Data Management in R

Political Methodology R Workshops (Workshop 2)

Eric Dunford
Department of Government and Politics
University of Maryland, College Park

Fall 2016

Contents

Quick Review: `R Basics` Workshop

Last week we covered:

- Objects and Classes
- Packages
- Importing Data
- Simple Descriptive Statistics and Graphics

For the complete slides, please go to the following link.

Wait, what are objects again?

R is an *object oriented* statistical programming language, which means we assign specific pieces of information (say a dataset) a specific textual representation (the name "data"). This allows us to recall and re-use information for future analysis.

We can *assign* values to an object with one of the three following commands:

- `<-`
- `=`
- `assign()`

For example,

```
x1 <- 3
x2 = 3
assign("x3",3)
c(x1, x2, x3)
```

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

Why are we always talking about class?

Objects have specific classes – that is, objects have different properties given what information is assigned to it. When we assign a dataset to an object `x`, the `x` will have the class `data.frame`. This tells us

a. how the data is organized,
b. how to access the information within the object (because we know something about its structure),
c. what functions the object will work with.

For example, say you have a function that only take `data.frame` objects as an argument. The function will return an error if you try to read in a list. So understanding the fundamentals of classes is important for problem-solving.

> ! For a complete list of all class and data types in R, go to the following link!

For an applied example consider the following. Here we are going to create two vectors, one that is composed of 4 string values, and one that is composed of 4 integers. We are then going to combine them into a data frame using the `data.frame()` function. We are then going to examine the objects `class`, `structure`, and then `print` it's output.

```r
# Remember: c() stands for concatenate, which means link together in a chain or
# a series

x <- c("a","b","c","d")
y <- 1:4

# Create a Data Frame from our two equal in length vect
my_data = data.frame(x,y)

class(my_data)
```

```
## [1] "data.frame"
```

```r
str(my_data)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ x: Factor w/ 4 levels "a","b","c","d": 1 2 3 4
##  $ y: int  1 2 3 4
```

```r
my_data
```

```
##   x y
## 1 a 1
## 2 b 2
## 3 c 3
## 4 d 4
```

Objects have Structure?

Different classes have different structures. What we mean by this is that the data is *housed* differently depending on the class of an object. What this means for us is that **how we *get at* specific pieces of data within an object differs given that objects structure.**

```r
# Consider a vector, data.frame, and list.
vector <- c(1,2,3,4,5)
data_frame <- data.frame(variable1 = 1:4,
                         variable2 = c(5,6,.3,99))
List <- list(vector,data_frame)

# we can access bits and pieces of a data structure by using "brackets" -- Think
# of it like this, we are pointing to specific locations within the data and
# saying give me just that piece. To point in the right place, we need to know
# something about where in the house the data is.

# *** Accessing data in a vector ***

        # Only one "dimension"
        length(vector) # We can access one of the five values
```

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

```r
        str(vector)
```

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

```r
        vector[1]
```

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

```r
# *** Accessing data in a data.frame ***

        # More complex, two dimensions
        dim(data_frame) # 4 rows, 2 columns
```

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

```r
        str(data_frame)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ variable1: int  1 2 3 4
##  $ variable2: num  5 6 0.3 99
```

```r
        data_frame[1,2] # row 1, column 2
```

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

```r
        data_frame[4,] # All of row 4
```

```
##   variable1 variable2
## 4         4        99
```

```r
        data_frame[,2] # all of column 2
```

```
## [1]  5.0  6.0  0.3 99.0
```

```r
        # We can also use the call sign $ to access specific variables in a
        # data. frame
        data_frame$variable1
```

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

```r
# *** Accessing data in a list ***

        # Lists are "non-relational" meaning you can store a lot of differnt
        # kinds of data that are of different lengths and composition. Just look
        # at our list. We loaded both a vector and a data.frame into the SAME
        # OBJECT. Wow! Lists are what most package output will come as.

        length(List) # two "sets" of things in this list
```

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

```r
str(List) # we can see that we have a vector and a data frame
```

```
## List of 2
##  $ : num [1:5] 1 2 3 4 5
##  $ :'data.frame':    4 obs. of  2 variables:
##   ..$ variable1: int [1:4] 1 2 3 4
##   ..$ variable2: num [1:4] 5 6 0.3 99
```

```r
List[2] # Let's access the data.frame in the list
```

```
## [[1]]
##   variable1 variable2
## 1         1       5.0
## 2         2       6.0
## 3         3       0.3
## 4         4      99.0
```

```r
class(List[2])
```

```
## [1] "list"
```

```r
# Say we just wanted to retrieve the data.frame. Well this is where the
# bracket logic can get a little confusing because we can "go deeper"
# into the objects structure.
List[[2]]
```

```
##   variable1 variable2
## 1         1       5.0
## 2         2       6.0
## 3         3       0.3
## 4         4      99.0
```

```r
class(List[[2]]) # See we are drawing out the data frame that is inside the list
```

```
## [1] "data.frame"
```

```r
# Now let's grab the fourth row of the data.frame that's in the list
List[[2]][4,]
```

```
##   variable1 variable2
## 4         4        99
```

```r
List[[2]][,1] # or the first column
```

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

Importing Data?

Importing data is straightforward, but not all the **base packages** in R can handle every data source. To load in different types of data, you'll need to utilize different packages. Last week we reviewed the following packages:

- `foreign` – import sas, spss, stata (version 12 <=)
- `readstata13` – importing stata (version 13 >=)
- `XLconnect` – importing excel worksheets.

Reading in `Rdata` or `.csv` is built into R's base functionality, which makes it the "easiest" data formats to use.

Remember: to access a data frame, you first have to tell `R` where that data is located. We can do this one of two ways:

1. set a working directory with 'setwd("/Users/Your_computer/Desktop")
2. specify the path when loading the data, e.g., `"/Users/Your_computer/Desktop/data.csv"`

For example, let's write data to our desktop and then read it back in.

```
path <- "/Users/edunford/Desktop/data.csv"
write.csv(data_frame,file=path,row.names = F)
new_data <- read.csv(file=path)
new_data
```

```
##   variable1 variable2
## 1         1       5.0
## 2         2       6.0
## 3         3       0.3
## 4         4      99.0
```

Dropping Objects

What is some thing that we should have covered last week but didn't? Oh **dropping objects**, of course! We reviewed how to *create* an object, but what if I want to get rid of it? That's easy: use the command `rm()` to remove all objects that you don't want.

```
ls() # What are all the objects that we've created?
```

```
##  [1] "data_frame" "List"       "my_data"    "new_data"   "path"
##  [6] "vector"     "x"          "x1"         "x2"         "x3"
## [11] "y"
```

```
x # remember our vector
```

```
## [1] "a" "b" "c" "d"
```

```
# Let's drop it!
rm(x)
ls() # Gone!
```

```
##  [1] "data_frame" "List"       "my_data"    "new_data"   "path"
##  [6] "vector"     "x1"         "x2"         "x3"         "y"
```

```
# What if I want to completely clear the workspace? Do the following:
rm(list=ls(all=T))
ls() # if only starting from scratch was this easy in real life!
```

```
## character(0)
```

Basic Data Manipulations

Today we are going to cover all aspects of data manipulation and management. Now that we understand what objects *are* and how to access the information *within* them, we are on good ground to understand how to craft an object into something that is analytically useful. We'll focus primarily on `data.frames` as this is the dominant data structure used in political science research.

Here we are going to use a dataset that is inherent in R, called `sleep`. These data show the effect of two soporific drugs (increase in hours of sleep compared to control) on 10 patients.

```
data <- sleep
str(data)
```

```
## 'data.frame':    20 obs. of  3 variables:
##  $ extra: num  0.7 -1.6 -0.2 -1.2 -0.1 3.4 3.7 0.8 0 2 ...
##  $ group: Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 1 1 ...
##  $ ID   : Factor w/ 10 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
```

Variables

Creating Variables

Recall that we can access a variables contents using the call sign `$`. We can also use this same call logic to create a new variable.

```
data$extra # varible regarding extra-sleep
```

```
##  [1]  0.7 -1.6 -0.2 -1.2 -0.1  3.4  3.7  0.8  0.0  2.0  1.9  0.8  1.1  0.1
## [15] -0.1  4.4  5.5  1.6  4.6  3.4
```

```
data$Add_2 <- data$extra + 2
head(data)
```

```
##   extra group ID Add_2
## 1   0.7     1  1   2.7
## 2  -1.6     1  2   0.4
## 3  -0.2     1  3   1.8
## 4  -1.2     1  4   0.8
## 5  -0.1     1  5   1.9
## 6   3.4     1  6   5.4
```

We can also use other aspects of a data frame's structure to the same end.

```
data[,5] <- 1 # As Column 4, load the value 1 for all obs.
head(data) # Assign arbitrary name
```

```
##   extra group ID Add_2 V5
## 1   0.7     1  1   2.7  1
## 2  -1.6     1  2   0.4  1
## 3  -0.2     1  3   1.8  1
## 4  -1.2     1  4   0.8  1
## 5  -0.1     1  5   1.9  1
## 6   3.4     1  6   5.4  1
```

Or we can call the columns name.

```
data[,"V4"] <- 41:60
head(data)
```

```
##   extra group ID Add_2 V5 V4
## 1   0.7     1  1   2.7  1 41
## 2  -1.6     1  2   0.4  1 42
## 3  -0.2     1  3   1.8  1 43
## 4  -1.2     1  4   0.8  1 44
## 5  -0.1     1  5   1.9  1 45
## 6   3.4     1  6   5.4  1 46
```

The creation of any variable follows this same logic *as long as the vector being inserted is of the* ***correct length***.

```
nrow(data)
```

```
## [1] 20
```

```
data[,"New_Variable"] <- rnorm(n=20,mean = 40,sd=5)
head(data) # Works!
```

```
##   extra group ID Add_2 V5 V4 New_Variable
## 1   0.7     1  1   2.7  1 41     46.84230
## 2  -1.6     1  2   0.4  1 42     39.69103
## 3  -0.2     1  3   1.8  1 43     37.39332
## 4  -1.2     1  4   0.8  1 44     33.93577
## 5  -0.1     1  5   1.9  1 45     35.17994
## 6   3.4     1  6   5.4  1 46     41.46829
```

```
data[,"New_Variable"] <- rnorm(n=21,mean = 40,sd=5) # Breaks
data[,"New_Variable"] <- rnorm(n=19,mean = 40,sd=5) # Breaks
```

Note that all transformation of any variable follow the same logic.

```
data$NV_ln <- log(data$New_Variable) # Natural Log
data$NV_ln10 <- log10(data$New_Variable) # Log Base 10
data$NV_e <- exp(data$New_Variable) # Exponentiate
data$NV_sqrt <- sqrt(data$New_Variable) # Square Root
data$NV_abs <- abs(data$New_Variable) # Absolute Value
head(data)
```

```
##   extra group ID Add_2 V5 V4 New_Variable    NV_ln  NV_ln10         NV_e
## 1   0.7     1  1   2.7  1 41     46.84230 3.846787 1.670638 2.204715e+20
## 2  -1.6     1  2   0.4  1 42     39.69103 3.681125 1.598692 1.728209e+17
## 3  -0.2     1  3   1.8  1 43     37.39332 3.621492 1.572794 1.736656e+16
## 4  -1.2     1  4   0.8  1 44     33.93577 3.524470 1.530658 5.471649e+14
## 5  -0.1     1  5   1.9  1 45     35.17994 3.560476 1.546295 1.898680e+15
## 6   3.4     1  6   5.4  1 46     41.46829 3.724929 1.617716 1.022001e+18
##    NV_sqrt   NV_abs
## 1 6.844143 46.84230
## 2 6.300082 39.69103
## 3 6.115008 37.39332
## 4 5.825442 33.93577
## 5 5.931268 35.17994
## 6 6.439588 41.46829
```

Categorical Variables

As we discussed last week, there are two class types for strings in R (again, a "string" is anything contained within quotes): `factors` and `characters`. Factors are useful when creating categorical variables as they retain `levels`, which are numeric place holders. As long as a character variable is a factor, it can be used as a categorical variable

In the sleep data, the `groups` variable is already a factor.

```
data$group
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
## Levels: 1 2
```

To create a categorical, variable, we just need to create our own factor.

```
vec <- rep(c("a","b","c","d","e"),4)  # repeating the vector 4 times
vec <- factor(vec)
vec
```

```
##  [1] a b c d e a b c d e a b c d e a b c d e
## Levels: a b c d e
```

```
data$my_cat <- vec
# by coercing the value into a numeric, we can retrieve the underlying levels.
data$my_cat
```

```
##  [1] a b c d e a b c d e a b c d e a b c d e
## Levels: a b c d e
```

```r
as.numeric(data$my_cat)
```

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

Ordinal Variables (`ifelse()` conditionals)

Often we need to chop up a distribution into an ordered variable. This is straightforward when using the `ifelse()` conditional statement. Essentially, we are saying: if the variable meets this criteria, code it as this; else do this.

For an example, let's break the `extra` variable up into a dichotomous indicator.

```r
mean(data$extra)
```

```
## [1] 1.54
```

```r
data$extra_dich <- ifelse(data$extra>=mean(data$extra),1,0)
data[,c("extra","extra_dich")]
```

```
##    extra extra_dich
## 1    0.7          0
## 2   -1.6          0
## 3   -0.2          0
## 4   -1.2          0
## 5   -0.1          0
## 6    3.4          1
## 7    3.7          1
## 8    0.8          0
## 9    0.0          0
## 10   2.0          1
## 11   1.9          1
## 12   0.8          0
## 13   1.1          0
## 14   0.1          0
## 15  -0.1          0
## 16   4.4          1
## 17   5.5          1
## 18   1.6          1
## 19   4.6          1
## 20   3.4          1
```

To build more complex ordinal values, we can expand this process by linking a bunch of `ifelse()` statements.

```r
sum <- summary(data$extra)
cat(sum[2],sum[3],sum[5]) # 1st Q, median, 3rd Q
```

```
## -0.025 0.95 3.4
```

```r
data$extra_ord <- ifelse(data$extra>=sum[2] & data$extra<sum[3],1,
                      ifelse(data$extra>=sum[3] & data$extra<sum[4],2,
                             ifelse(data$extra>=sum[4],3,0)))
data[,c("extra","extra_dich","extra_ord")]
```

```
##    extra extra_dich extra_ord
## 1    0.7          0         1
## 2   -1.6          0         0
## 3   -0.2          0         0
## 4   -1.2          0         0
## 5   -0.1          0         0
## 6    3.4          1         3
## 7    3.7          1         3
## 8    0.8          0         1
## 9    0.0          0         1
## 10   2.0          1         3
## 11   1.9          1         3
## 12   0.8          0         1
## 13   1.1          0         2
## 14   0.1          0         1
## 15  -0.1          0         0
## 16   4.4          1         3
## 17   5.5          1         3
## 18   1.6          1         3
## 19   4.6          1         3
## 20   3.4          1         3
```

Dropping Variables

We can also reverse this process by loading `NULL` to any variable. This in effect "drops"" the variable.

```
data$Add_2 <- NULL
head(data) # Gone!
```

```
##   extra group ID V5 V4 New_Variable    NV_ln  NV_ln10         NV_e
## 1   0.7     1  1  1 41     46.84230 3.846787 1.670638 2.204715e+20
## 2  -1.6     1  2  1 42     39.69103 3.681125 1.598692 1.728209e+17
## 3  -0.2     1  3  1 43     37.39332 3.621492 1.572794 1.736656e+16
## 4  -1.2     1  4  1 44     33.93577 3.524470 1.530658 5.471649e+14
## 5  -0.1     1  5  1 45     35.17994 3.560476 1.546295 1.898680e+15
## 6   3.4     1  6  1 46     41.46829 3.724929 1.617716 1.022001e+18
##    NV_sqrt   NV_abs my_cat extra_dich extra_ord
## 1 6.844143 46.84230      a          0         1
## 2 6.300082 39.69103      b          0         0
## 3 6.115008 37.39332      c          0         0
## 4 5.825442 33.93577      d          0         0
## 5 5.931268 35.17994      e          0         0
## 6 6.439588 41.46829      a          1         3
```

Or use **negative values in the brackets** to specify variables you'd like to drop.

```
head(data[,c(-3,-4,-5,-6,-7)])
```

```
##   extra group  NV_ln10         NV_e  NV_sqrt   NV_abs my_cat extra_dich
## 1   0.7     1 1.670638 2.204715e+20 6.844143 46.84230      a          0
## 2  -1.6     1 1.598692 1.728209e+17 6.300082 39.69103      b          0
## 3  -0.2     1 1.572794 1.736656e+16 6.115008 37.39332      c          0
```

```
## 4  -1.2     1 1.530658 5.471649e+14 5.825442 33.93577      d           0
## 5  -0.1     1 1.546295 1.898680e+15 5.931268 35.17994      e           0
## 6   3.4     1 1.617716 1.022001e+18 6.439588 41.46829      a           1
##   extra_ord
## 1         1
## 2         0
## 3         0
## 4         0
## 5         0
## 6         3
```

```
# Or a quicker way would be to do the following...
head(data[,c(3:7)*-1])
```

```
##   extra group  NV_ln10         NV_e  NV_sqrt   NV_abs my_cat extra_dich
## 1   0.7     1 1.670638 2.204715e+20 6.844143 46.84230      a           0
## 2  -1.6     1 1.598692 1.728209e+17 6.300082 39.69103      b           0
## 3  -0.2     1 1.572794 1.736656e+16 6.115008 37.39332      c           0
## 4  -1.2     1 1.530658 5.471649e+14 5.825442 33.93577      d           0
## 5  -0.1     1 1.546295 1.898680e+15 5.931268 35.17994      e           0
## 6   3.4     1 1.617716 1.022001e+18 6.439588 41.46829      a           1
##   extra_ord
## 1         1
## 2         0
## 3         0
## 4         0
## 5         0
## 6         3
```

```
# as we know that
c(3:7)*-1
```

```
## [1] -3 -4 -5 -6 -7
```

We can also **subset out** a variable.

```
new_data <- data[,c(1,2)]
head(new_data) # only selected two variables and made a new object.
```

```
##   extra group
## 1   0.7     1
## 2  -1.6     1
## 3  -0.2     1
## 4  -1.2     1
## 5  -0.1     1
## 6   3.4     1
```

Renaming Variables

Inevitably, you we'll need to rename variables. Doing so is straightforward with the `colnames()` function.

```r
colnames(data)
```

```
##  [1] "extra"        "group"        "ID"           "V5"
##  [5] "V4"           "New_Variable" "NV_ln"        "NV_ln10"
##  [9] "NV_e"         "NV_sqrt"      "NV_abs"       "my_cat"
## [13] "extra_dich"   "extra_ord"
```

```r
# colnames behaves like any vector, and as such, we can access the information
# as we would any vector
colnames(data)[4]
```

```
## [1] "V5"
```

```r
colnames(data)[4:5]
```

```
## [1] "V5" "V4"
```

```r
# Renaming a variable is as easy as inserting a new value in the data structure.
colnames(data)[4] <- "constant"
colnames(data)
```

```
##  [1] "extra"        "group"        "ID"           "constant"
##  [5] "V4"           "New_Variable" "NV_ln"        "NV_ln10"
##  [9] "NV_e"         "NV_sqrt"      "NV_abs"       "my_cat"
## [13] "extra_dich"   "extra_ord"
```

```r
colnames(data)[1:5] <- c("var1","var2","var3","var4","var5")
colnames(data)
```

```
##  [1] "var1"         "var2"         "var3"         "var4"
##  [5] "var5"         "New_Variable" "NV_ln"        "NV_ln10"
##  [9] "NV_e"         "NV_sqrt"      "NV_abs"       "my_cat"
## [13] "extra_dich"   "extra_ord"
```

```r
head(data)
```

```
##   var1 var2 var3 var4 var5 New_Variable    NV_ln  NV_ln10         NV_e
## 1  0.7    1    1    1   41     46.84230 3.846787 1.670638 2.204715e+20
## 2 -1.6    1    2    1   42     39.69103 3.681125 1.598692 1.728209e+17
## 3 -0.2    1    3    1   43     37.39332 3.621492 1.572794 1.736656e+16
## 4 -1.2    1    4    1   44     33.93577 3.524470 1.530658 5.471649e+14
## 5 -0.1    1    5    1   45     35.17994 3.560476 1.546295 1.898680e+15
## 6  3.4    1    6    1   46     41.46829 3.724929 1.617716 1.022001e+18
##    NV_sqrt   NV_abs my_cat extra_dich extra_ord
## 1 6.844143 46.84230      a          0         1
## 2 6.300082 39.69103      b          0         0
## 3 6.115008 37.39332      c          0         0
## 4 5.825442 33.93577      d          0         0
## 5 5.931268 35.17994      e          0         0
## 6 6.439588 41.46829      a          1         3
```

Subsetting Data

As noted above, it's straightforward to subset data given what we know about an object's structure. But there are also a few functions that make our life easier.

```
# The many ways to subset
data <- sleep

# Let's subset the data so that we only have group 2. There are many ways to do
# this, let's explore a few.

      # (1) Use the what we know about boolean operators from last week.
      data[data$group==2,]
```

```
##    extra group ID
## 11   1.9     2  1
## 12   0.8     2  2
## 13   1.1     2  3
## 14   0.1     2  4
## 15  -0.1     2  5
## 16   4.4     2  6
## 17   5.5     2  7
## 18   1.6     2  8
## 19   4.6     2  9
## 20   3.4     2 10
```

```
            # More complex?
            data[data$group==2 & data$extra>=4,]
```

```
##    extra group ID
## 16   4.4     2  6
## 17   5.5     2  7
## 19   4.6     2  9
```

```
            # Subset and only give me the first column
            data[data$group==2 & data$extra>=4,1]
```

```
## [1] 4.4 5.5 4.6
```

```
      # (2) Use the subset function which is a base function in R
      subset(data, subset = group==2)
```

```
##    extra group ID
## 11   1.9     2  1
## 12   0.8     2  2
## 13   1.1     2  3
## 14   0.1     2  4
## 15  -0.1     2  5
## 16   4.4     2  6
## 17   5.5     2  7
## 18   1.6     2  8
## 19   4.6     2  9
## 20   3.4     2 10
```

```r
# More complex?
subset(data, group==2 & extra>=4)
```

```
##    extra group ID
## 16   4.4     2  6
## 17   5.5     2  7
## 19   4.6     2  9
```

```r
# Subset and only give me the first column
subset(data, group==2 & extra>=4,select = extra)
```

```
##    extra
## 16   4.4
## 17   5.5
## 19   4.6
```

```r
# Or more
subset(data, group==2 & extra>=4,select = c(extra,group))
```

```
##    extra group
## 16   4.4     2
## 17   5.5     2
## 19   4.6     2
```

Merging Data

Merging data is a *must* in quantitative political analysis by bringing various datasets together we can enrich our analysis. But this isn't always straightforward. Sometimes observations can be dropped if one is not vigilant of the dimensions of each data frame being input.

The Basics

```r
# Let's create two example data frames. Note that rep() is a function to repeat
# a sequence a specific number of times.

countries <- rep(c("China","Russia","US","Benin"),2)
years <- c(rep(1999,4),rep(2000,4))

data1 <- data.frame(country=countries,
                    year=years,
                    repress = c(1,2,4,3,2,3,4,1),stringsAsFactors = F)

data2 <- data.frame(country=countries,
                    year=years,
                    GDPpc= round(runif(8,2e3,20e3),3),stringsAsFactors = F)

head(data1);head(data2)
```

```
##   country year repress
## 1   China 1999       1
## 2  Russia 1999       2
```

```
## 3      US 1999       4
## 4   Benin 1999       3
## 5   China 2000       2
## 6  Russia 2000       3
```

```
##   country year     GDPpc
## 1   China 1999 12080.659
## 2  Russia 1999 13189.585
## 3      US 1999  7149.912
## 4   Benin 1999  2757.655
## 5   China 2000 15129.235
## 6  Russia 2000 11230.965
```

```
# Merging the datasets: here we'll merge the data utilizing a unqiue identifier
# that is common across the two datasets

        merge(data1,data2,by="country") # Just countries
```

```
##    country year.x repress year.y     GDPpc
## 1    Benin   1999       3   1999  2757.655
## 2    Benin   1999       3   2000 11266.643
## 3    Benin   2000       1   1999  2757.655
## 4    Benin   2000       1   2000 11266.643
## 5    China   1999       1   1999 12080.659
## 6    China   1999       1   2000 15129.235
## 7    China   2000       2   1999 12080.659
## 8    China   2000       2   2000 15129.235
## 9   Russia   1999       2   1999 13189.585
## 10  Russia   1999       2   2000 11230.965
## 11  Russia   2000       3   1999 13189.585
## 12  Russia   2000       3   2000 11230.965
## 13      US   1999       4   1999  7149.912
## 14      US   1999       4   2000 16604.270
## 15      US   2000       4   1999  7149.912
## 16      US   2000       4   2000 16604.270
```

```
        merge(data1,data2,by=c("country","year")) # country-years
```

```
##   country year repress     GDPpc
## 1   Benin 1999       3  2757.655
## 2   Benin 2000       1 11266.643
## 3   China 1999       1 12080.659
## 4   China 2000       2 15129.235
## 5  Russia 1999       2 13189.585
## 6  Russia 2000       3 11230.965
## 7      US 1999       4  7149.912
## 8      US 2000       4 16604.270
```

Inevitable Issues

```r
# Assume that we are merging two data frames that do not contain the exact same
# units.

dataA = data1[data1$year==1999,] # subset the working data
dataB = data2[data2$year==1999,] # subset the working data
dataA[1,"country"] <- "Belize"
dataB[2,"country"] <- "Turkey"

dataA;dataB # Here we have slighly different countries in each DF
```

```
##   country year repress
## 1  Belize 1999       1
## 2  Russia 1999       2
## 3      US 1999       4
## 4   Benin 1999       3

##   country year     GDPpc
## 1   China 1999 12080.659
## 2  Turkey 1999 13189.585
## 3      US 1999  7149.912
## 4   Benin 1999  2757.655
```

```r
merge(dataA,dataB,by=c("country","year")) # Ah! Only a few merged?
```

```
##   country year repress    GDPpc
## 1   Benin 1999       3 2757.655
## 2      US 1999       4 7149.912
```

```r
# We need to specify to the merge function that we want all observations back
merge(dataA,dataB,by=c("country","year"),all=T)
```

```
##   country year repress     GDPpc
## 1  Belize 1999       1        NA
## 2   Benin 1999       3  2757.655
## 3   China 1999      NA 12080.659
## 4  Russia 1999       2        NA
## 5  Turkey 1999      NA 13189.585
## 6      US 1999       4  7149.912
```

```r
# We can preference a particular data set when doing this
merge(dataA,dataB,by=c("country","year"),all.x=T)
```

```
##   country year repress    GDPpc
## 1  Belize 1999       1       NA
## 2   Benin 1999       3 2757.655
## 3  Russia 1999       2       NA
## 4      US 1999       4 7149.912
```

```r
merge(dataA,dataB,by=c("country","year"),all.y=T)
```

```
##   country year repress     GDPpc
## 1   Benin 1999       3  2757.655
## 2   China 1999      NA 12080.659
## 3  Turkey 1999      NA 13189.585
## 4      US 1999       4  7149.912
```

When you have a variable with the same name

Sometimes we have a variable that is named the same in both datasets. R has a built in convention for dealing with these issues. It will automatically assign a temporary naming convention to deal with the duplicates.

```
dataA$GDPpc <- round(runif(4,2e3,20e3),3) # Create a similar var
merge(dataA,dataB,by=c("country","year"),all=T)
```

```
##   country year repress   GDPpc.x   GDPpc.y
## 1  Belize 1999       1  5679.474        NA
## 2   Benin 1999       3  6775.899  2757.655
## 3   China 1999      NA        NA 12080.659
## 4  Russia 1999       2  4869.432        NA
## 5  Turkey 1999      NA        NA 13189.585
## 6      US 1999       4 15307.703  7149.912
```

Loops

As one quickly notes, doing any task in R can become redundant. Loops and functions can dramatically increase our workflow when a task is *systematic and repeatable*.

As a motivating example, say we wanted to calculate the **group mean** of the `extra` variable for each group in our `sleep` data, and then save the output in a vector

```
data <- sleep
# Here we are going to paste the word "Group" for each group to create group
# names
data$group <- paste0("Group",data$group)
data$group
```

```
##  [1] "Group1" "Group1" "Group1" "Group1" "Group1" "Group1" "Group1"
##  [8] "Group1" "Group1" "Group1" "Group2" "Group2" "Group2" "Group2"
## [15] "Group2" "Group2" "Group2" "Group2" "Group2" "Group2"
```

```
# To do this, we'd need to subset by each group and then calculate the mean.
sub <- data[data$group=="Group1",]
mu1 <- mean(sub$extra)

sub <- data[data$group=="Group2",]
mu2 <- mean(sub$extra)

group_means <- c(mu1,mu2) # combine
group_means
```

```
## [1] 0.75 2.33
```

This works when there are only a few groups, but it would become quite the undertaking as the number of groups increased. Here is where **loops** can make one's life easier! By "**looping through**" all the respective groups, we can automate this process so that it goes a lot quicker.

A loop essentially works like this:

1. Specify a length of some thing you want to loop through. In our case, it's the number of groups.
2. Set the code up so that every iteration only performs a manipulation on a single subset at a time.
3. Save the contents of each iteration in a new object that won't be overwritten. Here we want to think in terms of "stacking" results or concatenating them.

In practice...

```r
# (1) Specify the length
no.of.groups = unique(data$group) # only unique entries
no.of.groups
```

```
## [1] "Group1" "Group2"
```

```r
1:length(no.of.groups)
```

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

```r
# (2) Make the code iterable
sub = data[data$group==no.of.groups[1],]
# Here, just by changing where we are in the vector "no.of.groups", we can draw
# out a unique subset

# (3) Save the contents
container = c() # create an empty container that we can append to.
# here we concatenate the mean with the empty object "container", which will
# save each additional value.

        # For example
            container = c()
            container <- c(container,1.2)
            container <- c(container,6.7)
            container <- c(container,9.8)
            container
```

```
## [1] 1.2 6.7 9.8
```

Now combine all these elements if a special base function called `for(){}` – note that all the code goes in-between the brackets. Here we need to establish an arbitrary iterator, which I'll call `i` in the example below. `i` will take the value of each entry in the vector `1:length(no.of.groups)`, e.g. `i=1` then `i=2`, and so on given how many groups we have.

```r
container = c() # Empty Container
for ( i in 1:length(no.of.groups) ){
  sub = data[data$group==no.of.groups[i],]
  mu <- mean(sub$extra)
  container <- c(container,mu)
}
container
```

```
## [1] 0.75 2.33
```

To really illustrate this point, let's make the data big! Note that `letters` is a vector of all the letters in the Latin alphabet. This vector is built into R. Also, check out just how arbitrary the iterator is!

```r
N = 1000 # Number of Observations
big_data = data.frame(group = rep(letters[1:10],N/10), # ten groups total
                      id = round(runif(N,1,100)),
                      value = rnorm(N,6,4),stringsAsFactors = F)
head(big_data)
```

```
##   group id    value
## 1     a 47 2.936952
## 2     b  5 5.900401
## 3     c 55 4.488903
## 4     d 86 4.995793
## 5     e 21 9.228599
## 6     f 70 4.284615
```

```r
# Now, let's construct the loop.
no.of.groups = unique(big_data$group)
container1 = c() # Empty Container
container2 = c() # Empty Container
for ( super_arbitrary_iterator in 1:length(no.of.groups) ){
  sub = big_data[big_data$group==no.of.groups[super_arbitrary_iterator],]
  mu <- mean(sub$value)

  # Let's have two containers. One that appends in a list, the other that stacks
  # values in a matrix using "rbind()", which stands for row bind.
  container1 <- c(container1,mu)
  container2 <- rbind(container2,mu)
}
container1
```

```
##  [1] 5.892415 6.569435 5.952398 6.252992 5.977208 5.461174 6.030014
##  [8] 5.997285 5.694578 5.836705
```

```r
container2
```

```
##        [,1]
## mu 5.892415
## mu 6.569435
## mu 5.952398
```

```
## mu 6.252992
## mu 5.977208
## mu 5.461174
## mu 6.030014
## mu 5.997285
## mu 5.694578
## mu 5.836705
```

Oh the possibilities!

Functions

If you'll recall, a `package` is really a bag of functions that someone threw together to perform specific tasks. More often than not, we have specific tasks that we have to implement *all the time*. Building a function for these tasks can really make life easier, and often it makes one's work more reproducible and transparent. **The secret to a good function is that the task can be generalized.** That is, across many different data contexts, it can be applied.

For example, consider calculating the mean for any variable.

```
var <- c(4,7,90,.3)
(var[1] + var[2] + var[3] + var[4])/length(var) # Mean by hand!
```

```
## [1] 25.325
```

```
# Just imagine if we had to spell this out every time we wanted know the mean!
# Luckily, there is a nice built-in function that does this for us.
mean(var)
```

```
## [1] 25.325
```

Let's go through the process of **building our own functions in `R`**. In basic terms, a function is a specific set of arguments that perform a specific task.

Let's build a simple function that **adds two values**. Here the function will have two arguments, or put differently, two *values* that need to be entered for the function to perform. As you'll note, this looks a lot like the set up for a loop!

```
add_me <- function( argument1, argument2 ){
  value <- argument1 + argument2
  return(value) # "return" means "send this back once the function is done"
}

add_me(2,3)
```

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

```r
add_me(100,123)
```

```
## [1] 223
```

```r
add_me(60,3^4)
```

```
## [1] 141
```

```r
# We can set "default" values for an argument, so if there is no inputs, the
# function will still run.
add_me <- function( argument1=1, argument2=2 ){
  value <- argument1 + argument2
  return(value)
}
add_me()
```

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

```r
add_me(4,5)
```

```
## [1] 9
```

Now, let's build a function for our **group mean loop** that we constructed in the last section. The arguments we would need are straight forward. We need the `data`, the name of the `group` column, and the name of the `value` column.

```r
group_mean <- function ( data, group.var, value.var ){

  no.of.groups = unique(data[,group.var])
  # Does anyone know why I am accessing the data this way?

  container = c() # Empty Container

  for ( super_arbitrary_iterator in 1:length(no.of.groups) ){
    sub = data[data[,group.var]==no.of.groups[super_arbitrary_iterator],]
    mu <- mean(sub[,value.var])
    container <- rbind(container,mu) # return as matrix
  }

  # Lastly, let's also enter in the group names as row names
  rownames(container) <- no.of.groups

  return(container)
}

group_mean(big_data,group.var = "group",value.var = "value")
```

```
##       [,1]
## a 5.892415
## b 6.569435
```

```
## c 5.952398
## d 6.252992
## e 5.977208
## f 5.461174
## g 6.030014
## h 5.997285
## i 5.694578
## j 5.836705
```

```
# Does it travel across different data contexts?

# Recall the fake country data?
head(data1)
```

```
##   country year repress
## 1   China 1999       1
## 2  Russia 1999       2
## 3      US 1999       4
## 4   Benin 1999       3
## 5   China 2000       2
## 6  Russia 2000       3
```

```
group_mean(data1,group.var = "country",value.var = "repress") # beautiful!
```

```
##        [,1]
## China   1.5
## Russia  2.5
## US      4.0
## Benin   2.0
```

Clearly, when done well functions and loops can make one's life easier. But they are also more than that. They make `R` unique, customizable, transparent, and flexible. Instead of saying, "I can't wait until someone builds software that does [insert specific thing here]", we can just build it ourselves! This is amazingly liberating.

Vectorization

Vectorization is a faster way to manipulate data sources in `R`. Technically, we do it all the time without realizing.

For example, here we are multiplying the value `3` across all values of `x` simultaneously.

```
x <- 1:10
x*3
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

When we talk about vectorization, we are talking about the **`apply` family** of functions. Think of `apply` as a loop. There is some value that the function can iterate across and it does so across all values simultaneously. This often (not always) results in faster code.

What is nice about `apply` functions is that you can write a quick function, and then apply it to all parts of your data without having to construct an iterative loop.

The "family" includes:

- `apply` – basic, great for matrix and data frame manipulations. Systematically moves across rows or columns and "applies" some function.
- `lapply` – apply a function across a list, does something to each aspect of the list, then returns a list.
- `sapply` – simpler than `lapply`. Reads in a list and returns a vector.
- `tapply` – apply a function to data in "cells".
- `by` – a more complex `tapply` that allows one to move across a data.frame "by" subgroups.
- and more! `mapply`, `eapply`, `vapply`

Using the `iris` data from last week, let's review a few of these functions.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

`apply`

```
# apply() has two main arguments, making it incredibly easy to use.
# X = data
# MARGIN = 1 means "move across rows"
# MARGIN = 2 means "move across columns"
# FUN = your function.

# Using only the numeric values of the iris data...
data <- iris[,1:4]

# Lets find the mean of each row.
apply(data,MARGIN = 1,mean)
```

```
##   [1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400 2.700
##  [12] 2.500 2.325 2.125 2.800 3.000 2.750 2.575 2.875 2.675 2.675 2.675
##  [23] 2.350 2.650 2.575 2.450 2.600 2.600 2.550 2.425 2.425 2.675 2.725
##  [34] 2.825 2.425 2.400 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675
##  [45] 2.800 2.375 2.675 2.350 2.675 2.475 4.075 3.900 4.100 3.275 3.850
##  [56] 3.575 3.975 2.900 3.850 3.300 2.875 3.650 3.300 3.775 3.350 3.900
##  [67] 3.650 3.400 3.600 3.275 3.925 3.550 3.800 3.700 3.725 3.850 3.950
##  [78] 4.100 3.725 3.200 3.200 3.150 3.400 3.850 3.600 3.875 4.000 3.575
##  [89] 3.500 3.325 3.425 3.775 3.400 2.900 3.450 3.525 3.525 3.675 2.925
## [100] 3.475 4.525 3.875 4.525 4.150 4.375 4.825 3.400 4.575 4.200 4.850
## [111] 4.200 4.075 4.350 3.800 4.025 4.300 4.200 5.100 4.875 3.675 4.525
## [122] 3.825 4.800 3.925 4.450 4.550 3.900 3.950 4.225 4.400 4.550 5.025
## [133] 4.250 3.925 3.925 4.775 4.425 4.200 3.900 4.375 4.450 4.350 3.875
## [144] 4.550 4.550 4.300 3.925 4.175 4.325 3.950
```

```r
# Now let's find the mean of each column
apply(data,MARGIN = 2,mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

This can be incredibly useful and powerful, because we can specify any function to perform the task we need – as long as it works on the data at hand.

sapply

```r
sapply(data,FUN = sd)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##    0.8280661    0.4358663    1.7652982    0.7622377
```

by

```r
by(iris[,1:4],INDICES = iris$Species,FUN = colMeans)
```

```
## iris$Species: setosa
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.006        3.428        1.462        0.246
## --------------------------------------------------------
## iris$Species: versicolor
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.936        2.770        4.260        1.326
## --------------------------------------------------------
## iris$Species: virginica
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        6.588        2.974        5.552        2.026
```

lapply

```r
L <- list(cats = 1:10, dogs = 11:20)
L
```

```
## $cats
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $dogs
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```r
lapply(L,FUN = mean)
```

```
## $cats
## [1] 5.5
##
## $dogs
## [1] 15.5
```

For a really great walk through on the apply family, see the swirl package.

The dplyr Approach

`dplyr` was designed to be a "grammar of data manipulation" – and it was quite successful at doing just that! With just a few simple combinations, one can manipulate data with ease. `dplyr` gets us:

- a clear syntax for data manipulation: "verbs" that do what you'd expect, i.e. the command matches the name.
- efficiency: the back-end of the functions are optimized in c++.

```
require(dplyr)
```

The Pipe

When we load dplyr, we activate the "pipe" (originally from the package `magrittr`). In simple terms, the pipe "passes" tasks from one function to the next, making it unnecessary to create a bunch of useless objects. This results in code that is easier to write and read – which is always great!

Let's consider running the same command three different ways: here we will take the standard deviation of a variable and then round it.

```
x <- rnorm(100,3,4) # generate a random variable

# (1) go object by object
x <- rnorm(100,3,4) # generate a random variable
x_sd <- sd(x)
x_sd_rounded <- round(x_sd,2)
x_sd_rounded
```

```
## [1] 4.14
```

```
# (2) nest the functions within each other
round(sd(rnorm(100,3,4)),2)
```

```
## [1] 4.08
```

```
# (3) pipe it
rnorm(100,3,4) %>% sd(.) %>% round(.,2)
```

```
## [1] 4.28
```

Again, the pipe (`%>%`) "passes" tasks between functions seamlessly, and it makes your code easier to read and debug. We can direct the data to a specific spot in a function using the pointer "`.`".

As we'll see, the pipe is a game changer, and when employed with the `dplyr` lexicon, you can really get stuff done.

dplyr "verbs"

The goal of the package is to have very simple verbs that do exactly what it sounds like they do. The language offers clarity to data manipulation, which makes it easy to translate what one has in his/her head into _R_eality!

We'll review a few (and some of the most useful) of these functions:

- filter() (and slice())
- select() (and rename())
- distinct()
- mutate() (and transmute())
- summarise()
- sample_n() and sample_frac()
- And more! Just check out the documentation.

Also, let's bring in some real **UCDP data** for the applied examples.

```
ucdp.loc <- "http://ucdp.uu.se/downloads/ucdpprio/ucdp-prio-acd-4-2016.csv"
ucdp <- read.csv(url(ucdp.loc),stringsAsFactors = F)
dim(ucdp) # dimensions
```

```
## [1] 2225   27
```

```
str(ucdp)
```

```
## 'data.frame':    2225 obs. of  27 variables:
##  $ ConflictId         : chr  "1-137" "1-137" "1-137" "1-137" ...
##  $ Location           : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
##  $ SideA              : chr  "Government of Afghanistan" "Government of Afghanistan" "Government of A
##  $ SideA2nd           : chr  "" "" "Government of Russia (Soviet Union)" "Government of Russia (Sovie
##  $ SideB              : chr  "PDPA" "Jam'iyyat-i Islami-yi Afghanistan" "Harakat-i Inqilab-i Islami-y
##  $ SideBID            : chr  "1133" "1134" "1135, 1141, 1136, 1137, 1134, 1138" "1135, 1141, 1136, 11
##  $ SideB2nd           : chr  "" "" "" "" ...
##  $ Incompatibility    : int  2 2 2 2 2 2 2 2 2 2 ...
##  $ TerritoryName      : chr  "" "" "" "" ...
##  $ Year               : int  1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 ...
##  $ IntensityLevel     : int  2 2 2 2 2 2 2 2 2 2 ...
##  $ CumulativeIntensity: int  1 1 1 1 1 1 1 1 1 1 ...
##  $ TypeOfConflict     : int  3 3 4 4 4 4 4 4 4 4 ...
##  $ StartDate          : chr  "1975-12-31" "1975-12-31" "1975-12-31" "1975-12-31" ...
##  $ StartPrec          : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ StartDate2         : chr  "1978-04-27" "1978-04-27" "1978-04-27" "1978-04-27" ...
##  $ StartPrec2         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ EpEnd              : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ EpEndDate          : chr  "" "" "" "" ...
##  $ EpEndPrec          : int  NA NA NA NA NA NA NA NA NA NA ...
##  $ GWNoA              : chr  "700" "700" "700" "700" ...
##  $ GWNoA2nd           : chr  "" "" "365" "365" ...
##  $ GWNoB              : chr  "" "" "" "" ...
##  $ GWNoB2nd           : chr  "" "" "" "" ...
##  $ GWNoLoc            : chr  "700" "700" "700" "700" ...
##  $ Region             : chr  "3" "3" "3" "3" ...
##  $ Version            : chr  "4.0-2016" "4.0-2016" "4.0-2016" "4.0-2016" ...
```

filter

Much like subset, you can filter allows you to select specific subsets of the data

```
filter(ucdp,Location=="Afghanistan" & StartDate=="2015-02-09")
```

```
##   ConflictId    Location                      SideA
## 1      1-288 Afghanistan Government of Afghanistan
##                                                               SideA2nd SideB
## 1 Government of Pakistan, Government of United States of America    IS
##   SideBID SideB2nd Incompatibility TerritoryName Year IntensityLevel
## 1    1076                        1 Islamic State 2015              1
##   CumulativeIntensity TypeOfConflict  StartDate StartPrec StartDate2
## 1                   0              4 2015-02-09         1 2015-03-03
##   StartPrec2 EpEnd EpEndDate EpEndPrec GWNoA GWNoA2nd GWNoB GWNoB2nd
## 1          1     0                  NA   700   770, 2
##   GWNoLoc Region  Version
## 1     700      3 4.0-2016
```

select

Select specific columns

```
# Select specific columns
select(ucdp[1:3,],Location,SideA,Year)
```

```
##      Location                     SideA Year
## 1 Afghanistan Government of Afghanistan 1978
## 2 Afghanistan Government of Afghanistan 1979
## 3 Afghanistan Government of Afghanistan 1980
```

```
# You can even employ ranges!
select(ucdp[1:2,],ConflictId:SideA)
```

```
##   ConflictId    Location                      SideA
## 1      1-137 Afghanistan Government of Afghanistan
## 2      1-137 Afghanistan Government of Afghanistan
```

```
# Or subset out columns you DON'T want (using the -)
select(ucdp[1:2,],-(ConflictId:GWNoA))
```

```
##   GWNoA2nd GWNoB GWNoB2nd GWNoLoc Region  Version
## 1                             700      3 4.0-2016
## 2                             700      3 4.0-2016
```

```
# Or select a column and RENAME IT simultaneously
select(ucdp[1:2,],Country=Location,Year,SideA)
```

```
##       Country Year                     SideA
## 1 Afghanistan 1978 Government of Afghanistan
## 2 Afghanistan 1979 Government of Afghanistan
```

rename

As the name would imply, rename variables

```
rename(ucdp[1:2,],country=Location,ID=ConflictId,government=SideA)
```

```
##       ID     country                 government SideA2nd
## 1 1-137 Afghanistan Government of Afghanistan
## 2 1-137 Afghanistan Government of Afghanistan
##                                  SideB SideBID SideB2nd Incompatibility
## 1                                 PDPA    1133                        2
## 2 Jam'iyyat-i Islami-yi Afghanistan    1134                        2
##   TerritoryName Year IntensityLevel CumulativeIntensity TypeOfConflict
## 1               1978              2                   1              3
## 2               1979              2                   1              3
##    StartDate StartPrec StartDate2 StartPrec2 EpEnd EpEndDate EpEndPrec
## 1 1975-12-31         5 1978-04-27          1     0                  NA
## 2 1975-12-31         5 1978-04-27          1     0                  NA
##   GWNoA GWNoA2nd GWNoB GWNoB2nd GWNoLoc Region  Version
## 1   700                             700      3 4.0-2016
## 2   700                             700      3 4.0-2016
```

distinct

The same as `unique`. Here we'll examine the distinct locations for the first 100 observations.

```
select(ucdp[1:100,],Location) %>% distinct(.)
```

```
##                                                 Location
## 1                                            Afghanistan
## 2                               United States of America
## 3 Afghanistan, United Kingdom, United States of America
## 4                     Afghanistan, Russia (Soviet Union)
## 5                                Albania, United Kingdom
## 6                                                Algeria
## 7                                       Algeria, Morocco
## 8                                                 Angola
```

```
# or combinations: distinct locations and distinct start-dates of conflicts
select(ucdp[1:100,],Location,StartDate) %>% distinct(.)
```

```
##                                                  Location  StartDate
## 1                                             Afghanistan 1975-12-31
## 2                                             Afghanistan 2015-02-09
## 3                                United States of America 2001-09-11
## 4  Afghanistan, United Kingdom, United States of America 2001-10-07
## 5                      Afghanistan, Russia (Soviet Union) 1979-12-27
## 6                                 Albania, United Kingdom 1946-10-22
## 7                                                 Algeria 1954-11-01
## 8                                                 Algeria 1985-08-27
## 9                                        Algeria, Morocco 1963-10-08
## 10                                                 Angola 1961-02-04
```

mutate

Create a new variable on the fly – this is similar to a function called `transform` which is in base `R`. However, `mutate` allows us to utilize columns we *just* created.

Below we are going to use the start date and create a variable called "start_year" and then subtract it by the current year to get a "duration" count.

```
dd <- mutate(ucdp,
             start_year = as.numeric(format.Date(StartDate,"%Y")),
             duration = Year - start_year)
dd[1:5,c("Location","Year","duration")]
```

```
##      Location Year duration
## 1 Afghanistan 1978        3
## 2 Afghanistan 1979        4
## 3 Afghanistan 1980        5
## 4 Afghanistan 1981        6
## 5 Afghanistan 1982        7
```

If you wish to *only* retain the columns you created, use `transmute`.

```
dd2  <- transmute(ucdp,
               start_year = as.numeric(format.Date(StartDate,"%Y")),
               duration = Year - start_year)
head(dd2)
```

```
##   start_year duration
## 1       1975        3
## 2       1975        4
## 3       1975        5
## 4       1975        6
## 5       1975        7
## 6       1975        8
```

summarize

Allows you to quickly summarize the data by whatever parameters you want and then prints it as a single row. This function can be useful for data exploration.

```
# Using the duration variable we just created to find the average duration of a
# conflict across the data and the standard deviation.
summarize(dd,ave_duration = mean(duration),sd_duration= sd(duration))
```

```
##   ave_duration sd_duration
## 1     17.94472    16.19125
```

sample_n and sample_frac

Allows you to randomly sample a specific number of rows (`sample_n`) or a specific proportion of the data (`sample_frac`)

```r
ucdp %>% select(.,Location,Year,SideA) %>% sample_n(.,4)
```

```
##          Location Year                      SideA
## 710         India 1979        Government of India
## 914     Indonesia 1999    Government of Indonesia
## 554      Ethiopia 1990     Government of Ethiopia
## 1887 South Africa 1979 Government of South Africa
```

```r
ucdp %>% select(.,Location,Year,SideA) %>% sample_frac(.,.002)
```

```
##                    Location Year                         SideA
## 466  Egypt, United Kingdom 1952           Government of Egypt
## 1559             Nicaragua 1978       Government of Nicaragua
## 1431       Myanmar (Burma) 1957 Government of Myanmar (Burma)
## 633              Guatemala 1963       Government of Guatemala
```

group_by

This is a gem of a function. Group the data by a specific variable and then perform specific tasks.

```r
# Again using our duration variable...
dd %>% group_by(.,Location) %>%
  summarize(.,total_years = n(),
            ave_dur = round(mean(duration)),
            # n_distinct == length(unique(.))
            no_conflicts = n_distinct(StartDate)) %>%
  sample_n(.,5) # Randomly sample
```

```
## Source: local data frame [5 x 4]
##
##                         Location total_years ave_dur no_conflicts
##                            (chr)       (int)   (dbl)        (int)
## 1                     Bangladesh          19       8            2
## 2           Cambodia (Kampuchea)          38      13            2
## 3                       Thailand          23      35            2
## 4                         Greece           4       2            1
## 5 Cambodia (Kampuchea), Thailand           3      14            1
```

The above demonstrates how one can employ powerful manipulations of the data rather succinctly. Here we now know the average duration in a country, the total number of conflict years (i.e. how many years the country is in the data), and the total number of unique conflicts.

Finally, the `tbl_df` is a useful print function when you don't have a lot of room to display a dataset.

```r
tbl_df(ucdp)
```

```
## Source: local data frame [2,225 x 27]
##
##    ConflictId    Location                      SideA
##         (chr)       (chr)                      (chr)
## 1       1-137 Afghanistan Government of Afghanistan
```

```
## 2       1-137 Afghanistan Government of Afghanistan
## 3       1-137 Afghanistan Government of Afghanistan
## 4       1-137 Afghanistan Government of Afghanistan
## 5       1-137 Afghanistan Government of Afghanistan
## 6       1-137 Afghanistan Government of Afghanistan
## 7       1-137 Afghanistan Government of Afghanistan
## 8       1-137 Afghanistan Government of Afghanistan
## 9       1-137 Afghanistan Government of Afghanistan
## 10      1-137 Afghanistan Government of Afghanistan
## ..        ...         ...                        ...
## Variables not shown: SideA2nd (chr), SideB (chr), SideBID (chr), SideB2nd
##   (chr), Incompatibility (int), TerritoryName (chr), Year (int),
##   IntensityLevel (int), CumulativeIntensity (int), TypeOfConflict (int),
##   StartDate (chr), StartPrec (int), StartDate2 (chr), StartPrec2 (int),
##   EpEnd (int), EpEndDate (chr), EpEndPrec (int), GWNoA (chr), GWNoA2nd
##   (chr), GWNoB (chr), GWNoB2nd (chr), GWNoLoc (chr), Region (chr), Version
##   (chr)
```

Advanced Data Manipulations

Lags

The need to lag a variable is a common in political science analysis, but the task is not always straightforward in R. Consider the following data: if we use the`lag()` feature, we get the desired result, but not in a way that takes into account the groupings in the data.

```
data <- data.frame(country=c(rep("A",10),rep("B",10)),
                   year=rep(1995:2004,2),
                   v1=runif(20,0,10),stringsAsFactors = F)

data$lag1 <- lag(data$v1,1) # Lag

# Country A's values "bleed into" Country B's...no good.
data
```

```
##    country year        v1      lag1
## 1        A 1995 9.2124683        NA
## 2        A 1996 6.2989772 9.2124683
## 3        A 1997 2.2580940 6.2989772
## 4        A 1998 9.0481356 2.2580940
## 5        A 1999 1.7582181 9.0481356
## 6        A 2000 3.0038076 1.7582181
## 7        A 2001 8.0946926 3.0038076
## 8        A 2002 2.8788433 8.0946926
## 9        A 2003 5.7717121 2.8788433
## 10       A 2004 0.6735399 5.7717121
## 11       B 1995 7.9438699 0.6735399
## 12       B 1996 4.3447079 7.9438699
## 13       B 1997 7.0452568 4.3447079
## 14       B 1998 9.0827410 7.0452568
## 15       B 1999 4.4980984 9.0827410
## 16       B 2000 8.0441715 4.4980984
```

```
## 17       B 2001 3.3350871 8.0441715
## 18       B 2002 4.2226494 3.3350871
## 19       B 2003 7.0047119 4.2226494
## 20       B 2004 8.7837692 7.0047119
```

To get around this, we need to lag *by subgroup*.

First, we can utilize a **loop** to deal with the sub-grouping issue.

```
groups = unique(data$country)
for (i in 1:length(groups) ){
  lagged_value = lag(data[data$country==groups[i],"v1"])
  data[data$country==groups[i],"lag2"] <- lagged_value
}
data
```

```
##    country year        v1      lag1     lag2
## 1        A 1995 9.2124683        NA       NA
## 2        A 1996 6.2989772 9.2124683 9.212468
## 3        A 1997 2.2580940 6.2989772 6.298977
## 4        A 1998 9.0481356 2.2580940 2.258094
## 5        A 1999 1.7582181 9.0481356 9.048136
## 6        A 2000 3.0038076 1.7582181 1.758218
## 7        A 2001 8.0946926 3.0038076 3.003808
## 8        A 2002 2.8788433 8.0946926 8.094693
## 9        A 2003 5.7717121 2.8788433 2.878843
## 10       A 2004 0.6735399 5.7717121 5.771712
## 11       B 1995 7.9438699 0.6735399       NA
## 12       B 1996 4.3447079 7.9438699 7.943870
## 13       B 1997 7.0452568 4.3447079 4.344708
## 14       B 1998 9.0827410 7.0452568 7.045257
## 15       B 1999 4.4980984 9.0827410 9.082741
## 16       B 2000 8.0441715 4.4980984 4.498098
## 17       B 2001 3.3350871 8.0441715 8.044171
## 18       B 2002 4.2226494 3.3350871 3.335087
## 19       B 2003 7.0047119 4.2226494 4.222649
## 20       B 2004 8.7837692 7.0047119 7.004712
```

Which worked! But that was a lot of code for something that should be easier. The **`dplyr` approach** offers it's own solution, and it's quick and clean. Here we are first grouping the data by country, then creating a new variable which is a one-year lag (also, we are ordering the data by year).

```
data <- data %>%
  group_by(country) %>%
  mutate(lag3 = lag(v1, order_by=year))
data
```

```
## Source: local data frame [20 x 6]
## Groups: country [2]
##
##    country  year        v1      lag1     lag2     lag3
##      (chr) (int)     (dbl)     (dbl)    (dbl)    (dbl)
## 1        A  1995 9.2124683        NA       NA       NA
```

```
## 2         A  1996 6.2989772 9.2124683 9.212468 9.212468
## 3         A  1997 2.2580940 6.2989772 6.298977 6.298977
## 4         A  1998 9.0481356 2.2580940 2.258094 2.258094
## 5         A  1999 1.7582181 9.0481356 9.048136 9.048136
## 6         A  2000 3.0038076 1.7582181 1.758218 1.758218
## 7         A  2001 8.0946926 3.0038076 3.003808 3.003808
## 8         A  2002 2.8788433 8.0946926 8.094693 8.094693
## 9         A  2003 5.7717121 2.8788433 2.878843 2.878843
## 10        A  2004 0.6735399 5.7717121 5.771712 5.771712
## 11        B  1995 7.9438699 0.6735399       NA       NA
## 12        B  1996 4.3447079 7.9438699 7.943870 7.943870
## 13        B  1997 7.0452568 4.3447079 4.344708 4.344708
## 14        B  1998 9.0827410 7.0452568 7.045257 7.045257
## 15        B  1999 4.4980984 9.0827410 9.082741 9.082741
## 16        B  2000 8.0441715 4.4980984 4.498098 4.498098
## 17        B  2001 3.3350871 8.0441715 8.044171 8.044171
## 18        B  2002 4.2226494 3.3350871 3.335087 3.335087
## 19        B  2003 7.0047119 4.2226494 4.222649 4.222649
## 20        B  2004 8.7837692 7.0047119 7.004712 7.004712
```

The most straightforward way to deal with any data manipulation is to really think through what is going on. Note that we can lag to any depth using `lag(variable,k)`, where k is the number of units one wishes to lag by.

Time Since An Event (Duration Counters)

In event history analysis, a duration variable counts the time since an event last occurred. This is also central to calculating a cubic polynomial to deal with time specific effects in a glm.

For an example, consider the following data. Here we have a country-year, with a 1 denoting the occurrence of some event.

```
event_data <- data.frame(country="Nigeria",
                         year=1999:2010,
                         event = c(1,0,0,0,1,0,0,0,0,
                                   1,0,0))
event_data
```

```
##    country year event
## 1  Nigeria 1999     1
## 2  Nigeria 2000     0
## 3  Nigeria 2001     0
## 4  Nigeria 2002     0
## 5  Nigeria 2003     1
## 6  Nigeria 2004     0
## 7  Nigeria 2005     0
## 8  Nigeria 2006     0
## 9  Nigeria 2007     0
## 10 Nigeria 2008     1
## 11 Nigeria 2009     0
## 12 Nigeria 2010     0
```

To create a duration counter, we can do one of three things: run a loop, use dplyr, use a duration package. My preference is for the dplyr approach, as it fits in the same framework as most other manipulations, and it only requires a few lines of code, but I will show all three.

(1) using a **Loop**

```
j = 0
event_data$time_since <- 0
for(i in 1:nrow(event_data)){
  if(event_data$event[i]==1){
    j <- 0
    event_data$time_since[i] <- j
  }else{
    j <- j + 1
    event_data$time_since[i] <- j
  }
}
event_data
```

```
##    country year event time_since
## 1  Nigeria 1999     1          0
## 2  Nigeria 2000     0          1
## 3  Nigeria 2001     0          2
## 4  Nigeria 2002     0          3
## 5  Nigeria 2003     1          0
## 6  Nigeria 2004     0          1
## 7  Nigeria 2005     0          2
## 8  Nigeria 2006     0          3
## 9  Nigeria 2007     0          4
## 10 Nigeria 2008     1          0
## 11 Nigeria 2009     0          1
## 12 Nigeria 2010     0          2
```

(2) **`dplyr` approach**: here we are grouping the data by country (assuming that we have multiple countries), and a number unique variable that we are calling "spell_id", which creates unique periods for each spell, then we are counting up the number of rows for each spell and subtracting by one (to exclude the initiation period); finally, we are dropping our unique ID variable and ungrouping the data.

```
event_data2 <- event_data %>%
  group_by(country, spell_id = cumsum(event == 1)) %>%
  mutate(time_since2 = row_number()-1) %>%
  select(-spell_id) %>% ungroup(.)
event_data2
```

```
## Source: local data frame [12 x 6]
##
##    spell_id country  year event time_since time_since2
##       (int)  (fctr) (int) (dbl)      (dbl)       (dbl)
## 1         1 Nigeria  1999     1          0           0
## 2         1 Nigeria  2000     0          1           1
## 3         1 Nigeria  2001     0          2           2
## 4         1 Nigeria  2002     0          3           3
```

```
## 5         2 Nigeria  2003     1          0           0
## 6         2 Nigeria  2004     0          1           1
## 7         2 Nigeria  2005     0          2           2
## 8         2 Nigeria  2006     0          3           3
## 9         2 Nigeria  2007     0          4           4
## 10        3 Nigeria  2008     1          0           0
## 11        3 Nigeria  2009     0          1           1
## 12        3 Nigeria  2010     0          2           2
```

(3) Use a survival analysis **package**: there are a number of duration modeling packages that will also perform this task for you. One of the better one's out there is called, `spduration`.

```
require(spduration)
```

```
## Loading required package: spduration
```

```
event_data3 <- add_duration(event_data,
                            y="event",
                            unitID="country",
                            tID="year",
                            freq="year")

# Clean so that it doesn't count the initiation period
event_data3[event_data3$event==1,"duration"] <- 0
event_data3[,c(1:4,11)]
```

```
##    country year event time_since duration
## 12 Nigeria 1999     1          0        0
## 8  Nigeria 2000     0          1        1
## 10 Nigeria 2001     0          2        2
## 11 Nigeria 2002     0          3        3
## 9  Nigeria 2003     1          0        0
## 3  Nigeria 2004     0          1        1
## 6  Nigeria 2005     0          2        2
## 7  Nigeria 2006     0          3        3
## 4  Nigeria 2007     0          4        4
## 5  Nigeria 2008     1          0        0
## 1  Nigeria 2009     0          1        1
## 2  Nigeria 2010     0          2        2
```

Expansions and Contractions

Sometimes we need to **expand episodal data** out so that we have an observation for each year. Note that more often than not, this is unwise to do, as we are making an assumption that the indicator we wish to expand maps equally onto all temporal units that we are expanding by — but if you do need to do it, here is how we'd get it done.

```
data <- data.frame(country = c("Russia","US","Belize","Mexico"),
                   start=c(1994,1992,2000,1990),
                   end=c(1998,1995,2002,2003),
                   eventA=c(1,0,1,0),
                   measureA=c(56.89,72.90,81.32,34.89))
data
```

```
##   country start  end eventA measureA
## 1  Russia  1994 1998      1    56.89
## 2      US  1992 1995      0    72.90
## 3  Belize  2000 2002      1    81.32
## 4  Mexico  1990 2003      0    34.89
```

Using the "to" and "from" dates, we can expand the time variable along a sequence. That said, the `seq()` function can only hand a value with a length of 1, meaning that it cannot simultaneously process a vector of numbers. We get around this by running a loop.

```
out <- c() # Create a container
for(i in 1:nrow(data)){
  data_range <- seq(from=data$start[i],to=data$end[i],by=1)
  dd <- data.frame(year=data_range)
  vars = data[i,c("country","eventA","measureA")] # The other vars
  rownames(vars) <- NULL
  data_out <- data.frame(dd,vars)
  out <- rbind(out,data_out)
}
out
```

```
##    year country eventA measureA
## 1  1994  Russia      1    56.89
## 2  1995  Russia      1    56.89
## 3  1996  Russia      1    56.89
## 4  1997  Russia      1    56.89
## 5  1998  Russia      1    56.89
## 6  1992      US      0    72.90
## 7  1993      US      0    72.90
## 8  1994      US      0    72.90
## 9  1995      US      0    72.90
## 10 2000  Belize      1    81.32
## 11 2001  Belize      1    81.32
## 12 2002  Belize      1    81.32
## 13 1990  Mexico      0    34.89
## 14 1991  Mexico      0    34.89
## 15 1992  Mexico      0    34.89
## 16 1993  Mexico      0    34.89
## 17 1994  Mexico      0    34.89
## 18 1995  Mexico      0    34.89
## 19 1996  Mexico      0    34.89
## 20 1997  Mexico      0    34.89
## 21 1998  Mexico      0    34.89
## 22 1999  Mexico      0    34.89
## 23 2000  Mexico      0    34.89
## 24 2001  Mexico      0    34.89
## 25 2002  Mexico      0    34.89
## 26 2003  Mexico      0    34.89
```

Contracting data back down is straightforward. We've already covered this with the `summarise` function from `dplyr`.

```r
out %>% group_by(country) %>%
  summarise(.,start=min(year),end=max(year),eventA=max(eventA),measureA=max(measureA))
```

```
## Source: local data frame [4 x 5]
##
##   country start   end eventA measureA
##    (fctr) (dbl) (dbl)  (dbl)    (dbl)
## 1  Belize  2000  2002      1    81.32
## 2  Mexico  1990  2003      0    34.89
## 3  Russia  1994  1998      1    56.89
## 4      US  1992  1995      0    72.90
```

Dealing with Wide Data

Though less of an issue in political science than other social sciences, sometimes we need a way to deal with "wide data". This is most often an issue when dealing with any `World Bank` data, as it often comes in a format where the years are printed as individual data. This is in contrast to the "long data" that we are accustomed to where we just have country-years.

```r
data <- data.frame(country=c("A","B","C","D","E"),
                   matrix(rnorm(50,6,10),nrow=5,ncol=10))
colnames(data)[2:11] <- 2000:2009
data
```

```
##   country     2000      2001     2002      2003      2004      2005
## 1       A 12.76673 11.113916 15.66914  4.714978  9.591855 14.666206
## 2       B  2.90933 27.018801 11.03697 -9.562899  6.394411 -4.717597
## 3       C 13.69114 -5.153421 12.73798  4.420895  4.251806 17.051196
## 4       D 23.91547  2.176639 29.26806 11.491380 18.969253 10.685338
## 5       E 25.15310 11.254971 -1.20707 14.166521 -4.684688 15.797733
##        2006       2007       2008     2009
## 1 -3.514298 20.6924414  11.876263 2.193581
## 2  7.534054 10.7422626  -8.036541 6.156117
## 3 -9.105811  2.4975825  17.596627 3.703154
## 4 17.604374  0.9244678 -21.363612 3.181310
## 5  7.474524  1.4844322  10.733546 1.848861
```

The main way to "reshape" the data is to use a loop or the package `rshape2`.

(1) using a **loop**

```r
out <- c() # create a container
for( country in 1:nrow(data)){
  x <- t(data[country,-1]) %>% as.data.frame(.)
  colnames(x) <- "indicator"
  x$year <- rownames(x)
  rownames(x) <- NULL
  x$country <- data[country,1]
  the_goods <- x[,c("country","year","indicator")]
  out <- rbind(out,the_goods)
}
out
```

```
##    country year   indicator
## 1        A 2000  12.7667271
## 2        A 2001  11.1139160
## 3        A 2002  15.6691363
## 4        A 2003   4.7149779
## 5        A 2004   9.5918549
## 6        A 2005  14.6662058
## 7        A 2006  -3.5142982
## 8        A 2007  20.6924414
## 9        A 2008  11.8762627
## 10       A 2009   2.1935814
## 11       B 2000   2.9093304
## 12       B 2001  27.0188011
## 13       B 2002  11.0369740
## 14       B 2003  -9.5628988
## 15       B 2004   6.3944114
## 16       B 2005  -4.7175971
## 17       B 2006   7.5340543
## 18       B 2007  10.7422626
## 19       B 2008  -8.0365410
## 20       B 2009   6.1561167
## 21       C 2000  13.6911444
## 22       C 2001  -5.1534207
## 23       C 2002  12.7379766
## 24       C 2003   4.4208952
## 25       C 2004   4.2518056
## 26       C 2005  17.0511957
## 27       C 2006  -9.1058105
## 28       C 2007   2.4975825
## 29       C 2008  17.5966274
## 30       C 2009   3.7031545
## 31       D 2000  23.9154720
## 32       D 2001   2.1766387
## 33       D 2002  29.2680627
## 34       D 2003  11.4913801
## 35       D 2004  18.9692533
## 36       D 2005  10.6853384
## 37       D 2006  17.6043744
## 38       D 2007   0.9244678
## 39       D 2008 -21.3636116
## 40       D 2009   3.1813096
## 41       E 2000  25.1531009
## 42       E 2001  11.2549711
## 43       E 2002  -1.2070702
## 44       E 2003  14.1665206
## 45       E 2004  -4.6846878
## 46       E 2005  15.7977330
## 47       E 2006   7.4745240
## 48       E 2007   1.4844322
## 49       E 2008  10.7335463
## 50       E 2009   1.8488607
```

Nice! But wow that was a lot of code... This is where the `reshape2` package becomes really useful.

(2) `melt()` in the `reshape2` package: Here we are melting the data down from something that was wide to something that is long, and we are doing so by a specific ID (which in this case is the country name).

```
require(reshape2) # load the package

data2 <- melt(data,id="country")
colnames(data2) <- c("country","year","indicator")
data2
```

```
FALSE    country year   indicator
FALSE 1        A 2000  12.7667271
FALSE 2        B 2000   2.9093304
FALSE 3        C 2000  13.6911444
FALSE 4        D 2000  23.9154720
FALSE 5        E 2000  25.1531009
FALSE 6        A 2001  11.1139160
FALSE 7        B 2001  27.0188011
FALSE 8        C 2001  -5.1534207
FALSE 9        D 2001   2.1766387
FALSE 10       E 2001  11.2549711
FALSE 11       A 2002  15.6691363
FALSE 12       B 2002  11.0369740
FALSE 13       C 2002  12.7379766
FALSE 14       D 2002  29.2680627
FALSE 15       E 2002  -1.2070702
FALSE 16       A 2003   4.7149779
FALSE 17       B 2003  -9.5628988
FALSE 18       C 2003   4.4208952
FALSE 19       D 2003  11.4913801
FALSE 20       E 2003  14.1665206
FALSE 21       A 2004   9.5918549
FALSE 22       B 2004   6.3944114
FALSE 23       C 2004   4.2518056
FALSE 24       D 2004  18.9692533
FALSE 25       E 2004  -4.6846878
FALSE 26       A 2005  14.6662058
FALSE 27       B 2005  -4.7175971
FALSE 28       C 2005  17.0511957
FALSE 29       D 2005  10.6853384
FALSE 30       E 2005  15.7977330
FALSE 31       A 2006  -3.5142982
FALSE 32       B 2006   7.5340543
FALSE 33       C 2006  -9.1058105
FALSE 34       D 2006  17.6043744
FALSE 35       E 2006   7.4745240
FALSE 36       A 2007  20.6924414
FALSE 37       B 2007  10.7422626
FALSE 38       C 2007   2.4975825
FALSE 39       D 2007   0.9244678
FALSE 40       E 2007   1.4844322
FALSE 41       A 2008  11.8762627
FALSE 42       B 2008  -8.0365410
FALSE 43       C 2008  17.5966274
FALSE 44       D 2008 -21.3636116
```

```
FALSE 45        E 2008  10.7335463
FALSE 46        A 2009   2.1935814
FALSE 47        B 2009   6.1561167
FALSE 48        C 2009   3.7031545
FALSE 49        D 2009   3.1813096
FALSE 50        E 2009   1.8488607
```

Way easier! And just one line…