Shaping & Transforming Your Data with R

*Up to 80% of data analysis is spent on the process of cleaning and preparing data.* - cf. [Wickham, 2014](https://www.jstatsoft.org/article/view/v059i10) and [Dasu and Johnson, 2003](http://onlinelibrary.wiley.com/doi/10.1002/0471448354.ch4/summary)

A tremendous amount of time is spent on fundamental preprocessing tasks to get your data into the right form in order to feed it into the visualization and modeling stages. This typically requires a large amount of reshaping and transformation of your data. 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. The [RStudio team](https://www.rstudio.com/home/) has been driving a lot of new packages to collate data management tasks and better integrate them with other analysis activities. As a result, a lot of data processing tasks are becoming packaged in more cohesive and consistent ways which leads to more efficient code and easier to read syntax. This section covers two of these packages: tidyr and dplyr.

In this section, I start by providing a fundamental understanding of tidy data followed by demonstrating [how to to use tidyr](#tidyr) to turn wide data to long, long data to wide, splitting and combining variables, along with illustrating some lesser-known functions. Subsequently, I provide an [introduction to the dplyr package](#dplyr) by covering seven primary functions dplyr provides for simplified data transformation and manipulation. This includes tasks such as filtering, summarizing, ordering, joining, and much more. Understanding and using these two packages will help to significantly reduce the time you spend on the data wrangling process.

# Reshaping Your Data with tidyr

*"Cannot emphasize enough how much time you save by putting analysis efforts into tidying data first."* - Hilary Parker

[Jenny Bryan](https://twitter.com/JennyBryan) stated that "classroom data are like teddy bears and real data are like a grizzley bear with salmon blood dripping out its mouth." In essence, she was getting to the point that often when we learn how to perform a modeling approach in the classroom, the data used is provided in a format that appropriately feeds into the modeling tool of choice. In reality, datasets are messy and "every messy dataset is messy in its own way."[[1]](#footnote-26) The concept of "tidy data" was established by Hadley Wickham and represents "standardized way to link the structure of a dataset (its physical layout) with its semantics (its meaning)."[[2]](#footnote-28) The objective should always to be to get a dataset into a tidy form which consists of:

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

To create tidy data you need to be able to reshape your data; preferably via efficient and simple code. To help with this process Hadley created the [tidyr](https://cran.r-project.org/web/packages/tidyr/index.html) package. This chapter covers the basics of tidyr to help you reshape your data as necessary. I demonstrate how to [turn wide data to long](#gather), [long data to wide](#spread), [splitting](#separate) and [combining](#unite) variables, and finally I will cover some [lesser known functions](#tidyr_addtl) in tidyr that are useful. Note that throughout I use the %>% operator we covered in the [last chapter](#pipe). Although not required, the tidyr package has the %>% operator baked in to its functionality, which allows you to [sequence multiple tidy functions together](#tidyr_seq).

## Making wide data long

There are times when our data is considered "wide" or "unstacked" and a common attribute/variable 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.

For example, let's say we have the given data frame.

library(dplyr) # I'm using dplyr just to create the data frame with tbl\_df()  
  
wide <- tbl\_df(read.table(header = TRUE, text = "  
 Group Year Qtr.1 Qtr.2 Qtr.3 Qtr.4  
 1 2006 15 16 19 17  
 1 2007 12 13 27 23  
 1 2008 22 22 24 20  
 1 2009 10 14 20 16  
 2 2006 12 13 25 18  
 2 2007 16 14 21 19  
 2 2008 13 11 29 15  
 2 2009 23 20 26 20  
 3 2006 11 12 22 16  
 3 2007 13 11 27 21  
 3 2008 17 12 23 19  
 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.

library(tidyr)  
  
long <- wide %>% gather(Quarter, Revenue, Qtr.1:Qtr.4)  
  
# note, for brevity, I only show the first 15 observations  
head(long, 15)   
## Source: local data frame [15 x 4]  
##   
## Group Year Quarter Revenue  
## (int) (int) (fctr) (int)  
## 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  
## 11 3 2008 Qtr.1 17  
## 12 3 2009 Qtr.1 14  
## 13 1 2006 Qtr.2 16  
## 14 1 2007 Qtr.2 13  
## 15 1 2008 Qtr.2 22

It's important to note that there is flexibility in how you specify the columns you would like to gather. These all produce the same results:

wide %>% gather(Quarter, Revenue, Qtr.1:Qtr.4)  
wide %>% gather(Quarter, Revenue, -Group, -Year)  
wide %>% gather(Quarter, Revenue, 3:6)  
wide %>% gather(Quarter, Revenue, Qtr.1, Qtr.2, Qtr.3, Qtr.4)

## Making long data wide

There are also times when we are required to turn long formatted data into wide formatted data. As a complement to gather(), the spread() function spreads a key-value pair across multiple columns. So now let's take our long data frame from above and and turn the Quarter variable into column headings and spread the Revenue values across the quarters they are related to.

back2wide <- long %>% spread(Quarter, Revenue)  
  
back2wide  
## Source: local data frame [12 x 6]  
##   
## Group Year Qtr.1 Qtr.2 Qtr.3 Qtr.4  
## (int) (int) (int) (int) (int) (int)  
## 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

## Splitting a single column into multiple columns

Many times a single column variable will capture multiple variables, or even parts of a variable you just don't care about. This is exemplified in the following messy\_df data frame. Here, the Grp\_Ind variable combines an individual variable (a, b, c) with the group variable (1, 2, 3), the Yr\_Mo variable combines a year variable with a month variable, etc. In each case there may be a purpose for separating parts of these columns into *separate* variables.

messy\_df  
## Grp\_Ind Yr\_Mo City\_State Extra\_variable  
## 1 1.a 2006\_Jan Dayton (OH) XX01person\_1  
## 2 1.b 2006\_Feb Grand Forks (ND) XX02person\_2  
## 3 1.c 2006\_Mar Fargo (ND) XX03person\_3  
## 4 2.a 2007\_Jan Rochester (MN) XX04person\_4

This can be accomplished using the separate() function which turns a single character column into multiple columns. Additional arguments provide some flexibility with separating columns.

# separate Grp\_Ind column into two variables named "Grp" & "Ind"  
messy\_df %>% separate(col = Grp\_Ind, into = c("Grp", "Ind"))  
## Grp Ind Yr\_Mo City\_State Extra\_variable  
## 1 1 a 2006\_Jan Dayton (OH) XX01person\_1  
## 2 1 b 2006\_Feb Grand Forks (ND) XX02person\_2  
## 3 1 c 2006\_Mar Fargo (ND) XX03person\_3  
## 4 2 a 2007\_Jan Rochester (MN) XX04person\_4  
  
# default separater is any non alpha-numeric character but you can   
# specify the specific character to separate at  
messy\_df %>% separate(col = Extra\_variable, into = c("X", "Y"), sep = "\_")  
## Grp\_Ind Yr\_Mo City\_State X Y  
## 1 1.a 2006\_Jan Dayton (OH) XX01person 1  
## 2 1.b 2006\_Feb Grand Forks (ND) XX02person 2  
## 3 1.c 2006\_Mar Fargo (ND) XX03person 3  
## 4 2.a 2007\_Jan Rochester (MN) XX04person 4  
  
# you can keep the original column that you are separating  
messy\_df %>% separate(col = Grp\_Ind, into = c("Grp", "Ind"), remove = FALSE)  
## Grp\_Ind Grp Ind Yr\_Mo City\_State Extra\_variable  
## 1 1.a 1 a 2006\_Jan Dayton (OH) XX01person\_1  
## 2 1.b 1 b 2006\_Feb Grand Forks (ND) XX02person\_2  
## 3 1.c 1 c 2006\_Mar Fargo (ND) XX03person\_3  
## 4 2.a 2 a 2007\_Jan Rochester (MN) XX04person\_4

## Combining multiple columns into a single column

Similarly, there are times when we would like to combine the values of two variables. As a compliment to separate(), the unite() function is a convenient function to paste together multiple variable values into one. Consider the following data frame that has separate date variables. To perform time series analysis or for visualizations we may desire to have a single date column.

expenses <- tbl\_df(read.table(header = TRUE, text = "  
 Year Month Day Expense  
 2015 01 01 500  
 2015 02 05 90  
 2015 02 22 250  
 2015 03 10 325  
"))

To perform time series analysis or for visualizations we may desire to have a single date column. We can accomplish this by *uniting* these columns into one variable with unite().

# default separator when uniting is "\_"  
expenses %>% unite(col = "Date", c(Year, Month, Day))  
## Source: local data frame [4 x 2]  
##   
## Date Expense  
## (chr) (int)  
## 1 2015\_1\_1 500  
## 2 2015\_2\_5 90  
## 3 2015\_2\_22 250  
## 4 2015\_3\_10 325  
  
# specify sep argument to change separater  
expenses %>% unite(col = "Date", c(Year, Month, Day), sep = "-")  
## Source: local data frame [4 x 2]  
##   
## Date Expense  
## (chr) (int)  
## 1 2015-1-1 500  
## 2 2015-2-5 90  
## 3 2015-2-22 250  
## 4 2015-3-10 325

## Additional tidyr functions

The previous four functions (gather, spread, separate and unite) are the primary functions you will find yourself using on a continuous basis; however, there are some handy functions that are lesser known with the tidyr package.

expenses <- tbl\_df(read.table(header = TRUE, text = "  
 Dept Year Month Day Cost  
 A 2015 01 01 $500.00  
 NA NA 02 05 $90.00  
 NA NA 02 22 $1,250.45  
 NA NA 03 NA $325.10  
 B NA 01 02 $260.00  
 NA NA 02 05 $90.00  
", stringsAsFactors = FALSE))

Often Excel reports will not repeat certain variables. When we read these reports in, the empty cells are typically filled in with NA such as in the Dept and Year columns of our expense data frame. We can fill these values in with the previous entry using fill().

expenses %>% fill(Dept, Year)  
## Source: local data frame [6 x 5]  
##   
## Dept Year Month Day Cost  
## (chr) (int) (int) (int) (chr)  
## 1 A 2015 1 1 $500.00  
## 2 A 2015 2 5 $90.00  
## 3 A 2015 2 22 $1,250.45  
## 4 A 2015 3 NA $325.10  
## 5 B 2015 1 2 $260.00  
## 6 B 2015 2 5 $90.00

Also, sometimes accounting values in Excel spreadsheet get read in as a character value, which is the case for the Cost variable. We may wish to extract only the numeric part of this regular expression, which can be done with extract\_numeric(). Note that extract\_numeric works on a single variable so when you pipe the expense data frame into the function you need to use %$% operator as discussed in the [last chapter](#pipe).

library(magrittr)  
  
expenses %$% extract\_numeric(Cost)  
## [1] 500.00 90.00 1250.45 325.10 260.00 90.00  
  
# you can use this to convert and save the Cost column to a  
# numeric variable  
expenses$Cost <- expenses %$% extract\_numeric(Cost)  
  
expenses  
## Source: local data frame [6 x 5]  
##   
## Dept Year Month Day Cost  
## (chr) (int) (int) (int) (dbl)  
## 1 A 2015 1 1 500.00  
## 2 NA NA 2 5 90.00  
## 3 NA NA 2 22 1250.45  
## 4 NA NA 3 NA 325.10  
## 5 B NA 1 2 260.00  
## 6 NA NA 2 5 90.00

You can also easily replace missing (or NA) values with a specified value:

library(magrittr)  
  
# replace the missing Day value  
expenses %>% replace\_na(replace = list(Day = "unknown"))  
## Source: local data frame [6 x 5]  
##   
## Dept Year Month Day Cost  
## (chr) (int) (int) (chr) (dbl)  
## 1 A 2015 1 1 500.00  
## 2 NA NA 2 5 90.00  
## 3 NA NA 2 22 1250.45  
## 4 NA NA 3 unknown 325.10  
## 5 B NA 1 2 260.00  
## 6 NA NA 2 5 90.00  
  
# replace both the missing Day and Year values  
expenses %>% replace\_na(replace = list(Year = 2015, Day = "unknown"))  
## Source: local data frame [6 x 5]  
##   
## Dept Year Month Day Cost  
## (chr) (dbl) (int) (chr) (dbl)  
## 1 A 2015 1 1 500.00  
## 2 NA 2015 2 5 90.00  
## 3 NA 2015 2 22 1250.45  
## 4 NA 2015 3 unknown 325.10  
## 5 B 2015 1 2 260.00  
## 6 NA 2015 2 5 90.00

## Sequencing your tidyr operations

Since the %>% operator is embedded in tidyr, we can string multiple operations together to efficiently tidy data *and* make the process easy to read and follow. To illustrate, let's use the following data, which has multiple *messy* attributes.

a\_mess <- tbl\_df(read.table(header = TRUE, text = "  
 Dep\_Unt Year Q1 Q2 Q3 Q4  
 A.1 2006 15 NA 19 17  
 B.1 NA 12 13 27 23  
 A.2 NA 22 22 24 20  
 B.2 NA 12 13 25 18  
 A.1 2007 16 14 21 19  
 B.2 NA 13 11 16 15  
 A.2 NA 23 20 26 20  
 B.2 NA 11 12 22 16  
"))

In this case, a tidy dataset should result in columns of Dept, Unit, Year, Quarter, and Cost. Furthermore, we want to fill in the year column where NAs currently exist. And we'll assume that we know the missing value that exists in the Q2 column, and we'd like to update it.

a\_mess %>%  
 fill(Year) %>%  
 gather(Quarter, Cost, Q1:Q4) %>%  
 separate(Dep\_Unt, into = c("Dept", "Unit")) %>%  
 replace\_na(replace = list(Cost = 17))  
## Source: local data frame [32 x 5]  
##   
## Dept Unit Year Quarter Cost  
## (chr) (chr) (int) (fctr) (dbl)  
## 1 A 1 2006 Q1 15  
## 2 B 1 2006 Q1 12  
## 3 A 2 2006 Q1 22  
## 4 B 2 2006 Q1 12  
## 5 A 1 2007 Q1 16  
## 6 B 2 2007 Q1 13  
## 7 A 2 2007 Q1 23  
## 8 B 2 2007 Q1 11  
## 9 A 1 2006 Q2 17  
## 10 B 1 2006 Q2 13  
## .. ... ... ... ... ...

## Additional resources

This chapter covers most, but not all, of what tidyr provides. There are several other resources you can check out to learn more.

* [Data wrangling presentation](http://bradleyboehmke.github.io/2015/10/data-wrangling-presentation.html) I gave at Miami University
* Hadley Wickham's [tidy data paper](http://jstatsoft.org/v59/i10)
* [tidyr reference manual](https://cran.r-project.org/web/packages/tidyr/tidyr.pdf)
* 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/)

# Transforming Your Data with dplyr

Transforming your data is a basic part of data wrangling. This can include filtering, summarizing, and ordering your data by different means. This also includes combining disperate data sets, creating new variables, and many other manipulation tasks. Although many fundamental data transformation and manipulation functions exist in R, historically they have been a bit convoluted and lacked a consistent and cohesive code structure. Consequently, Hadley Wickham developed the very popular dplyr package to make these data processing tasks more efficient along with a syntax that is consistent and easier to remember and read.

dplyr's roots originate in the popular [plyr](https://cran.r-project.org/web/packages/plyr/index.html) package, also produced by Hadley Wickham. plyr covers data transformation and manipulation for a range of data structures (data frames, lists, arrays) whereas dplyr is focused on transformation and manipulation of data frames. And since the bulk of data analysis leverages data frames I am going to focus on dplyr. Even so, dplyr offers far more functionality than I can cover in one chapter. Consequently, I'm going to cover the seven primary functions dplyr provides for data transformation and manipulation. Throughout, I also mention additional, useful functions that can be integrated with these functions. The full list of capabilities can be found in the [dplyr reference manual](https://cran.r-project.org/web/packages/dplyr/dplyr.pdf); I highly recommend going through it as there are many great functions provided by dplyr that I will not cover here. Also, similar to tidyr, dplyr has the %>% operator baked in to its functionality.

For most of 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 (50 states and has data through year 2011). Here I only show you a subset of the 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  
## 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

## Selecting variables of interest

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. 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' (?select to see the available functions):

expenditures %>%   
 select(starts\_with("X")) %>%  
 head  
## 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 following produces the inverse of functions above:

expenditures %>% select(-X1980:-X2006)  
expenditures %>% select(-starts\_with("X"))

And for convenience, you can rename selected variables with two options:

# select and rename a single column  
expenditures %>% select(Yr\_1980 = X1980)  
  
# Select and rename the multiple variables with an "X" prefix:  
expenditures %>% select(Yr\_ = starts\_with("X"))  
  
# keep all variables and rename a single variable  
expenditures %>% rename(`2011` = X2011)

## Filtering rows

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. 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 being excluded since it falls within division 3 and its expenditures were < $10B*(FYI - the raw census data are reported in units of $1,000)*.

# Raw census data are in units of $1,000  
sub\_exp %>% filter(Division == 3, X2011 > 10000000)  
## 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

There are additional filtering and subsetting functions that are quite useful:

# remove duplicate rows  
sub\_exp %>% distinct()   
  
# random sample, 50% sample size without replacement  
sub\_exp %>% sample\_frac(size = 0.5, replace = FALSE)  
  
# random sample of 10 rows with replacement  
sub\_exp %>% sample\_n(size = 10, replace = TRUE)  
  
# select rows 3-5  
sub\_exp %>% slice(3:5)  
  
# select top n entries - in this case ranks variable X2011 and selects  
# the rows with the top 5 values  
sub\_exp %>% top\_n(n = 5, wt = X2011)

## Grouping data by categorical variables

Often, observations are nested within groups or categories and our goal 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.

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, when 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. In the example that follows you'll notice that we grouped by Division and there are nine categories for this variable. 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)  
  
group.exp  
## Source: local data frame [50 x 7]  
## Groups: Division [9]  
##   
## Division State X2007 X2008 X2009 X2010 X2011  
## (int) (chr) (int) (int) (int) (int) (int)  
## 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  
## 7 1 Connecticut 7855459 8336789 8708294 8853337 9094036  
## 8 5 Delaware 1437707 1489594 1518786 1549812 1613304  
## 9 5 Florida 22887024 24224114 23328028 23349314 23870090  
## 10 5 Georgia 14828715 16030039 15976945 15730409 15527907  
## .. ... ... ... ... ... ... ...  
  
# we can ungroup our data with  
ungroup(group.exp)  
## Source: local data frame [50 x 7]  
##   
## Division State X2007 X2008 X2009 X2010 X2011  
## (int) (chr) (int) (int) (int) (int) (int)  
## 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  
## 7 1 Connecticut 7855459 8336789 8708294 8853337 9094036  
## 8 5 Delaware 1437707 1489594 1518786 1549812 1613304  
## 9 5 Florida 22887024 24224114 23328028 23349314 23870090  
## 10 5 Georgia 14828715 16030039 15976945 15730409 15527907  
## .. ... ... ... ... ... ... ...

## Performing summary statistics on variables

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 summary statistics when performing exploratory data analysis.

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

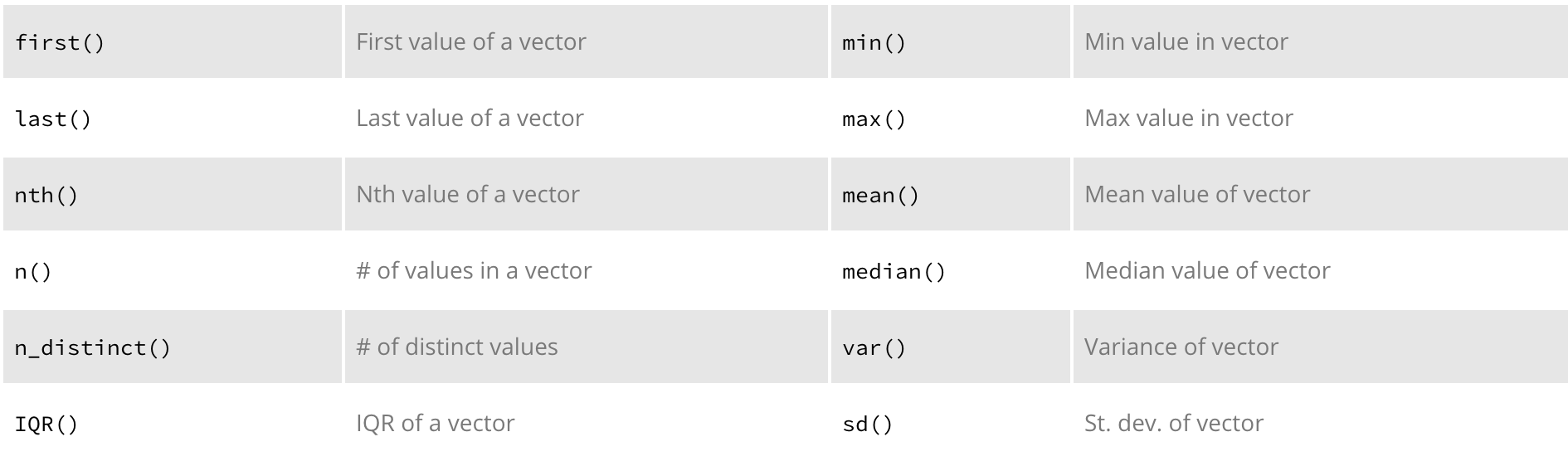
This information is useful, but being able to compare summary statistics at multiple levels is when you really start to gather some insights. This is where the group\_by() function comes in. First, let's group by Division and see how the different regions compared in by 2010 and 2011.

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  
## (int) (dbl) (dbl)  
## 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.

library(tidyr)  
  
sub\_exp %>%  
 gather(Year, Expenditure, X2007:X2011) %>% # turn wide data to long  
 filter(Division == 3) %>% # only assess Division 3  
 group\_by(State) %>% # summarize data by state  
 summarise(Mean = mean(Expenditure), # calculate mean & SD  
 SD = sd(Expenditure))  
## Source: local data frame [5 x 3]  
##   
## State Mean SD  
## (chr) (dbl) (dbl)  
## 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

There are several built-in summary functions in dplyr as displayed below. You can also build in your own functions as well.



Built-in Summary Functions

## Arranging variables by value

Sometimes we wish 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. Let's say we want to assess the average expenditures by 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  
## (int) (dbl) (dbl)  
## 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 a *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  
## (int) (dbl) (dbl)  
## 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

## Joining datasets

Often we have separate dataframes that can have common and differing variables for similar observations and we wish to *join* these dataframes together. dplyr offers multiple joining functions (xxx\_join()) that provide alternative ways to join data frames:

* inner\_join()
* left\_join()
* right\_join()
* full\_join()
* semi\_join()
* anti\_join()

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 data frame which provides inflation adjustment factors for base-year 2012 dollars *(obviously I should use 2015 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 me to join these two data frames. I can apply the following functions to accomplish this:

long\_exp <- sub\_exp %>%  
 gather(Year, Expenditure, X2007:X2011) %>%   
 separate(Year, into=c("x", "Year"), sep = "X") %>%   
 select(-x) %>%   
 mutate(Year = as.numeric(Year))   
  
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 data frame as new variables.

join\_exp <- long\_exp %>% left\_join(inflation)  
  
head(join\_exp)  
## Division State Year Expenditure Annual Inflation  
## 1 6 Alabama 2007 6245031 207.342 0.9030811  
## 2 9 Alaska 2007 1634316 207.342 0.9030811  
## 3 8 Arizona 2007 7815720 207.342 0.9030811  
## 4 7 Arkansas 2007 3997701 207.342 0.9030811  
## 5 9 California 2007 57352599 207.342 0.9030811  
## 6 8 Colorado 2007 6579053 207.342 0.9030811

To illustrate the other joining methods we can use the a and b data frames from the EDAWR package:

library(EDAWR)  
  
a  
## x1 x2  
## 1 A 1  
## 2 B 2  
## 3 C 3  
  
b  
## x1 x2  
## 1 A TRUE  
## 2 B FALSE  
## 3 D TRUE

# include all of a, and join matching rows of b  
left\_join(a, b, by = "x1")  
## x1 x2.x x2.y  
## 1 A 1 TRUE  
## 2 B 2 FALSE  
## 3 C 3 NA  
  
# include all of b, and join matching rows of a  
right\_join(a, b, by = "x1")  
## x1 x2.x x2.y  
## 1 A 1 TRUE  
## 2 B 2 FALSE  
## 3 D NA TRUE  
  
# join data, retain only matching rows in both data frames  
inner\_join(a, b, by = "x1")  
## x1 x2.x x2.y  
## 1 A 1 TRUE  
## 2 B 2 FALSE  
  
# join data, retain all values, all rows  
full\_join(a, b, by = "x1")  
## x1 x2.x x2.y  
## 1 A 1 TRUE  
## 2 B 2 FALSE  
## 3 C 3 NA  
## 4 D NA TRUE  
  
# keep all rows in a that have a match in b  
semi\_join(a, b, by = "x1")  
## x1 x2  
## 1 A 1  
## 2 B 2  
  
# keep all rows in a that do not have a match in b  
anti\_join(a, b, by = "x1")  
## x1 x2  
## 1 C 3

There are additional dplyr functions for merging data sets worth exploring:

intersect(y, z) # Rows that appear in both y and z  
union(y, z) # Rows that appear in either or both y and z  
setdiff(y, z) # Rows that appear in y but not z  
bind\_rows(y, z) # Append z to y as new rows  
bind\_cols(y, z) # Append z to y as new columns

## Creating new variables

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

## Division State Year Expenditure Annual Inflation  
## 1 6 Alabama 2007 6245031 207.342 0.9030811  
## 2 9 Alaska 2007 1634316 207.342 0.9030811  
## 3 8 Arizona 2007 7815720 207.342 0.9030811  
## 4 7 Arkansas 2007 3997701 207.342 0.9030811  
## 5 9 California 2007 57352599 207.342 0.9030811  
## 6 8 Colorado 2007 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)  
## Division State Year Expenditure Annual Inflation Adj\_Exp  
## 1 6 Alabama 2007 6245031 207.342 0.9030811 6915249  
## 2 9 Alaska 2007 1634316 207.342 0.9030811 1809711  
## 3 8 Arizona 2007 7815720 207.342 0.9030811 8654505  
## 4 7 Arkansas 2007 3997701 207.342 0.9030811 4426735  
## 5 9 California 2007 57352599 207.342 0.9030811 63507696  
## 6 8 Colorado 2007 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)  
## Division State Year Expenditure Annual Inflation Adj\_Exp Rank  
## 1 9 California 2010 58248662 218.056 0.9497461 61330774 1  
## 2 2 New York 2010 50251461 218.056 0.9497461 52910417 2  
## 3 7 Texas 2010 42621886 218.056 0.9497461 44877138 3  
## 4 3 Illinois 2010 24695773 218.056 0.9497461 26002501 4  
## 5 2 New Jersey 2010 24261392 218.056 0.9497461 25545135 5  
## 6 5 Florida 2010 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))  
## Division State Year Expenditure Annual Inflation Adj\_Exp Perc\_Chg  
## 1 3 Ohio 2007 18251361 207.342 0.9030811 20210102 NA  
## 2 3 Ohio 2008 18892374 215.303 0.9377553 20146378 -0.003153057  
## 3 3 Ohio 2009 19387318 214.537 0.9344190 20747992 0.029862103  
## 4 3 Ohio 2010 19801670 218.056 0.9497461 20849436 0.004889357  
## 5 3 Ohio 2011 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)*

cum\_pct <- inflation\_adj %>%  
 filter(Year == 2011) %>%  
 arrange(desc(Adj\_Exp)) %>%  
 mutate(Pct\_of\_Total = Adj\_Exp/sum(Adj\_Exp),  
 Cum\_Perc = cumsum(Pct\_of\_Total)) %>%  
 select(-Expenditure, -Annual, -Inflation)  
   
head(cum\_pct, 8)  
## Division State Year Adj\_Exp Pct\_of\_Total Cum\_Perc  
## 1 9 California 2011 58717324 0.10943237 0.1094324  
## 2 2 New York 2011 52575244 0.09798528 0.2074177  
## 3 7 Texas 2011 43751346 0.08154005 0.2889577  
## 4 3 Illinois 2011 25062609 0.04670957 0.3356673  
## 5 5 Florida 2011 24364070 0.04540769 0.3810750  
## 6 2 New Jersey 2011 24128484 0.04496862 0.4260436  
## 7 2 Pennsylvania 2011 23971218 0.04467552 0.4707191  
## 8 3 Ohio 2011 20402582 0.03802460 0.5087437

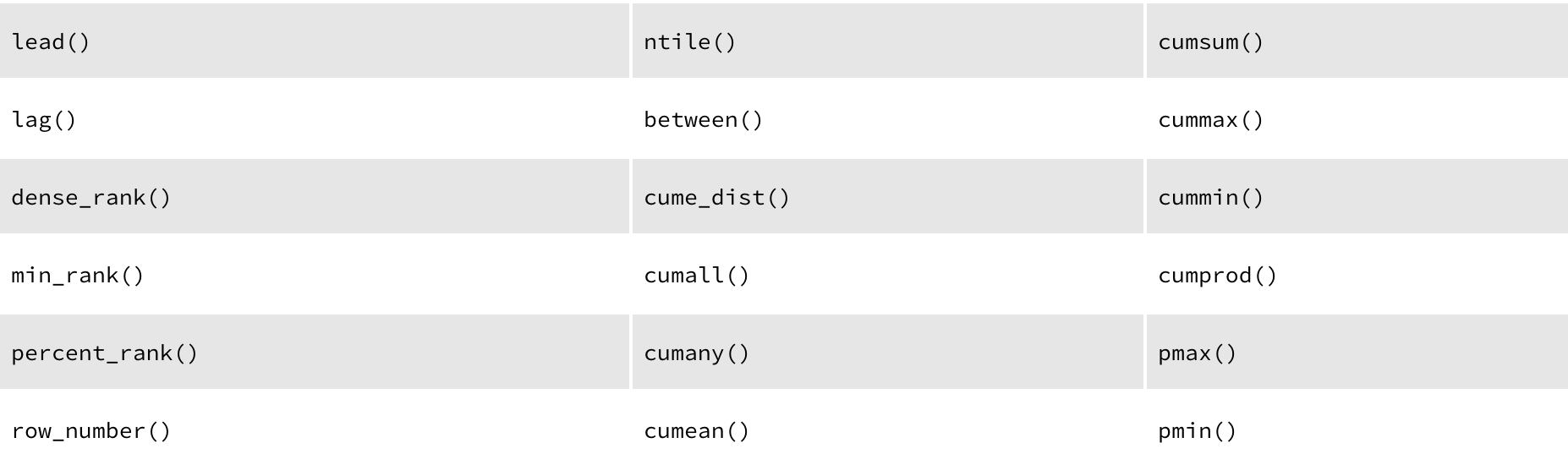
An alternative to mutate() is transmute() which creates a new variable and then drops the other variables. In essence, it allows you to create a new data frame with only the new variables created. We can perform the same string of functions as above but this time use transmute to only keep the newly created variables.

inflation\_adj %>%  
 filter(Year == 2011) %>%  
 arrange(desc(Adj\_Exp)) %>%  
 transmute(Pct\_of\_Total = Adj\_Exp/sum(Adj\_Exp),  
 Cum\_Perc = cumsum(Pct\_of\_Total)) %>%  
 head()  
## Pct\_of\_Total Cum\_Perc  
## 1 0.10943237 0.1094324  
## 2 0.09798528 0.2074177  
## 3 0.08154005 0.2889577  
## 4 0.04670957 0.3356673  
## 5 0.04540769 0.3810750  
## 6 0.04496862 0.4260436

Lastly, you can easily also apply the summarise and mutate functions to multiple columns by using summarise\_each() and mutate\_each() respectively.

# calculate the mean for each division with summarise\_each  
# call the function of interest with the `funs()` argument  
sub\_exp %>%  
 select(-State) %>%  
 group\_by(Division) %>%  
 summarise\_each(funs(mean)) %>%  
 head()  
## Source: local data frame [6 x 6]  
##   
## Division X2007 X2008 X2009 X2010 X2011  
## (int) (dbl) (dbl) (dbl) (dbl) (dbl)  
## 1 1 4680691 4952992 5173184 5121003 5222277  
## 2 2 28844158 30652645 31304697 32415457 32877923  
## 3 3 14823590 15293644 15895459 16322489 16270159  
## 4 4 4175766 4425739 4658533 4672332 4672687  
## 5 5 10230416 10857410 11018102 10975194 11023526  
## 6 6 5584277 6023424 6076507 6161967 6267490  
  
# for each division calculate the percent of total   
# expenditures for each state across each year  
sub\_exp %>%  
 select(-State) %>%  
 group\_by(Division) %>%  
 mutate\_each(funs(. / sum(.))) %>%  
 head()  
## Source: local data frame [6 x 6]  
## Groups: Division [4]  
##   
## Division X2007 X2008 X2009 X2010 X2011  
## (int) (dbl) (dbl) (dbl) (dbl) (dbl)  
## 1 6 0.27958099 0.28357787 0.27498705 0.27063262 0.26298109  
## 2 9 0.02184221 0.02387438 0.02515947 0.02682018 0.02846193  
## 3 8 0.28093187 0.27793321 0.28144201 0.27229536 0.26854292  
## 4 7 0.07854895 0.07565703 0.07402700 0.07474621 0.07630156  
## 5 9 0.76650258 0.76625202 0.75304632 0.74962818 0.74380904  
## 6 8 0.23648054 0.24272678 0.23179279 0.23848536 0.23857413

Similar to the summary function, dplyr allows you to build in your own functions to be applied within mutate\_each() and also has the following built in functions that can be applied.



Built-in Functions for mutate\_each()

## Additional resources

This chapter introduced you to dplyr's basic set of tools and demonstrated how to use them on data frames. Additional resources are available that go into more detail or provide additional examples of how to use dpyr. In addition, there are other resouces that illustrate how dplyr can perform tasks not mentioned in this chapter such as connecting to remote databases and translating your R code into SQL code for data pulls.

* [Data wrangling presentation](http://bradleyboehmke.github.io/2015/10/data-wrangling-presentation.html) I gave at Miami University
* [dplyr reference manual](https://cran.r-project.org/web/packages/dplyr/dplyr.pdf)
* 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/)

1. Wickham, H. (2014). "Tidy data." Journal of Statistical Software, 59(10). [[document](http://jstatsoft.org/v59/i10)] [↑](#footnote-ref-26)
2. Ibid [↑](#footnote-ref-28)