WEEK 7: MANIPULATING ENTIRE DATASETS

THOMAS ELLIOTT

While all of the following manipulations are possible in base R, the packages Dplyr and Tidyr make these manipulations so much easier and more predictable, so I'm going to focus on how to do them using these packages. Remember, to use functions in downloaded packages, you must first load the package with the `library()` function:

```
library(dplyr)
library(tidyr)
```

1. Append

In R it is possible to bind both rows and columns to an existing dataset. Dplyr supplies a `bind_rows()` function to add rows to a data frame and `bind_cols()` to add columns. `bind_cols()` is different than a merge, in that binding columns adds rows based on position, so the columns being added must have the same number of rows as the original data frame, and they are literally just pasted onto the end of the data frame. Merging, on the other hand, uses some identification variable to match rows in the original and binding data frame (more on merging in the next section).

`bind_rows()` attempts to match up column names. This means even if the columns are not in the same order in the two data frames, `bind_rows()` will re-order the columns to match.

```
> df1
    A  B
1   4 10
2   1 20
3   8 15
4  10  5
5   7  8
6   4  7
7   1 16
8   2 26
9   4 24
10  7 23
> df2
    B  A
1  20 54
2  25 59
3  29 58
```

Date: February 24, 2016.

```
4  21 51
5  30 55
6  23 60
7  30 51
8  26 53
9  27 50
10 20 55
> df.new<-bind_rows(df1,df2)
> print(df.new,n=Inf)
Source: local data frame [20 x 2]

       A     B
   (int) (int)
1      4    10
2      1    20
3      8    15
4     10     5
5      7     8
6      4     7
7      1    16
8      2    26
9      4    24
10     7    23
11    54    20
12    59    25
13    58    29
14    51    21
15    55    30
16    60    23
17    51    30
18    53    26
19    50    27
20    55    20
```

If `bind_rows()` finds a column in one data frame that is not in the other, it will append the new column, filling in missing values with `NA`

```
> df1
    A  B
1   4 10
2   1 20
3   8 15
4  10  5
5   7  8
6   4  7
7   1 16
8   2 26
```

```
9   4 24
10  7 23
> df3
     C   D
1  145 237
2  128 239
3  135 244
4  118 263
5  136 277
6  110 298
7  122 247
8  110 234
9  109 267
10 114 219
> df.new<-bind_rows(df1,df3)
> print(df.new,n=Inf)
Source: local data frame [20 x 4]

       A     B     C     D
   (int) (int) (int) (int)
1      4    10    NA    NA
2      1    20    NA    NA
3      8    15    NA    NA
4     10     5    NA    NA
5      7     8    NA    NA
6      4     7    NA    NA
7      1    16    NA    NA
8      2    26    NA    NA
9      4    24    NA    NA
10     7    23    NA    NA
11    NA    NA   145   237
12    NA    NA   128   239
13    NA    NA   135   244
14    NA    NA   118   263
15    NA    NA   136   277
16    NA    NA   110   298
17    NA    NA   122   247
18    NA    NA   110   234
19    NA    NA   109   267
20    NA    NA   114   219
```

`bind_cols()` makes not attempt to match rows (that's what merging is for) beyond making sure there are the same number of rows in the two data frames. If there are not the same number of rows, `bind_cols()` will throw an error.

```
> df.new<-bind_cols(df1,df3)
> print(df.new,n=Inf)
```

```
Source: local data frame [10 x 4]

       A     B     C     D
   (int) (int) (int) (int)
1      4    10   145   237
2      1    20   128   239
3      8    15   135   244
4     10     5   118   263
5      7     8   136   277
6      4     7   110   298
7      1    16   122   247
8      2    26   110   234
9      4    24   109   267
10     7    23   114   219
```

Finally, for both functions, you can specify more than two data frames at a time:

```
> df.new<-bind_rows(df1,df2,df3)
> print(df.new,n=Inf)
Source: local data frame [30 x 4]

       A     B     C     D
   (int) (int) (int) (int)
1      4    10    NA    NA
2      1    20    NA    NA
3      8    15    NA    NA
4     10     5    NA    NA
5      7     8    NA    NA
6      4     7    NA    NA
7      1    16    NA    NA
8      2    26    NA    NA
9      4    24    NA    NA
10     7    23    NA    NA
11    54    20    NA    NA
12    59    25    NA    NA
13    58    29    NA    NA
14    51    21    NA    NA
15    55    30    NA    NA
16    60    23    NA    NA
17    51    30    NA    NA
18    53    26    NA    NA
19    50    27    NA    NA
20    55    20    NA    NA
21    NA    NA   145   237
22    NA    NA   128   239
23    NA    NA   135   244
24    NA    NA   118   263
```

```
25    NA    NA   136   277
26    NA    NA   110   298
27    NA    NA   122   247
28    NA    NA   110   234
29    NA    NA   109   267
30    NA    NA   114   219
```

2. Merge

Dplyr uses SQL languages to name many of its functions, including its merge functions. As a result, Dplyr contains six merging functions it labels "joins."

inner_join: return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned. This would be similar to suppling the `keep(match)` option to Stata's merge function.

left_join: return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned. This is similar to suppling the `keep(master match)` option to Stata's merge function.

right_join: return all rows from y, and all columns from x and y. Rows in y with no match in x will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned. This is similar to supplying the `keep(using match)` option to Stata's merge function.

full_join: return all rows and all columns from both x and y. Where there are not matching values, returns NA for the one missing. This is the default behavior of Stata's merge function.

semi_join: return all rows from x where there are matching values in y, keeping just columns from x. A semi join differs from an inner join because an inner join will return one row of x for each matching row of y, where a semi join will never duplicate rows of x.

anti_join: return all rows from x where there are not matching values in y, keeping just columns from x.

So an inner join will only return rows that are matching in both data frames.

```
> df1
   id  A  B
1   1  1 12
2   2  5 25
3   3  9 24
4   4  2  4
5   5 10 16
6   6  4 28
7   7 10  4
```

```
8   8  7  9
9   9  7  2
10 10  1 16
> df2
   id  C  D
1   6 22 54
2   7 20 60
3   8 24 60
4   9 28 53
5  10 26 57
6  11 30 51
7  12 27 51
8  13 30 53
9  14 29 59
10 15 26 58
> df.new<-inner_join(df1,df2,by="id")
> df.new
  id  A  B  C  D
1  6  4 28 22 54
2  7 10  4 20 60
3  8  7  9 24 60
4  9  7  2 28 53
5 10  1 16 26 57
```

A left join will return all rows in the first data frame (the left-hand data frame), filling in missing values for rows that don't match.

```
> df.new<-left_join(df1,df2,by="id")
> df.new
   id  A  B  C  D
1   1  1 12 NA NA
2   2  5 25 NA NA
3   3  9 24 NA NA
4   4  2  4 NA NA
5   5 10 16 NA NA
6   6  4 28 22 54
7   7 10  4 20 60
8   8  7  9 24 60
9   9  7  2 28 53
10 10  1 16 26 57
```

Similarly, a right join will return all rows in the second data frame (the right-hand data frame), filling in missing values for rows that don't match.

```
> df.new<-right_join(df1,df2,by="id")
> df.new
   id  A  B  C  D
1   6  4 28 22 54
```

```
2   7 10  4 20 60
3   8  7  9 24 60
4   9  7  2 28 53
5  10  1 16 26 57
6  11 NA NA 30 51
7  12 NA NA 27 51
8  13 NA NA 30 53
9  14 NA NA 29 59
10 15 NA NA 26 58
```

A full join will return everything, regardless of whether there is a match.

```
> df.new<-full_join(df1,df2,by="id")
> df.new
   id  A  B  C  D
1   1  1 12 NA NA
2   2  5 25 NA NA
3   3  9 24 NA NA
4   4  2  4 NA NA
5   5 10 16 NA NA
6   6  4 28 22 54
7   7 10  4 20 60
8   8  7  9 24 60
9   9  7  2 28 53
10 10  1 16 26 57
11 11 NA NA 30 51
12 12 NA NA 27 51
13 13 NA NA 30 53
14 14 NA NA 29 59
15 15 NA NA 26 58
```

A semi join will return rows that match in both data frames, but on the columns from the first.

```
> df.new<-semi_join(df1,df2,by="id")
> df.new
  id  A  B
1  6  4 28
2  7 10  4
3  8  7  9
4  9  7  2
5 10  1 16
```

And an anti join will return rows in the first data frame that do not match in the second data frame.

```
> df.new<-anti_join(df1,df2,by="id")
> df.new
  id  A  B
```

```
1  5 10 16
2  4  2  4
3  3  9 24
4  2  5 25
5  1  1 12
```

This and the semi join can be useful for filtering data based on whether the observation also appears (or not) in a second data frame. For example, say you have a data frame of objects that should be coded, and a second data frame of objects that are already coded. An anti join of the second onto the first will produce a list of objects that haven't yet been coded.

3. Contract & Collapse

Dplyr has the `count()` function for duplicating the contract command in Stata. Count takes as arguments a data frame and one or more variable names and produces a data frame containing counts of the combinations of variables.

```
> data(mtcars)
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
> count(mtcars,cyl)
Source: local data frame [3 x 2]

    cyl     n
  (dbl) (int)
1     4    11
2     6     7
3     8    14
> count(mtcars,cyl,gear)
Source: local data frame [8 x 3]
Groups: cyl [?]

    cyl  gear     n
  (dbl) (dbl) (int)
1     4     3     1
2     4     4     8
3     4     5     2
4     6     3     2
5     6     4     4
6     6     5     1
```

```
7     8     3    12
8     8     5     2
```

To reproduce the functionality of collapse, dplyr has two functions: `group_by()` and `summarize()`. You first use group_by to define the groups over which you will generate summary statistics, and then you use summarize to define the statistics you want to generate. As an example, here is how you would generate mean MPG by cylinder:

```
> mtcars.cyl<-group_by(mtcars,cyl)
> mtcars.cyl<-summarize(mtcars.cyl,mpg=mean(mpg))
> mtcars.cyl
Source: local data frame [3 x 2]

    cyl      mpg
  (dbl)    (dbl)
1     4 26.66364
2     6 19.74286
3     8 15.10000
```

You can generate multiple summary statistics. To generate a count of observations in each group, use `n()`.

```
> mtcars.cyl<-group_by(mtcars,cyl)
> mtcars.cyl<-summarize(mtcars.cyl,mpg=mean(mpg),hp=mean(hp),count=n())
> mtcars.cyl
Source: local data frame [3 x 4]

    cyl      mpg        hp count
  (dbl)    (dbl)     (dbl) (int)
1     4 26.66364  82.63636    11
2     6 19.74286 122.28571     7
3     8 15.10000 209.21429    14
```

As with count, you can group by more than one variable so that groups are defined by the different combinations of values of the grouping variables. Within the summarize command, you can use any function that takes a vector and returns a singular value, including user defined functions. This makes summarize especially powerful.

If you want to perform the same operation on a series of values, say you want to calculate the mean value for all variables by cylinder, you can use `summarize_each()`. The syntax is a little more complicated. The first argument is a data frame, the second argument is a list of functions you want to run on each variable, wrapped in the `funs()` function, and then you supply the list of variables you want to summarize. As an example, to calculate the mean for all variables in the mtcars dataset:

```
> mtcars.cyl<-group_by(mtcars,cyl)
> mtcars.cyl<-summarize_each(mtcars.cyl,funs(mean),mpg,disp:carb)
> mtcars.cyl
Source: local data frame [3 x 11]
```

```
    cyl      mpg     disp        hp     drat       wt     qsec        vs        am     g
  (dbl)    (dbl)    (dbl)     (dbl)    (dbl)    (dbl)    (dbl)     (dbl)     (dbl)    (d
1     4 26.66364 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909 0.7272727 4.090
2     6 19.74286 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286 0.4285714 3.857
3     8 15.10000 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000 0.1428571 3.285
```

You can supply multiple functions in the `funs()` command, including user defined functions. You can also supply full expressions within the `funs()` command - see the help file for examples of this.

4. Reshape

To reshape we will use functions found in the package tidyr. To reshape long, we will use the function `gather()`, and to reshape wide we will use the function `spread()`. Spread takes as arguments first the data frame, second the key variable (in Stata this is j), and third the value variable (in Stata this is the variable listed after wide). The values of the key variable become the column names of the new variables. Unfortunately, `spread()` cannot currently spread multiple columns at the same time. If you need to do this, take a look at the functions contained in the `reshape2` package.

```
> mydata
   year    sex income
1  2000   male  40611
2  2001   male  38042
3  2002   male  46794
4  2003   male  31954
5  2004   male  32259
6  2005   male  23285
7  2006   male  39946
8  2007   male  20019
9  2008   male  33665
10 2009   male  28431
11 2010   male  48714
12 2000 female  32032
13 2001 female  28398
14 2002 female  24699
15 2003 female  31921
16 2004 female  32052
17 2005 female  24197
18 2006 female  47245
19 2007 female  24605
20 2008 female  29849
21 2009 female  39546
22 2010 female  34593
> mydata.wide<-spread(mydata,sex,income)
> mydata.wide
   year female  male
```

```
1  2000  32032 40611
2  2001  28398 38042
3  2002  24699 46794
4  2003  31921 31954
5  2004  32052 32259
6  2005  24197 23285
7  2006  47245 39946
8  2007  24605 20019
9  2008  29849 33665
10 2009  39546 28431
11 2010  34593 48714
```

To reshape long, we use `gather()`, as in gathering up a bunch of columns into one. Gather takes as arguments first the data frame, second the name of the key column which will be created, third the name of the value column that will be created, and fourth a list of columns to "gather."

```
> mydata.long<-gather(mydata.wide,sex,income,female,male)
> mydata.long
   year    sex income
1  2000 female  32032
2  2001 female  28398
3  2002 female  24699
4  2003 female  31921
5  2004 female  32052
6  2005 female  24197
7  2006 female  47245
8  2007 female  24605
9  2008 female  29849
10 2009 female  39546
11 2010 female  34593
12 2000   male  40611
13 2001   male  38042
14 2002   male  46794
15 2003   male  31954
16 2004   male  32259
17 2005   male  23285
18 2006   male  39946
19 2007   male  20019
20 2008   male  33665
21 2009   male  28431
22 2010   male  48714
```