Data Processing with dplyr & tidyr

Brad Boehmke

February 13th, 2015

This tutorial can be accessed at (http://rpubs.com/bradleyboehmke/data_wrangling)http://rpubs.com/bradleyboehmke/data_wrangling (http://rpubs.com/bradleyboehmke/data_wrangling)

Introduction

Analytic Process

Analysts tend to follow 4 fundamental processes to turn data into understanding, knowledge & insight:

1. **Data manipulation**
2. Data visualization
3. Statistical analysis/modeling
4. Deployment of results

This tutorial will focus on **data manipulation**

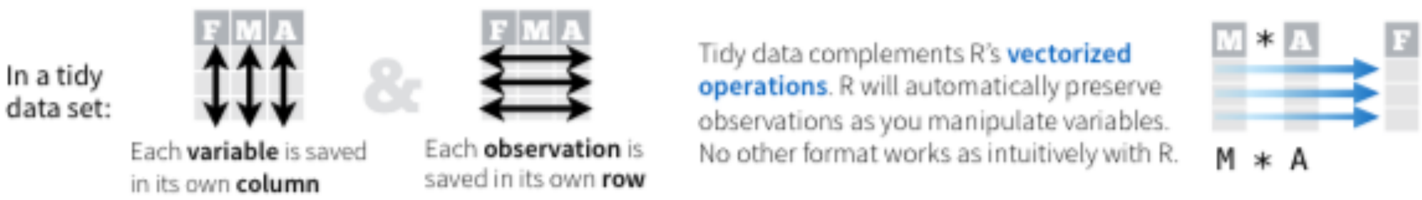
Data Manipulation

> It is often said that 80% of data analysis is spent on the process of cleaning and preparing the data. (Dasu and Johnson, 2003)

Well structured data serves two purposes:

1. Makes data suitable for software processing whether that be mathematical functions, visualization, etc.
2. Reveals information and insights

Hadley Wickham's paper on Tidy Data (http://vita.had.co.nz/papers/tidy-data.html) provides a great explanation behind the concept of "tidy data"



Why Use tidyr & dplyr

- Although many fundamental data processing functions exist in R, they have been a bit convoluted to date and have lacked consistent coding and the ability to easily *flow* together → leads to difficult-to-read nested functions and/or *choppy* code.
- R Studio (http://www.rstudio.com/) is driving a lot of new packages to collate data management tasks and better integrate them with other analysis activities → led by Hadley Wickham (https://twitter.com/hadleywickham) & the R Studio team (http://www.rstudio.com/about/) → Garrett Grolemund (https://twitter.com/StatGarrett), Winston Chang (https://twitter.com/winston_chang), Yihui Xie (https://twitter.com/xieyihui) among others.
- As a result, a lot of data processing tasks are becoming packaged in more cohesive and consistent ways → leads to:
  - More efficient code
  - Easier to remember syntax
  - Easier to read syntax

Packages Utilized

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

tidyr and dplyr packages provide fundamental functions for Cleaning, Processing, & Manipulating Data

- tidyr
  - `gather()`
  - `spread()`
  - `separate()`
  - `unite()`
- dplyr
  - `select()`
  - `filter()`
  - `group_by()`
  - `summarise()`
  - `arrange()`
  - `join()`
  - `mutate()`

Go to top

%>% Operator

Although not required, the tidyr and dplyr packages make use of the pipe operator `%>%` developed by Stefan Milton Bache (https://twitter.com/stefanbache) in the R package magrittr (http://cran.r-project.org/web/packages/magrittr/magrittr.pdf). Although all the functions in tidyr and dplyr *can be used without the pipe operator*, one of the great conveniences these packages provide is the ability to string multiple functions together by incorporating `%>%` .

This operator will forward a value, or the result of an expression, into the next function call/expression. For instance a function to filter data can be written as:

filter(data, variable == *numeric_value*)
or
data %>% filter(variable == *numeric_value*)

Both functions complete the same task and the benefit of using `%>%` is not evident; however, when you desire to perform multiple functions its advantage becomes obvious. For instance, if we want to filter some data, summarize it, and then order the summarized results we would write it out as:

Nested Option:

```
arrange(
        summarize(
            filter(data, variable == numeric_value),
            Total = sum(variable)
        ),
    desc(Total)
)
```

or

Multiple Object Option:

```
a <- filter(data, variable == numeric_value)
b <- summarise(a, Total = sum(variable))
c <- arrange(b, desc(Total))
```

or

%>% Option:

```
data %>%
       filter(variable == “value”) %>%
       summarise(Total = sum(variable)) %>%
       arrange(desc(Total))
```

As your function tasks get longer the `%>%` operator becomes more efficient *and* makes your code more legible. In addition, although not covered in this tutorial, the `%>%` operator allows you to flow from data manipulation tasks straight into vizualization functions *(via ggplot and ggvis)* and also into many analytic functions.

To learn more about the `%>%` operator and the magrittr package visit any of the following:

- (http://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html)http://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html (http://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html)
- (http://www.r-bloggers.com/simpler-r-coding-with-pipes-the-present-and-future-of-the-magrittr-package/)http://www.r-bloggers.com/simpler-r-coding-with-pipes-the-present-and-future-of-the-magrittr-package/ (http://www.r-bloggers.com/simpler-r-coding-with-pipes-the-present-and-future-of-the-magrittr-package/)
- (http://blog.revolutionanalytics.com/2014/07/magrittr-simplifying-r-code-with-pipes.html)http://blog.revolutionanalytics.com/2014/07/magrittr-simplifying-r-code-with-pipes.html (http://blog.revolutionanalytics.com/2014/07/magrittr-simplifying-r-code-with-pipes.html)

Go to top

tidyr Operations

There are four fundamental functions of data tidying:

- `gather()` takes multiple columns, and gathers them into key-value pairs: it makes "wide" data longer
- `spread()` takes two columns (key & value) and spreads in to multiple columns, it makes "long" data wider
- `separate()` splits a single column into multiple columns
- `unite()` combines multiple columns into a single column

gather() function:

Objective: Reshaping wide format to long format

Description: There are times when our data is considered unstacked and a common attribute of concern is spread out across columns. To reformat the data such that these common attributes are *gathered* together as a single variable, the `gather()` function will take multiple columns and collapse them into key-value pairs, duplicating all other columns as needed.

Complement to: `spread()`

messy

id	trt	work.T1	home.T1	work.T2	home.T2
1	treatment	0.08513597	0.6158293	0.1135090	0.05190332
2	control	0.22543662	0.4296715	0.5959253	0.26417767
3	treatment	0.27453052	0.6516557	0.3580500	0.39879073
4	control	0.27230507	0.5677378	0.4288094	0.83613414

tidier

id	trt	key	time
1	treatment	work.T1	0.08513597
2	control	work.T1	0.22543662
3	treatment	work.T1	0.27453052
4	control	work.T1	0.27230507
1	treatment	home.T1	0.61582931
2	control	home.T1	0.42967153
3	treatment	home.T1	0.65165567
4	control	home.T1	0.56773775
1	treatment	work.T2	0.11350898
2	control	work.T2	0.59592531
3	treatment	work.T2	0.35804998
4	control	work.T2	0.42880942
1	treatment	home.T2	0.05190332
2	control	home.T2	0.26417767
3	treatment	home.T2	0.39879073
4	control	home.T2	0.83613414

```
Function:       gather(data, key, value, ..., na.rm = FALSE, convert = FALSE)
Same as:        data %>% gather(key, value, ..., na.rm = FALSE, convert = FALSE)

Arguments:
        data:           data frame
        key:            column name representing new variable
        value:          column name representing variable values
        ...:            names of columns to gather (or not gather)
        na.rm:          option to remove observations with missing values (represe
nted by NAs)
        convert:        if TRUE will automatically convert values to logical, inte
ger, numeric, complex or
                        factor as appropriate
```

Example

We'll start with the following data set:

```
## Source: local data frame [12 x 6]
##
##    Group Year Qtr.1 Qtr.2 Qtr.3 Qtr.4
## 1      1 2006    15    16    19    17
## 2      1 2007    12    13    27    23
## 3      1 2008    22    22    24    20
## 4      1 2009    10    14    20    16
## 5      2 2006    12    13    25    18
## 6      2 2007    16    14    21    19
## 7      2 2008    13    11    29    15
## 8      2 2009    23    20    26    20
## 9      3 2006    11    12    22    16
## 10     3 2007    13    11    27    21
## 11     3 2008    17    12    23    19
## 12     3 2009    14     9    31    24
```

This data is considered wide since the *time* variable (represented as quarters) is structured such that each quarter represents a variable. To re-structure the time component as an individual variable, we can *gather* each quarter within one column variable and also *gather* the values associated with each quarter in a second column variable.

```
long_DF <- DF %>% gather(Quarter, Revenue, Qtr.1:Qtr.4)
head(long_DF, 24)  # note, for brevity, I only show the data for the first two yea
rs
```

```
## Source: local data frame [24 x 4]
##
##    Group Year Quarter Revenue
## 1      1 2006   Qtr.1      15
## 2      1 2007   Qtr.1      12
## 3      1 2008   Qtr.1      22
## 4      1 2009   Qtr.1      10
## 5      2 2006   Qtr.1      12
## 6      2 2007   Qtr.1      16
## 7      2 2008   Qtr.1      13
## 8      2 2009   Qtr.1      23
## 9      3 2006   Qtr.1      11
## 10     3 2007   Qtr.1      13
## ..   ...  ...     ...     ...
```

```
These all produce the same results:
        DF %>% gather(Quarter, Revenue, Qtr.1:Qtr.4)
        DF %>% gather(Quarter, Revenue, -Group, -Year)
        DF %>% gather(Quarter, Revenue, 3:6)
        DF %>% gather(Quarter, Revenue, Qtr.1, Qtr.2, Qtr.3, Qtr.4)

Also note that if you do not supply arguments for na.rm or convert values then the
defaults are used
```

Go to top

separate( ) function:

Objective: Splitting a single variable into two

Description: Many times a single column variable will capture multiple variables, or even parts of a variable you just don't care about. Some examples include:

```
##   Grp_Ind    Yr_Mo       City_State        First_Last Extra_variable
## 1     1.a 2006_Jan      Dayton (OH) George Washington   XX01person_1
## 2     1.b 2006_Feb Grand Forks (ND)        John Adams   XX02person_2
## 3     1.c 2006_Mar       Fargo (ND)  Thomas Jefferson   XX03person_3
## 4     2.a 2007_Jan   Rochester (MN)     James Madison   XX04person_4
## 5     2.b 2007_Feb     Dubuque (IA)      James Monroe   XX05person_5
## 6     2.c 2007_Mar Ft. Collins (CO)        John Adams   XX06person_6
## 7     3.a 2008_Jan   Lake City (MN)    Andrew Jackson   XX07person_7
## 8     3.b 2008_Feb    Rushford (MN)  Martin Van Buren   XX08person_8
## 9     3.c 2008_Mar          Unknown  William Harrison   XX09person_9
```

In each of these cases, our objective may be to *separate* characters within the variable string. This can be accomplished using the `separate()` function which turns a single character column into multiple columns.

Complement to: `unite()`

```
Function:       separate(data, col, into, sep = " ", remove = TRUE, convert = FALS
E)
Same as:        data %>% separate(col, into, sep = " ", remove = TRUE, convert = F
ALSE)

Arguments:
        data:           data frame
        col:            column name representing current variable
        into:           names of variables representing new variables
        sep:            how to separate current variable (char, num, or symbol)
        remove:         if TRUE, remove input column from output data frame
        convert:        if TRUE will automatically convert values to logical, inte
ger, numeric, complex or
                        factor as appropriate
```

Example

We can go back to our **long_DF** dataframe we created above in which way may desire to clean up or separate the *Quarter* variable.

```
## Source: local data frame [6 x 4]
##
##   Group Year Quarter Revenue
## 1     1 2006   Qtr.1      15
## 2     1 2007   Qtr.1      12
## 3     1 2008   Qtr.1      22
## 4     1 2009   Qtr.1      10
## 5     2 2006   Qtr.1      12
## 6     2 2007   Qtr.1      16
```

By applying the `separate()` function we get the following:

```
separate_DF <- long_DF %>% separate(Quarter, c("Time_Interval", "Interval_ID"))
head(separate_DF, 10)
```

```
## Source: local data frame [10 x 5]
##
##    Group Year Time_Interval Interval_ID Revenue
## 1      1 2006           Qtr           1      15
## 2      1 2007           Qtr           1      12
## 3      1 2008           Qtr           1      22
## 4      1 2009           Qtr           1      10
## 5      2 2006           Qtr           1      12
## 6      2 2007           Qtr           1      16
## 7      2 2008           Qtr           1      13
## 8      2 2009           Qtr           1      23
## 9      3 2006           Qtr           1      11
## 10     3 2007           Qtr           1      13
```

```
These produce the same results:
        long_DF %>% separate(Quarter, c("Time_Interval", "Interval_ID"))
        long_DF %>% separate(Quarter, c("Time_Interval", "Interval_ID"), sep = "\\
.")
```

Go to top

unite( ) function:

Objective: Merging two variables into one

Description: There may be a time in which we would like to combine the values of two variables. The `unite()` function is a convenience function to paste together multiple variable values into one. In essence, it combines two variables of a single observation into one variable.

Complement to: `separate()`

```
Function:       unite(data, col, ..., sep = " ", remove = TRUE)
Same as:        data %>% unite(col, ..., sep = " ", remove = TRUE)

Arguments:
        data:           data frame
        col:            column name of new "merged" column
        ...:            names of columns to merge
        sep:            separator to use between merged values
        remove:         if TRUE, remove input column from output data frame
```

Example

Using the **separate_DF** dataframe we created above, we can re-unite the *Time_Interval* and *Interval_ID* variables we created and re-create the original *Quarter* variable we had in the **long_DF** dataframe.

```
unite_DF <- separate_DF %>% unite(Quarter, Time_Interval, Interval_ID, sep = ".")
head(unite_DF, 10)
```

```
## Source: local data frame [10 x 4]
##
##    Group Year Quarter Revenue
## 1      1 2006   Qtr.1      15
## 2      1 2007   Qtr.1      12
## 3      1 2008   Qtr.1      22
## 4      1 2009   Qtr.1      10
## 5      2 2006   Qtr.1      12
## 6      2 2007   Qtr.1      16
## 7      2 2008   Qtr.1      13
## 8      2 2009   Qtr.1      23
## 9      3 2006   Qtr.1      11
## 10     3 2007   Qtr.1      13
```

```
These produce the same results:
        separate_DF %>% unite(Quarter, Time_Interval, Interval_ID, sep = "_")
        separate_DF %>% unite(Quarter, Time_Interval, Interval_ID)

If no spearator is identified, "_" will automatically be used
```

Go to top

spread( ) function:

Objective: Reshaping long format to wide format

Description: There are times when we are required to turn long formatted data into wide formatted data. The `spread()` function spreads a key-value pair across multiple columns.

Complement to: `gather()`

```
Function:       spread(data, key, value, fill = NA, convert = FALSE)
Same as:        data %>% spread(key, value, fill = NA, convert = FALSE)

Arguments:
        data:           data frame
        key:            column values to convert to multiple columns
        value:          single column values to convert to multiple columns' value
s
        fill:           If there isn't a value for every combination of the other
variables and the key
                        column, this value will be substituted
        convert:        if TRUE will automatically convert values to logical, inte
ger, numeric, complex or
                        factor as appropriate
```

Example

```
wide_DF <- unite_DF %>% spread(Quarter, Revenue)
head(wide_DF, 24)
```

```
## Source: local data frame [12 x 6]
##
##    Group Year Qtr.1 Qtr.2 Qtr.3 Qtr.4
## 1      1 2006    15    16    19    17
## 2      1 2007    12    13    27    23
## 3      1 2008    22    22    24    20
## 4      1 2009    10    14    20    16
## 5      2 2006    12    13    25    18
## 6      2 2007    16    14    21    19
## 7      2 2008    13    11    29    15
## 8      2 2009    23    20    26    20
## 9      3 2006    11    12    22    16
## 10     3 2007    13    11    27    21
## 11     3 2008    17    12    23    19
## 12     3 2009    14     9    31    24
```

Go to top

dplyr Operations

There are seven fundamental functions of data transformation:

- `select()` selecting variables

- `filter()` provides basic filtering capabilities
- `group_by()` groups data by categorical levels
- `summarise()` summarise data by functions of choice
- `arrange()` ordering data
- `join()` joining separate dataframes
- `mutate()` create new variables

For these examples we'll use the following census data (http://www.census.gov/en.html) which includes the K-12 public school expenditures by state. This dataframe currently is 50x16 and includes expenditure data for 14 unique years.

Left half of data:

```
##   Division      State   X1980    X1990    X2000    X2001    X2002    X2003
## 1        6    Alabama 1146713  2275233  4176082  4354794  4444390  4657643
## 2        9     Alaska  377947   828051  1183499  1229036  1284854  1326226
## 3        8    Arizona  949753  2258660  4288739  4846105  5395814  5892227
## 4        7   Arkansas  666949  1404545  2380331  2505179  2822877  2923401
## 5        9 California 9172158 21485782 38129479 42908787 46265544 47983402
## 6        8   Colorado 1243049  2451833  4401010  4758173  5151003  5551506
```

Right half of data:

```
##      X2004    X2005    X2006    X2007    X2008    X2009    X2010    X2011
## 1  4812479  5164406  5699076  6245031  6832439  6683843  6670517  6592925
## 2  1354846  1442269  1529645  1634316  1918375  2007319  2084019  2201270
## 3  6071785  6579957  7130341  7815720  8403221  8726755  8482552  8340211
## 4  3109644  3546999  3808011  3997701  4156368  4240839  4459910  4578136
## 5 49215866 50918654 53436103 57352599 61570555 60080929 58248662 57526835
## 6  5666191  5994440  6368289  6579053  7338766  7187267  7429302  7409462
```

Go to top

select( ) function:

Objective: Reduce dataframe size to only desired variables for current task

Description: When working with a sizable dataframe, often we desire to only assess specific variables. The `select()` function allows you to select and/or rename variables.

```
Function:       select(data, ...)
Same as:        data %>% select(...)

Arguments:
        data:           data frame
        ...:            call variables by name or by function

Special functions:
        starts_with(x, ignore.case = TRUE): names starts with x
        ends_with(x, ignore.case = TRUE):   names ends in x
        contains(x, ignore.case = TRUE):    selects all variables whose name conta
ins x
        matches(x, ignore.case = TRUE):     selects all variables whose name match
es the regular expression x
```

Example Let's say our goal is to only assess the 5 most recent years worth of expenditure data. Applying the `select()` function we can *select* only the variables of concern.

```
sub.exp <- expenditures %>% select(Division, State, X2007:X2011)
head(sub.exp)  # for brevity only display first 6 rows
```

```
##   Division      State    X2007    X2008    X2009    X2010    X2011
## 1        6    Alabama  6245031  6832439  6683843  6670517  6592925
## 2        9     Alaska  1634316  1918375  2007319  2084019  2201270
## 3        8    Arizona  7815720  8403221  8726755  8482552  8340211
## 4        7   Arkansas  3997701  4156368  4240839  4459910  4578136
## 5        9 California 57352599 61570555 60080929 58248662 57526835
## 6        8   Colorado  6579053  7338766  7187267  7429302  7409462
```

We can also apply some of the special functions within `select()`. For instance we can select all variables that start with 'X':

```
head(expenditures %>% select(starts_with("X")))
```

```
##     X1980    X1990    X2000    X2001    X2002    X2003    X2004    X2005
## 1 1146713  2275233  4176082  4354794  4444390  4657643  4812479  5164406
## 2  377947   828051  1183499  1229036  1284854  1326226  1354846  1442269
## 3  949753  2258660  4288739  4846105  5395814  5892227  6071785  6579957
## 4  666949  1404545  2380331  2505179  2822877  2923401  3109644  3546999
## 5 9172158 21485782 38129479 42908787 46265544 47983402 49215866 50918654
## 6 1243049  2451833  4401010  4758173  5151003  5551506  5666191  5994440
##      X2006    X2007    X2008    X2009    X2010    X2011
## 1  5699076  6245031  6832439  6683843  6670517  6592925
## 2  1529645  1634316  1918375  2007319  2084019  2201270
## 3  7130341  7815720  8403221  8726755  8482552  8340211
## 4  3808011  3997701  4156368  4240839  4459910  4578136
## 5 53436103 57352599 61570555 60080929 58248662 57526835
## 6  6368289  6579053  7338766  7187267  7429302  7409462
```

```
You can also de-select variables by using "-" prior to name or function.  The foll
owing produces the inverse of functions above
        expenditures %>% select(-X1980:-X2006)
        expenditures %>% select(-starts_with("X"))
```

Go to top

filter( ) function:

Objective: Reduce rows/observations with matching conditions

Description: Filtering data is a common task to identify/select observations in which a particular variable matches a specific value/condition. The `filter()` function provides this capability.

```
Function:       filter(data, ...)
Same as:        data %>% filter(...)

Arguments:
        data:           data frame
        ...:            conditions to be met
```

Examples

Continuing with our **sub.exp** dataframe which includes only the recent 5 years worth of expenditures, we can filter by *Division*:

```
sub.exp %>% filter(Division == 3)
```

```
##   Division     State    X2007    X2008    X2009    X2010    X2011
## 1        3  Illinois 20326591 21874484 23495271 24695773 24554467
## 2        3   Indiana  9497077  9281709  9680895  9921243  9687949
## 3        3  Michigan 17013259 17053521 17217584 17227515 16786444
## 4        3      Ohio 18251361 18892374 19387318 19801670 19988921
## 5        3 Wisconsin  9029660  9366134  9696228  9966244 10333016
```

We can apply multiple logic rules in the `filter()` function such as:

```
<   Less than                    !=      Not equal to
>   Greater than                 %in%    Group membership
==  Equal to                     is.na   is NA
<=  Less than or equal to        !is.na  is not NA
>=  Greater than or equal to     &,|,!   Boolean operators
```

For instance, we can filter for Division 3 and expenditures in 2011 that were greater than $10B. This results in Indiana, which is in Division 3, being excluded since its expenditures were < $10B *(FYI - the raw census data are reported in units of $1,000)*.

```
sub.exp %>% filter(Division == 3, X2011 > 10000000)  # Raw census data are in unit
s of $1,000
```

```
##   Division     State    X2007    X2008    X2009    X2010    X2011
## 1        3  Illinois 20326591 21874484 23495271 24695773 24554467
## 2        3  Michigan 17013259 17053521 17217584 17227515 16786444
## 3        3      Ohio 18251361 18892374 19387318 19801670 19988921
## 4        3 Wisconsin  9029660  9366134  9696228  9966244 10333016
```

Go to top

group_by( ) function:

Objective: Group data by categorical variables

Description: Often, observations are nested within groups or categories and our goals is to perform statistical analysis both at the observation level and also at the group level. The `group_by()` function allows us to create these categorical groupings.

```
Function:       group_by(data, ...)
Same as:        data %>% group_by(...)

Arguments:
        data:           data frame
        ...:            variables to group_by

*Use ungroup(x) to remove groups
```

Example The `group_by()` function is a *silent* function in which no observable manipulation of the data is performed as a result of applying the function. Rather, the only change you'll notice is, if you print the dataframe you will notice underneath the *Source* information and prior to the actual dataframe, an indicator of what variable the data is grouped by will be provided. The real magic of the `group_by()` function comes when we perform summary statistics which we will cover shortly.

```
group.exp <- sub.exp %>% group_by(Division)

head(group.exp)
```

```
## Source: local data frame [6 x 7]
## Groups: Division
##
##   Division      State    X2007    X2008    X2009    X2010    X2011
## 1        6    Alabama  6245031  6832439  6683843  6670517  6592925
## 2        9     Alaska  1634316  1918375  2007319  2084019  2201270
## 3        8    Arizona  7815720  8403221  8726755  8482552  8340211
## 4        7   Arkansas  3997701  4156368  4240839  4459910  4578136
## 5        9 California 57352599 61570555 60080929 58248662 57526835
## 6        8   Colorado  6579053  7338766  7187267  7429302  7409462
```

Go to top

summarise( ) function:

Objective: Perform summary statistics on variables

Description: Obviously the goal of all this data *wrangling* is to be able to perform statistical analysis on our data. The `summarise()` function allows us to perform the majority of the initial summary statistics when performing exploratory data analysis.

```
Function:       summarise(data, ...)
Same as:        data %>% summarise(...)

Arguments:
        data:           data frame
        ...:            Name-value pairs of summary functions like min(), mean(),
max() etc.

*Developer is from New Zealand...can use "summarise(x)" or "summarize(x)"
```

Examples

Lets get the mean expenditure value across all states in 2011

```
sub.exp %>% summarise(Mean_2011 = mean(X2011))
```

```
##   Mean_2011
## 1  10513678
```

Not too bad, lets get some more summary stats

```
sub.exp %>% summarise(Min = min(X2011, na.rm=TRUE),
                     Median = median(X2011, na.rm=TRUE),
                     Mean = mean(X2011, na.rm=TRUE),
                     Var = var(X2011, na.rm=TRUE),
                     SD = sd(X2011, na.rm=TRUE),
                     Max = max(X2011, na.rm=TRUE),
                     N = n())
```

```
##       Min  Median     Mean         Var       SD      Max  N
## 1 1049772 6527404 10513678 1.48619e+14 12190938 57526835 50
```

This information is useful, but being able to compare summary statistics at multiple levels is when you really start to gather some insights. This is where the `group_by()` function comes in. First, let's group by *Division* and see how the different regions compared in by 2010 and 2011.

```
sub.exp %>%
        group_by(Division)%>%
        summarise(Mean_2010 = mean(X2010, na.rm=TRUE),
                  Mean_2011 = mean(X2011, na.rm=TRUE))
```

```
## Source: local data frame [9 x 3]
##
##   Division Mean_2010 Mean_2011
## 1        1   5121003   5222277
## 2        2  32415457  32877923
## 3        3  16322489  16270159
## 4        4   4672332   4672687
## 5        5  10975194  11023526
## 6        6   6161967   6267490
## 7        7  14916843  15000139
## 8        8   3894003   3882159
## 9        9  15540681  15468173
```

Now we're starting to see some differences pop out. How about we compare states within a Division? We can start to apply multiple functions we've learned so far to get the 5 year average for each state within Division 3.

```
sub.exp %>%
        gather(Year, Expenditure, X2007:X2011) %>%   # this turns our wide data to
a long format
        filter(Division == 3) %>%                    # we only want to compare sta
tes within Division 3
        group_by(State) %>%                          # we want to summarize data a
t the state level
        summarise(Mean = mean(Expenditure),
                  SD = sd(Expenditure))
```

```
## Source: local data frame [5 x 3]
##
##       State     Mean        SD
## 1  Illinois 22989317 1867527.7
## 2   Indiana  9613775  238971.6
## 3  Michigan 17059665  180245.0
## 4      Ohio 19264329  705930.2
## 5 Wisconsin  9678256  507461.2
```

Go to top

arrange( ) function:

Objective: Order variable values

Description: Often, we desire to view observations in rank order for a particular variable(s). The `arrange()` function allows us to order data by variables in accending or descending order.

```
Function:       arrange(data, ...)
Same as:        data %>% arrange(...)

Arguments:
        data:           data frame
        ...:            Variable(s) to order

*use desc(x) to sort variable in descending order
```

Examples

For instance, in the summarise example we compared the the mean expenditures for each division. We could apply the `arrange()` function at the end to order the divisions from lowest to highest expenditure for 2011. This makes it easier to see the significant differences between Divisions 8,4,1 & 6 as compared to Divisions 5,7,9,3 & 2.

```
sub.exp %>%
        group_by(Division)%>%
        summarise(Mean_2010 = mean(X2010, na.rm=TRUE),
                  Mean_2011 = mean(X2011, na.rm=TRUE)) %>%
        arrange(Mean_2011)
```

```
## Source: local data frame [9 x 3]
##
##   Division Mean_2010 Mean_2011
## 1        8   3894003   3882159
## 2        4   4672332   4672687
## 3        1   5121003   5222277
## 4        6   6161967   6267490
## 5        5  10975194  11023526
## 6        7  14916843  15000139
## 7        9  15540681  15468173
## 8        3  16322489  16270159
## 9        2  32415457  32877923
```

We can also apply an *descending* argument to rank-order from highest to lowest. The following shows the same data but in descending order by applying `desc()` within the `arrange()` function.

```
sub.exp %>%
        group_by(Division)%>%
        summarise(Mean_2010 = mean(X2010, na.rm=TRUE),
                  Mean_2011 = mean(X2011, na.rm=TRUE)) %>%
        arrange(desc(Mean_2011))
```

```
## Source: local data frame [9 x 3]
##
##   Division Mean_2010 Mean_2011
## 1        2  32415457  32877923
## 2        3  16322489  16270159
## 3        9  15540681  15468173
## 4        7  14916843  15000139
## 5        5  10975194  11023526
## 6        6   6161967   6267490
## 7        1   5121003   5222277
## 8        4   4672332   4672687
## 9        8   3894003   3882159
```

Go to top

join( ) functions:

Objective: Join two datasets together

Description: Often we have separate dataframes that can have common and differing variables for similar observations and we wish to *join* these dataframes together. The multiple `xxx_join()` functions provide multiple ways to join dataframes.

```
Description:    Join two datasets

Function:
                inner_join(x, y, by = NULL)
                left_join(x, y, by = NULL)
                semi_join(x, y, by = NULL)
                anti_join(x, y, by = NULL)

Arguments:
        x,y:            data frames to join
        by:             a character vector of variables to join by. If NULL, the de
fault, join will do a natural join, using all
                         variables with common names across the two tables.
```

Example

Our public education expenditure data represents then-year dollars. To make any accurate assessments of longitudinal trends and comparison we need to adjust for inflation. I have the following dataframe which provides inflation adjustment factors for base-year 2012 dollars *(obviously I should use 2014 values but I had these easily accessable and it only serves for illustrative purposes)*.

```
##    Year  Annual Inflation
## 28 2007 207.342 0.9030811
## 29 2008 215.303 0.9377553
## 30 2009 214.537 0.9344190
## 31 2010 218.056 0.9497461
## 32 2011 224.939 0.9797251
## 33 2012 229.594 1.0000000
```

To join to my expenditure data I obviously need to get my expenditure data in the proper form that allows my to join these two dataframes. I can apply the following functions to accomplish this:

```
long.exp <- sub.exp %>%
        gather(Year, Expenditure, X2007:X2011) %>%         # turn to long format
        separate(Year, into=c("x", "Year"), sep="X") %>%   # separate "X" from yea
r value
        select(-x)                                         # remove "x" column

long.exp$Year <- as.numeric(long.exp$ Year)  # convert from character to numeric

head(long.exp)
```

```
##   Division      State Year Expenditure
## 1        6    Alabama 2007     6245031
## 2        9     Alaska 2007     1634316
## 3        8    Arizona 2007     7815720
## 4        7   Arkansas 2007     3997701
## 5        9 California 2007    57352599
## 6        8   Colorado 2007     6579053
```

I can now apply the `left_join()` function to join the inflation data to the expenditure data. This aligns the data in both dataframes by the *Year* variable and then joins the remaining inflation data to the expenditure dataframe as new variables.

```
join.exp <- long.exp %>% left_join(inflation)

head(join.exp)
```

```
##   Year Division      State Expenditure  Annual Inflation
## 1 2007        6    Alabama     6245031 207.342 0.9030811
## 2 2007        9     Alaska     1634316 207.342 0.9030811
## 3 2007        8    Arizona     7815720 207.342 0.9030811
## 4 2007        7   Arkansas     3997701 207.342 0.9030811
## 5 2007        9 California    57352599 207.342 0.9030811
## 6 2007        8   Colorado     6579053 207.342 0.9030811
```

To illustrate the other joining methods we can use these two simple dateframes:

Dataframe “x”:

```
##     name instrument
## 1   John     guitar
## 2   Paul       bass
## 3 George     guitar
## 4  Ringo      drums
## 5 Stuart       bass
## 6   Pete      drums
```

Dataframe “y”:

```
##     name  band
## 1   John  TRUE
## 2   Paul  TRUE
## 3 George  TRUE
## 4  Ringo  TRUE
## 5  Brian FALSE
```

`inner_join()` : Include only rows in both x and y that have a matching value

```
inner_join(x,y)
```

```
##     name instrument band
## 1   John     guitar TRUE
## 2   Paul       bass TRUE
## 3 George     guitar TRUE
## 4  Ringo      drums TRUE
```

`left_join()` : Include all of x, and matching rows of y

```
left_join(x,y)
```

```
##     name instrument band
## 1   John     guitar TRUE
## 2   Paul       bass TRUE
## 3 George     guitar TRUE
## 4  Ringo      drums TRUE
## 5 Stuart       bass <NA>
## 6   Pete      drums <NA>
```

`semi_join()` : Include rows of x that match y but only keep the columns from x

```
semi_join(x,y)
```

```
##     name instrument
## 1   John     guitar
## 2   Paul       bass
## 3 George     guitar
## 4  Ringo      drums
```

`anti_join()` : Opposite of semi_join

```
anti_join(x,y)
```

```
##     name instrument
## 1   Pete      drums
## 2 Stuart       bass
```

Go to top

mutate( ) function:

Objective: Creates new variables

Description: Often we want to create a new variable that is a function of the current variables in our dataframe or even just add a new variable. The `mutate()` function allows us to add new variables while preserving the existing variables.

```
Function:
                mutate(data, ...)
Same as:        data %>% mutate(...)

Arguments:
        data:           data frame
        ...:            Expression(s)
```

Examples

If we go back to our previous **join.exp** dataframe, remember that we joined inflation rates to our non-inflation adjusted expenditures for public schools. The dataframe looks like:

```
##   Year Division      State Expenditure  Annual Inflation
## 1 2007        6    Alabama     6245031 207.342 0.9030811
## 2 2007        9     Alaska     1634316 207.342 0.9030811
## 3 2007        8    Arizona     7815720 207.342 0.9030811
## 4 2007        7   Arkansas     3997701 207.342 0.9030811
## 5 2007        9 California    57352599 207.342 0.9030811
## 6 2007        8   Colorado     6579053 207.342 0.9030811
```

If we wanted to adjust our annual expenditures for inflation we can use `mutate()` to create a new inflation adjusted cost variable which we'll name *Adj_Exp*:

```
inflation_adj <- join.exp %>% mutate(Adj_Exp = Expenditure/Inflation)

head(inflation_adj)
```

```
##   Year Division      State Expenditure  Annual Inflation  Adj_Exp
## 1 2007        6    Alabama     6245031 207.342 0.9030811  6915249
## 2 2007        9     Alaska     1634316 207.342 0.9030811  1809711
## 3 2007        8    Arizona     7815720 207.342 0.9030811  8654505
## 4 2007        7   Arkansas     3997701 207.342 0.9030811  4426735
## 5 2007        9 California    57352599 207.342 0.9030811 63507696
## 6 2007        8   Colorado     6579053 207.342 0.9030811  7285119
```

Lets say we wanted to create a variable that rank-orders state-level expenditures (inflation adjusted) for the year 2010 from the highest level of expenditures to the lowest.

```
rank_exp <- inflation_adj %>%
        filter(Year == 2010) %>%
        arrange(desc(Adj_Exp)) %>%
        mutate(Rank = 1:length(Adj_Exp))

head(rank_exp)
```

```
##   Year Division      State Expenditure  Annual Inflation  Adj_Exp Rank
## 1 2010        9 California    58248662 218.056 0.9497461 61330774    1
## 2 2010        2   New York    50251461 218.056 0.9497461 52910417    2
## 3 2010        7      Texas    42621886 218.056 0.9497461 44877138    3
## 4 2010        3   Illinois    24695773 218.056 0.9497461 26002501    4
## 5 2010        2 New Jersey    24261392 218.056 0.9497461 25545135    5
## 6 2010        5    Florida    23349314 218.056 0.9497461 24584797    6
```

If you wanted to assess the percent change in cost for a particular state you can use the `lag()` function within the `mutate()` function:

```
inflation_adj %>%
        filter(State == "Ohio") %>%
        mutate(Perc_Chg = (Adj_Exp-lag(Adj_Exp))/lag(Adj_Exp))
```

```
##   Year Division State Expenditure  Annual Inflation  Adj_Exp     Perc_Chg
## 1 2007        3  Ohio    18251361 207.342 0.9030811 20210102           NA
## 2 2008        3  Ohio    18892374 215.303 0.9377553 20146378 -0.003153057
## 3 2009        3  Ohio    19387318 214.537 0.9344190 20747992  0.029862103
## 4 2010        3  Ohio    19801670 218.056 0.9497461 20849436  0.004889357
## 5 2011        3  Ohio    19988921 224.939 0.9797251 20402582 -0.021432441
```

You could also look at what percent of all US expenditures each state made up in 2011. In this case we use `mutate()` to take each state's inflation adjusted expenditure and divide by the sum of the entire inflation adjusted expenditure column. We also apply a second function within `mutate()` that provides the cummalative percent in rank-order. This shows that in 2011, the top 8 states with the highest expenditures represented over 50% of the total U.S. expenditures in K-12 public schools. *(I remove the non-inflation adjusted Expenditure, Annual & Inflation columns so that the columns don't wrap on the screen view)*

```
perc.of.whole <- inflation_adj %>%
        filter(Year == 2011) %>%
        arrange(desc(Adj_Exp)) %>%
        mutate(Perc_of_Total = Adj_Exp/sum(Adj_Exp),
               Cum_Perc = cumsum(Perc_of_Total)) %>%
        select(-Expenditure, -Annual, -Inflation)

head(perc.of.whole, 8)
```

```
##   Year Division        State  Adj_Exp Perc_of_Total  Cum_Perc
## 1 2011        9   California 58717324    0.10943237 0.1094324
## 2 2011        2     New York 52575244    0.09798528 0.2074177
## 3 2011        7        Texas 43751346    0.08154005 0.2889577
## 4 2011        3     Illinois 25062609    0.04670957 0.3356673
## 5 2011        5      Florida 24364070    0.04540769 0.3810750
## 6 2011        2   New Jersey 24128484    0.04496862 0.4260436
## 7 2011        2 Pennsylvania 23971218    0.04467552 0.4707191
## 8 2011        3         Ohio 20402582    0.03802460 0.5087437
```

Go to top

Additional Resources

This tutorial simply touches on the basics that these two packages can do. There are several other resources you can check out to learn more. In addition, much of what I have learned and, therefore, much of the content in this tutorial is simply a modified regurgitation of the wonderful resources provided by R Studio (http://www.rstudio.com/), Hadley Wickham (https://twitter.com/hadleywickham), and Garrett Grolemund (https://twitter.com/StatGarrett).

- R Studio's Data wrangling with R and RStudio webinar (http://www.rstudio.com/resources/webinars/)
- R Studio's Data wrangling GitHub repository (https://github.com/rstudio/webinars/blob/master/2015-01/wrangling-webinar.pdf)
- R Studio's Data wrangling cheat sheet (http://www.rstudio.com/resources/cheatsheets/)
- Hadley Wickham's dplyr tutorial at useR! 2014, Part 1 (http://www.r-bloggers.com/hadley-wickhams-dplyr-tutorial-at-user-2014-part-1/)
- Hadley Wickham's dplyr tutorial at useR! 2014, Part 2 (http://www.r-bloggers.com/hadley-wickhams-dplyr-tutorial-at-user-2014-part-2/)
- Hadley Wickham's paper on Tidy Data (http://vita.had.co.nz/papers/tidy-data.html)

Go to top

Special thanks to Tom Filloon and Jason Freels for providing constructive comments while developing this tutorial.