
```

but if we want to keep our original vector lengths and do both, we can use `ave`:

```
ave( X, G )
```

```
## [1] 3.6 3.6 3.6 3.6 6.0 6.0 6.0 3.6
```

so those vectors would all fit in a data frame:

```
DF1 <- data.frame( Pos = seq_along( X ), X, G )
DF1$GroupMeans <- with( DF1, ave( X, G ) )
DF1
```

```
##   Pos X G GroupMeans
## 1   1 1 1        3.6
## 2   2 2 1        3.6
## 3   3 3 1        3.6
## 4   4 4 1        3.6
## 5   5 5 2        6.0
## 6   6 6 2        6.0
## 7   7 7 2        6.0
## 8   8 8 1        3.6
```

as opposed to getting the means as one result per group:

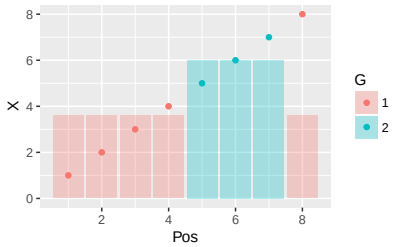

Figure 1: Group Mean by Ave

```
# using base R
# remember a data frame is a list, and indexing without a comma
# returns another data frame/list
DF2a <- aggregate( DF1[ "X" ] # list of columns to aggregate
                 , DF1[ "G" ] # list of columns to group by
                 , FUN = mean # what to do to each column
                 )
DF2a
```

```
##   G   X
## 1 1 3.6
## 2 2 6.0
```

```
# using dplyr
library( dplyr )
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
DF2b <- (   DF1
        %>% group_by( G )
        %>% summarise( X = mean( X ) )
        %>% as.data.frame
        )
DF2b
```

```
##   G   X
## 1 1 3.6
## 2 2 6.0
```

Using `diff` *to find transitions*

Suppose we know some (fictional) water stage values:

```
DF3 <- read.table( text =
"Dt         Stage
1990-01-01    9.0
1990-01-02   10.0
1990-01-03   11.0
1990-01-04   11.0
1990-01-05   10.0
1990-01-06    9.0
1990-01-07    8.0
1990-01-08    9.0
1990-01-09   10.0
1990-01-10   11.0
1990-01-11   12.0
1990-01-12   11.0
1990-01-13   10.0
1990-01-14    9.0
1990-01-15    8.0
", header = TRUE, as.is = TRUE )
DF3$Dt <- as.Date( DF3$Dt )
```

and we want to identify and group contiguous time intervals when stage is greater than 10.

Start by identifying when the level is exceeded:

```
DF3$F <- 10 < DF3$Stage
```

Now identify the rising edge of `F`:

```
DF3$FStart <- 1 == diff( c( 0, DF3$F ) )
```

Notice that `diff` returns a vector one shorter than the input because it makes no assumptions about how you want the beginning or end values treated. Extending `F` with a zero at the beginning allows a data set that starts out in flood conditions to be marked as starting the flood at the beginning of the data.

Now, for each start of flood, start a new level:

```
DF3$FNum <- cumsum( DF3$FStart )
```

Now shorten `FNum` to just the records when the level was exceeded:

```
DF3$FId <- with( DF3, FNum * F )
```

Now we can make a factor variable for identifying flood records:

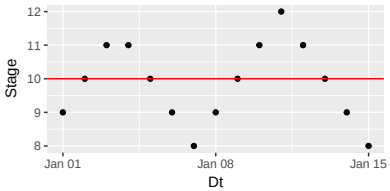

Figure 2: Raw (Fake) Stage Data

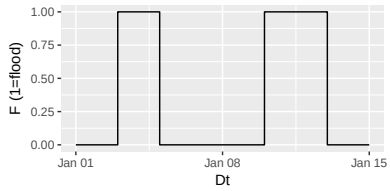

Figure 3: Level Exceeded Trend

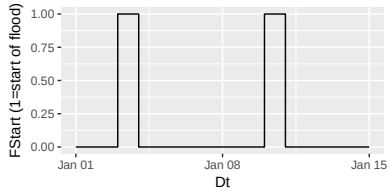

Figure 4: Start of Level

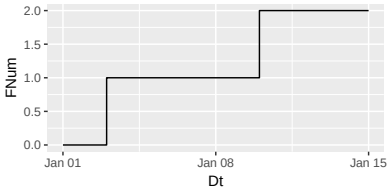

Figure 5: Level Counter

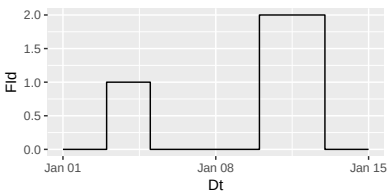

Figure 6: Level Counter

```
FloodIds <- unique( DF3$FId )
DF3$FloodStatus <- factor( DF3$FId
                         , levels = FloodIds
                         , labels = c( "No Flood"
                                     , paste( "Flood"
                                            , FloodIds[ -1 ]
                                            )
                                     )
                         )
```

Table 1: Table 1. Flood Identification

Dt	Stage	F	FStart	FNum	FId	FloodStatus
1990-01-01	9	FALSE	FALSE	0	0	No Flood
1990-01-02	10	FALSE	FALSE	0	0	No Flood
1990-01-03	11	TRUE	TRUE	1	1	Flood 1
1990-01-04	11	TRUE	FALSE	1	1	Flood 1
1990-01-05	10	FALSE	FALSE	1	0	No Flood
1990-01-06	9	FALSE	FALSE	1	0	No Flood
1990-01-07	8	FALSE	FALSE	1	0	No Flood
1990-01-08	9	FALSE	FALSE	1	0	No Flood
1990-01-09	10	FALSE	FALSE	1	0	No Flood
1990-01-10	11	TRUE	TRUE	2	2	Flood 2
1990-01-11	12	TRUE	FALSE	2	2	Flood 2
1990-01-12	11	TRUE	FALSE	2	2	Flood 2
1990-01-13	10	FALSE	FALSE	2	0	No Flood
1990-01-14	9	FALSE	FALSE	2	0	No Flood
1990-01-15	8	FALSE	FALSE	2	0	No Flood

The factor column can be used to identify flood stage points with alternate graphical features such as color, fill, alpha (transparency) or shape.

You can also wrap this up as a function so you don't end up with a bunch of extra columns in your data:

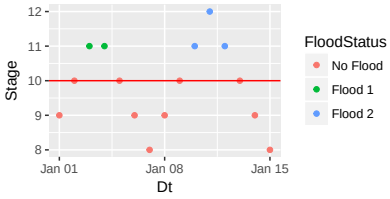

Figure 7: Flood Marking

```
MarkFloodStatus <- function( stage, floodlevel ) {
    F <-floodlevel < stage
    FStart <- 1 == diff( c( 0, F ) )
    FNum <- cumsum( FStart )
    FId <- FNum * F
    FloodIds <- unique( FId )
    FloodStatus <- factor( FId
                         , levels = FloodIds
                         , labels = c( "No Flood"
```

```
                                    , paste( "Flood"
                                           , FloodIds[ -1 ]
                                           )
                                    )
                            )
    FloodStatus
}
DF3b <- DF3[ 1:2 ]  # remove extra columns
DF3b$FloodStatus <- MarkFloodStatus( DF3b$Stage, 10 )
str( DF3b )
```

```
## 'data.frame':    15 obs. of  3 variables:
##  $ Dt         : Date, format:  ...
##  $ Stage      : num  9 10 11 11 10 9 8 9 10 11 ...
##  $ FloodStatus: Factor w/ 3 levels "No Flood","Flood 1",..: 1 1 2 2 1 1 1 1 1 3 ...
```

The ***tidyr*** *package*

Tables that use both rows and columns to locate data values are quite useful in many situations. However, the structure they impose is not always ideal for a couple of reasons. One is that every combination of row and column specification has a place, even if no data are available for that combination. Thus it is often convenient to store data in one column and use values entered in "key" rows to locate those values. A common term for these single-column data tables is *long*, with the alternative table referred to as *wide*. (You may have encountered this before… Excel can "pivot" long-form data into wide form using a "pivot table".)

Base R has the `reshape` function which can convert long to wide, or wide to long. However, the syntax of this function is rather tricky so a few packages have been built to make this process easier, including `reshape2` and `tidyr`. The `tidyr` package was introduced a couple of years ago in Hadley Wickam's "Tidy Data" article[1].

[1] Wickham, H., "Tidy Data", *Journal of Statistical Software*, V59N10 pp 1-23, 1994. DOI:10.18637/jss.v059.i10.

Let's put some data into a data frame to work with:

```
# long form data
DF4a <- read.table( text=
"X Y G
1  2 A
2  4 A
3  3 A
4  1 A
1  1 B
2  2 B
3  1 B
```

```
", header = TRUE, as.is = TRUE)
```

Figure~8 illustrates that these two curves have different numbers of points without requiring a blank or `NA` value in the data.

We can *widen* `DF4a` if we need to using the `tidyr spread` function:

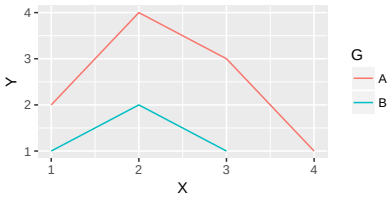

Figure 8: Long Data

```
library(tidyr)
DF4b <- (   DF4a
        %>% spread( G, Y )
        )
```

Table 2: Table 1. Wide version of `DF4a`

X	A	B
1	2	1
2	4	2
3	3	1
4	1	NA

Note that as many columns as needed will be created based on the contents of the `G` column.

If we need to go the other way, we can use the `gather` function, specifying the columns to collapse into `Y` as being "everything but column `X`":

```
DF4c <- (   DF4b
        %>% gather( G,  Y, -X )
        )
```

Table 3: Table 2. Long version of `DF4b`

X	G	Y
1	A	2
2	A	4
3	A	3
4	A	1
1	B	1
2	B	2
3	B	1
4	B	NA

or we can tell `gather` not to generate the NA row:

```
DF4d <- (   DF4b
        %>% gather( G, Y, -X, na.rm = TRUE)
        )
```

Table 4: Table 3. Long version of DF4b with no NA result

X	G	Y
1	A	2
2	A	4
3	A	3
4	A	1
1	B	1
2	B	2
3	B	1

Merging data frames

Often you will have a small data frame that summarizes some information, and you will want to look up certain rows/columns of data from that data frame and use it for calculations. (In Excel this is often accomplished with the VLOOKUP or HLOOKUP functions.) For example, you might have a short data frame:

```
DF5lookup <- read.table( text=
"WaterSeason Capacity Evap
Flood        550      8
Irrigation   250      26
", header = TRUE, as.is = TRUE )
```

And another data frame in which you want to do some calculations:

```
DF5calc <- read.table( text =
"Dt         Inflow  WaterSeason
1913-10-01   72.6  Flood
1914-04-01  233.6  Irrigation
1914-10-01  170.9  Flood
1915-04-01  591.4  Irrigation
1915-10-01  118.4  Flood
1916-04-01  406.1  Irrigation
", header = TRUE, as.is = TRUE )
DF5calc$Dt <- as.Date( DF5calc$Dt )
```

Suppose we want to compute *Capacity* − *Inflow*. We can use indexing to lookup the *Capacity* corresponding to each row of DF5calc:

```
DF5calc$Diff <- ( DF5lookup$Capacity[ match( DF5calc$WaterSeason
                                           , DF5lookup$WaterSeason ) ]
                - DF5calc$Inflow
                )
```

Table 5: Table 4. Calculation by indexing lookup

Dt	Inflow	WaterSeason	Diff
1913-10-01	72.6	Flood	477.4
1914-04-01	233.6	Irrigation	16.4
1914-10-01	170.9	Flood	379.1
1915-04-01	591.4	Irrigation	-341.4
1915-10-01	118.4	Flood	431.6
1916-04-01	406.1	Irrigation	-156.1

However, this is tedious and not particularly efficient. A more efficient approach is to *merge* the tables together using the `merge` function from base R:

```
DF5calc <- DF5calc[ -4 ] # drop the indexed calculation
DF5calc2 <- merge( DF5calc, DF5lookup, by = "WaterSeason" )
DF5calc2$Diff <- with( DF5calc2, Capacity - Inflow )
```

Table 6: Table 5. Calculation by `merge` from base R

WaterSeason	Dt	Inflow	Capacity	Evap	Diff
Flood	1913-10-01	72.6	550	8	477.4
Flood	1914-10-01	170.9	550	8	379.1
Flood	1915-10-01	118.4	550	8	431.6
Irrigation	1914-04-01	233.6	250	26	16.4
Irrigation	1915-04-01	591.4	250	26	-341.4
Irrigation	1916-04-01	406.1	250	26	-156.1

If you are using the `dplyr` package there is a more efficient alternative:

```
DF5calc3 <- (   DF5calc
            %>% inner_join( DF5lookup, by = "WaterSeason" )
            %>% mutate( Diff = Capacity - Inflow )
            )
```

Table 7: Table 6. Calculation by `inner_join` and `mutate`

Dt	Inflow	WaterSeason	Capacity	Evap	Diff
1913-10-01	72.6	Flood	550	8	477.4
1914-04-01	233.6	Irrigation	250	26	16.4
1914-10-01	170.9	Flood	550	8	379.1
1915-04-01	591.4	Irrigation	250	26	-341.4
1915-10-01	118.4	Flood	550	8	431.6
1916-04-01	406.1	Irrigation	250	26	-156.1