An Introduction to Tidyverse

Joey Stanley
Doctoral Candidate in Linguistics, University of Georgia
joeystanley.com
orcid.org/0000-0002-9185-0048

Presented at the UGA Willson Center DigiLab
Friday, November 10, 2017

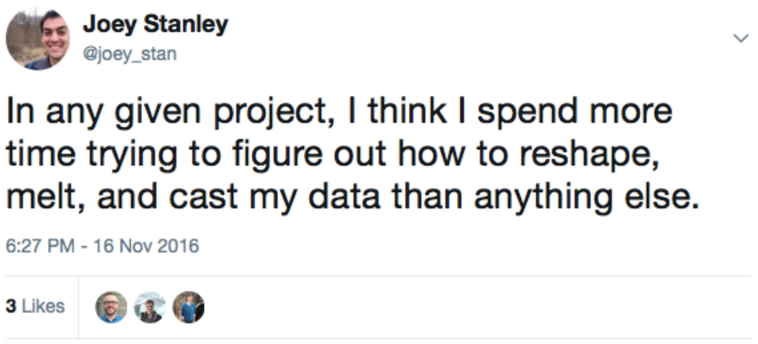
This is the third installment of the R workshop series. This document will cover some of the basics of importing, cleaning, and transforming your data using the Tidyverse: (1) some general thoughts on tidyverse; (2) getting data into R from csv files or Microsoft Excel with some explanation of "tibbles"; (3) transforming your data by removing, reordering, adding columns; (4) cleaning your data by renaming columns, renaming individual levels within columns, collapsing factors, anf iltering; (5) reshaping your data from wide to tall format and vice versa; (6) merging datasets and creating essentially an R version of Excel's lookup tables; and (7) some concluding remarks.

Download this PDF from my website at
joeystanley.com/r

(Updated November 10, 2017)

1 Introduction

Working with data can be super frustrating. I've been there. In fact, this tweet from just under a year ago, only hints at the frustration I was feeling at the time.


Joey Stanley
@joey_stan

In any given project, I think I spend more time trying to figure out how to reshape, melt, and cast my data than anything else.

6:27 PM - 16 Nov 2016

3 Likes

I'm pretty sure I had spent 6–8 hours working on transforming my data from "wide" to "tall" to get it to work in some visualization (we'll get to what that means later). I don't even think I ended up getting it done in the end, and I found an alternative solution.

That was about a year ago, and now I'm giving a workshop on this stuff. This just goes to show that in less than a year you can learn to do this stuff effectively in your own code as well.

1.1 What is the tidyverse?

According to its website (tidyverse.org), "The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying philosophy and common APIs." Let's break that down.

1. It's a collection of R packages—When you install the `tidyverse` package, all you're doing is installing several other packages that fall its umbrella. Some of the packages that are part of `tidyverse` include `dplyr`, `tidyr`, and `ggplot2`, which are among the most popular R packages. There are others that are super useful like `readxl`, `forcats`, and `stringr` that are part of the tidyverse, but don't come installed automatically with the `tidyverse` package, so you'll have to lead them explicitly.

2. They share an underlying philosophy and common APIs—This is one of the reasons that makes the tidyverse so great: all the packages seamlessly integrate and work together harmoniously. There's nothing worse than having to modify your dataframe in some way in order to get it to work for some function. With the tidyverse, you can jump from one of its packages to another and send data around and nothing will ever get mad at you.

3. They are designed for data science—A lot of what R can do is for statistical modeling on your data. Tidyverse doesn't do a lot of statistics, but it can help you every step of the way otherwise. You can read in your data, make any modifications, and visualize it with no

problem. When it actually comes to fancy statistical models though, you'll have to find more specialized packages. This is not to say that the tidyverse is deficient in any way: reshaping and tidying your data is no small feat and the tidyverse makes this a *lot* easier.

4. They are opinionated—The author of these packages, Hadley Wickham, is very open about how they are created, and says that this is just *one* way to work with data in R. You're certainly welcome to work with your data another way, but using the tidyverse is not the only solution: it's just Hadley's opinion of how it should be best done.

There's a ridiculous amount to cover with this suite. Just `ggplot2` alone would take a hundred workshops to cover everything. The possibilities are nearly endless and I obviously can't teach you everything in just a single 1-hour workshop. To give you an idea of what functions in the tidyverse can do, here's a brief summary of *some* of its packages.

1. `ggplot2` is a very popular suite for visualizing data. Unlike other visualizations you might do, `ggplot2` allows for pretty much infinite customizability in your plots, which is really handy if you have nitpicky details you want to control.

2. `dplyr` lets you manipulate your data by doing things like adding and removing columns, filtering and subsetting your data, and summarizing your data (such as getting the average of some value per group within your data).

3. `tidyr` lets you reshape your data from "wide" to "tall" format. It's like `reshape` and `reshape2` (also written by Hadley Wickham), but in my opinion it does the job better with code that's easier to interpret.

4. `readxl` makes allows you to read in Excel files directly into R without the need to convert to .csv first

5. `forcats` comes with a bunch of functions for working with categorical data.

6. `stringr` makes it easier to work with text in your data.

There are many other packages that are part of the tidyverse that handle more specific tasks like specific data types. There are also ones that are more for the R programmer and can help you publish your R scripts into your own libraries. For now, I'll stick with just the basics, but be aware that there is much, *much* more. For more information on what's possible, check out the book *R for Data Science* by Garrett Grolemund and Hadley Wickham (available for free viewing at http://r4ds.had.co.nz), which goes into more detail about what functions in the tidyverse can do. Much of this workshop is based on a couple chapters from that book.

1.2 Installation

Just like any other R package, installing `tidyverse` is straightforward.

```
install.packages("tidyverse")
```

You only need to run this once and then it's on your computer. This doesn't make it automatically available to R though, so you'll need to explicitly tell R you'll be working with the package, using the `library` function.

There will be some warning messages, but you can probably safely ignore those.

Side note, as of about two days ago (November 8, 2017), `tidyverse` updated to a new version (v1.2.0). Looking through the release notes, the changes look mostly minor so it shouldn't affect the code in this workshop, which was prepared using an older version of the package (v1.1.1).

2 GETTING DATA INTO R

The first step to any project in R is to get your data into R. For this we'll make use of two functions. The first is `read_csv` and is a part of the `readr` package, which was automatically installed and loaded when you loaded `tidyverse`. For this section I'll be drawing from Chapter 11 "Data Import" from *R for Data Science*, which can be read here: http://r4ds.had.co.nz/data-import.html.

2.1 CSV FILES

If you've read data into R before, you've probably used the standard `read.csv` (with a period) function. This one works just fine and you can get by perfectly well with it. However, tidyverse's `read_csv` (with an underscore), has some additional perks that the standard function doesn't come with. To show the first one, let's go ahead and read in some data. This data contains all the McDonalds menu items as well as some basic nutritional information and was downloaded for free at Kaggle.com. You can download it directly from my website using this code:

```
menu <- read_csv("http://joeystanley.com/data/menu.csv")

## Parsed with column specification:
## cols(
##   Category = col_character(),
##   Item = col_character(),
##   Oz = col_double(),
##   Calories = col_integer(),
##   Fat = col_double(),
##   Sugars = col_integer()
## )
```

The first perk of `read_csv` is that it gives you some output showing you how it parsed each column of your spreadsheet. Not all data should be treated the same: numbers are very different from text, and R (as well as you) should be aware of what data types are contained in your file so that it (and you) can work with it the best way possible. There are some heuristics that determine how `read_csv` parses your data which we won't get into here, but it's nice to see what the final result was just to make sure that numbers get treated as numbers, text as text, etc. Later in this workshop we'll see one way to change it in case the function got something wrong.

The second perk of `read_csv` is that it turns your data into what's called a "tibble". A tibble is tidyverse's version of a dataframe. Essentially, you can think of it as just a dataframe, but again, with some additional perks. One major difference between dataframes and tibbles is when you print them. When you print a regular dataframe, it'll vomit out *everything*. You get all rows (even if there are many of them) and all columns, which can be hard to read if your screen isn't wide enough to display them since they'll spill over into multiple rows. Here's our menu data printed out as a dataframe.

```
menu.df <- as.data.frame(menu)
menu.df

##     Category                                                         Item
## 1  Breakfast                                                 Egg McMuffin
## 2  Breakfast                                            Egg White Delight
## 3  Breakfast                                             Sausage McMuffin
## 4  Breakfast                                    Sausage McMuffin with Egg
## 5  Breakfast                             Sausage McMuffin with Egg Whites
## 6  Breakfast                                         Steak & Egg McMuffin
## 7  Breakfast                    Bacon, Egg & Cheese Biscuit (Regular Biscuit)
## 8  Breakfast                      Bacon, Egg & Cheese Biscuit (Large Biscuit)
## 9  Breakfast Bacon, Egg & Cheese Biscuit with Egg Whites (Regular Biscuit)
## 10 Breakfast   Bacon, Egg & Cheese Biscuit with Egg Whites (Large Biscuit)
##     Oz Calories Fat Sugars
## 1  4.8      300  13      3
## 2  4.8      250   8      3
## 3  3.9      370  23      2
## 4  5.7      450  28      2
## 5  5.7      400  23      2
## 6  6.5      430  23      3
## 7  5.3      460  26      3
## 8  5.8      520  30      4
## 9  5.4      410  20      3
## 10 5.9      470  25      4
```

I've truncated the output to save space, but on your screen you'll see that this displays the entire contents of the dataframe. When we print the tibble version, it's a lot shorter.

```
menu

## # A tibble: 260 x 6
##     Category                                                         Item
##        <chr>                                                        <chr>
##   1 Breakfast                                                Egg McMuffin
##   2 Breakfast                                           Egg White Delight
##   3 Breakfast                                            Sausage McMuffin
##   4 Breakfast                                   Sausage McMuffin with Egg
##   5 Breakfast                            Sausage McMuffin with Egg Whites
##   6 Breakfast                                        Steak & Egg McMuffin
##   7 Breakfast                   Bacon, Egg & Cheese Biscuit (Regular Biscuit)
##   8 Breakfast                     Bacon, Egg & Cheese Biscuit (Large Biscuit)
##  9 Breakfast Bacon, Egg & Cheese Biscuit with Egg Whites (Regular Biscuit)
## 10 Breakfast   Bacon, Egg & Cheese Biscuit with Egg Whites (Large Biscuit)
## # ... with 250 more rows, and 4 more variables: Oz <dbl>, Calories <int>,
## #   Fat <dbl>, Sugars <int>
```

When you look at a tibble, you only peek at the data. You get the first 10 rows and only as many columns as can fit on your screen without spilling over into a new column. If there are additional columns, they're lised at the bottom. You also get to see what datatype each column is at the top of each column below the column names. Ths printing feature makes it easy to examine just a portion of your data without flooding your R Console.

Finally, using `read_csv` is actually about 10 times faster than the regular `read.csv` function. If you work with very large datasets, this is a very good thing. Furthermore, if it does take more than about five seconds to read in your file, you'll actually get a little progress bar down in your R Console saying how much it's done and how long it has taken. This is nice to see so you know that R is making progress and didn't crash.

Side note, since the menu items are so long, I'm going to truncate it for display purposes only. You don't have to do this.

```
menu$Item <- str_sub(menu$Item, 1, 24)
head(menu)
```

```
## # A tibble: 6 x 6
##    Category                     Item    Oz Calories   Fat Sugars
##       <chr>                    <chr> <dbl>    <int> <dbl>  <int>
## 1 Breakfast              Egg McMuffin   4.8      300    13      3
## 2 Breakfast         Egg White Delight   4.8      250     8      3
## 3 Breakfast          Sausage McMuffin   3.9      370    23      2
## 4 Breakfast Sausage McMuffin with Eg   5.7      450    28      2
## 5 Breakfast Sausage McMuffin with Eg   5.7      400    23      2
## 6 Breakfast      Steak & Egg McMuffin   6.5      430    23      3
```

2.2 EXCEL FILES

In the Intro to R Workshop, I said that it's possible to load Excel files directly into R. This is possible thanks to the `read_excel` function which is in the `readxl` package. This package is part of the tidyverse, but does not come standard in the `tidyverse` library, so you'll have to load it explicitly (it should be installed already).

```
library(readxl)
```

The syntax of `read_excel` is very similar to `read_csv` (a trend you'll notice over and over in the tidyverse). All you need to do is specify the path to the file itself. In the case of Excel files, you can't read them directly from a website like you can with csv files, so you'll have to download the data (just go to http://joeystanley.com/data/snoozing.xlsx and it'll automatically download) and then load it from your computer from wherever you saved your file.

```
snooze <- read_excel("/Users/joeystanley/Desktop/snoozing.xlsx")
```

This file contains some fake data I created on how sleeping patterns. We'll get to it later in the workshop. The crucial thing now is that it's an Excel file and it can be read in directly into R which is pretty cool.

By default, this reads in the first sheet of the Excel file. You can change this by adding the `sheet` argument and providing either the name of the sheet, or the sheet number. This particular file contains two sheets: the first has all the data and the second is nonsense. We can specify that we want the second sheet (called "blank") by indicating it:

```
blank <- read_excel("/Users/joeystanley/Desktop/snoozing.xlsx",
                    sheet="blank")
```

The perks of using `read_excel` are essentially the same as using `read_csv`: your data is saved as a tibble and it's much faster. Personally, I love this function. I used to work in Excel to prepare my data, but then every time I wanted to make a change, I had to edit the Excel file and then save it as a csv, and *then* read it into R. It was just too many steps. Now, I can work with Excel directly and bypass the csv step entirely.

There is *much* more that you can do with reading data into R, not only with just those two functions, but with many other specialized functions in `readr` and `readxl`, including those for reading in specific data formats. For example, you can specify column names or column types as you read it in so you don't have to change them later. I'll let you explore those on your own.

3 Transforming your data

Now that you've got your data loaded in, you'll proably need to do some processing before moving on to analysis. In my experience, my R code always starts off with many lines of preprocessing to make sure everything is nice and clean and consistently formatted before moving on to the statistics and visualizations.

What do I mean by "processing"? Well, it's likely that your data isn't exactly the way it needs to be for analysis. Maybe the column names aren't right, or a number that's supposed to be a number is actually being treated like text, or you may want to add, remove, or reorder columns entirely. Even if your data is pristine, you may have to do set a reference level for categorical data or filter the data in some way. All of this can be done using functions in the `dplyr` package, one of the core `tidyverse` packages. I'll cover some basics, but you can read more about the topics in this section *R for Data Science* chapters 18 on Pipes (http://r4ds.had.co.nz/pipes.html), and Chapter 5 "Data Transformation" (http://r4ds.had.co.nz/transform.html)

3.1 Piping

Before we move on, I want to pause and introduce some tidyverse syntax. To explain how it's used, let's say we want to see the McDonalds menu items. We could split this up into two steps, reading it in and then printing it:

```
menu <- read_csv("http://joeystanley.com/data/menu.csv")
print(menu)
```

```
## # A tibble: 260 x 6
##     Category                    Item    Oz Calories   Fat Sugars
##        <chr>                   <chr> <dbl>    <int> <dbl>  <int>
##  1 Breakfast            Egg McMuffin   4.8      300    13      3
##  2 Breakfast       Egg White Delight   4.8      250     8      3
##  3 Breakfast        Sausage McMuffin   3.9      370    23      2
##  4 Breakfast Sausage McMuffin with Eg   5.7      450    28      2
##  5 Breakfast Sausage McMuffin with Eg   5.7      400    23      2
##  6 Breakfast      Steak & Egg McMuffin   6.5      430    23      3
##  7 Breakfast Bacon, Egg & Cheese Bisc   5.3      460    26      3
##  8 Breakfast Bacon, Egg & Cheese Bisc   5.8      520    30      4
##  9 Breakfast Bacon, Egg & Cheese Bisc   5.4      410    20      3
## 10 Breakfast Bacon, Egg & Cheese Bisc   5.9      470    25      4
## # ... with 250 more rows
```

But that seems a little repetitive. With menu it's not so bad because it's small name, but if you've got some long name that's similar to other dataframes in your script, it could get annoying. It's like listening to someone never using a pronoun and saying a peron's name every sentence: "Joey's my friend. Joey and I went to the store and then Joey's phone fell out of Joey's pocket…". Wouldn't it be nice if R could sort of "remember" what we're working with?

For this reason, we have the *pipe*. This little function is one of the most useful features of the tidyverse and it looks like this: %>%. What the pipe does is it takes the output of the function on the left and feeds it to the right function as its first argument.

So if we wanted to read in the McDonalds menu data and then print it, we can pipe the results of read_csv into the print function and take care of this all in one go. The following block is does the exact same thing as the previous one.

```
menu <- read_csv("http://joeystanley.com/data/menu.csv") %>%
    print()
```

Notice that in the print function, you don't need to put the same of the dataframe in. Basically, the pipe can be read as, "and then with this, do…". So "read this cvs file *and then with that*, print it".

There are lots of benefits to using the pipe. First, it saves on typing. You only have to type the name of the dataframe (menu) once. This is especially nice if you have some long dataframe name. Second, when you execute the command, it'll interpret both lines as a single line, so it saves a couple cliks/keystrokes, etc. And with the pipe, you're not limited to just two functions, you can stack as many of them as you want to make what looks like a "paragraph" of code that all gets treated as one. More on that in a sec. Finally, because it's all treated as one command, you don't need to come up with names for the various intermediate steps (which are confusing if you have too many, and get very long) like menu.clean or menu.clean.renamed or menu3.

The reason why this works is because in pretty much all the functions in the tidyverse, the first argument is `data=...`. Since the pipe takes the thing on the left and makes it the first argument on the right, if that first argument is `data`, then it'll fill that slot perfectly.

3.2 Removing Columns

So when we look at our McDonald's data, we see that the Item name is super wide (unless you truncated it like I did). It was so wide that when we printed anything, we didn't see any of the other columns. We could just remove it entirely, which would free things up. We can do this with the `select` function.

With `select`, as long as we're piping some data into it, all we need to do is specifiy which columns we want to keep, and it'll keep them. So if we wanted to just look at the Calories and Sugars, we could do that easily:

```
menu %>%
    select(Calories, Sugars)

## # A tibble: 260 x 2
##    Calories Sugars
##       <int>  <int>
##  1      300      3
##  2      250      3
##  3      370      2
##  4      450      2
##  5      400      2
##  6      430      3
##  7      460      3
##  8      520      4
##  9      410      3
## 10      470      4
## # ... with 250 more rows
```

Perfect, with just a simple line of code, we were able to do that. Notice that we don't even need quotes around the column names, which saves some typing. If we wanted to look at everything *except* the menu item name, we just put a minus sign before it:

```
menu %>%
    select(-Item)

## # A tibble: 260 x 5
##     Category    Oz Calories   Fat Sugars
##        <chr> <dbl>    <int> <dbl>  <int>
##  1 Breakfast   4.8      300    13      3
##  2 Breakfast   4.8      250     8      3
##  3 Breakfast   3.9      370    23      2
##  4 Breakfast   5.7      450    28      2
##  5 Breakfast   5.7      400    23      2
##  6 Breakfast   6.5      430    23      3
##  7 Breakfast   5.3      460    26      3
##  8 Breakfast   5.8      520    30      4
##  9 Breakfast   5.4      410    20      3
## 10 Breakfast   5.9      470    25      4
## # ... with 250 more rows
```

It removes that column.

What if we wanted the Oz, Colaries, Fat, and Sugars column and wanted to leave off Category and Item? You could add minus signs before each one you want to remove:

```
menu %>%
    select(-Category, -Item)

## # A tibble: 260 x 4
##         Oz Calories   Fat Sugars
##      <dbl>    <int> <dbl>  <int>
##   1   4.8       300    13      3
##   2   4.8       250     8      3
##   3   3.9       370    23      2
##   4   5.7       450    28      2
##   5   5.7       400    23      2
##   6   6.5       430    23      3
##   7   5.3       460    26      3
##   8   5.8       520    30      4
##   9   5.4       410    20      3
## 10   5.9       470    25      4
## # ... with 250 more rows
```

Alternatively, if the columns you want to keep are in order, you can just print the first and last one and put a colon (`:`) between them, and it'll get everything in between:

```
menu %>%
    select(Oz:Sugars)

## # A tibble: 260 x 4
##         Oz Calories   Fat Sugars
##      <dbl>    <int> <dbl>  <int>
##   1   4.8       300    13      3
##   2   4.8       250     8      3
##   3   3.9       370    23      2
##   4   5.7       450    28      2
##   5   5.7       400    23      2
##   6   6.5       430    23      3
##   7   5.3       460    26      3
##   8   5.8       520    30      4
##   9   5.4       410    20      3
## 10   5.9       470    25      4
## # ... with 250 more rows
```

This is analgous to the R syntax of creating sequential numbers (a list from 1 to 10 would be simply `1:10`).

So using the `select` function is super useful because you can very quickly pick out the columns you want to see or remove columns you don't need. The code is easy and intuitive to write and it's clear from reading it what it's doing. That's what makes tidyverse so useful.

3.3 Reordering Columns

You can also use `select` to reorder your columns. Just put the column names in whatever order you want and that's how they'll appear:

```
menu %>%
    select(Oz, Sugars, Fat, Calories)
```

```
## # A tibble: 260 x 4
##       Oz Sugars   Fat Calories
##    <dbl>  <int> <dbl>    <int>
##  1   4.8      3    13      300
##  2   4.8      3     8      250
##  3   3.9      2    23      370
##  4   5.7      2    28      450
##  5   5.7      2    23      400
##  6   6.5      3    23      430
##  7   5.3      3    26      460
##  8   5.8      4    30      520
##  9   5.4      3    20      410
## 10   5.9      4    25      470
## # ... with 250 more rows
```

To accomplish the same thing in base R, as far as I know you'd have to specify the columns by numbers (the above code would be something like `menu <- menu[c(3,6,5,4),]`). This is super annoying for lost of reasons: 1) you have to figure out what number each column is, which is not easy if you have many columns; 2) if your dataset changes, your code breaks because column 3 might not always be the `Oz` column; 3) it's not clear from reading the code what columns you're keeping. Tidyverse solves all these problems by just using the names instead of numbers.

An alternative in base R is to use the `subset` function, which behaves very similarly to `select`. Here's a base R equivalent:

```
# Base R equivalent
subset(menu, select=c(Oz, Sugars, Fat, Calories))
```

```
## # A tibble: 260 x 4
##       Oz Sugars   Fat Calories
##    <dbl>  <int> <dbl>    <int>
##  1   4.8      3    13      300
##  2   4.8      3     8      250
##  3   3.9      2    23      370
##  4   5.7      2    28      450
##  5   5.7      2    23      400
##  6   6.5      3    23      430
##  7   5.3      3    26      460
##  8   5.8      4    30      520
##  9   5.4      3    20      410
## 10   5.9      4    25      470
## # ... with 250 more rows
```

With just a few columns, it's not a big deal to type everything in tidyverse, but if you have a spreadsheet with many columns, it can still get tedious, especially if all you want to do is move one column around. For this reason, `dplyr` comes with the `everything` function, which is shorthand for "all other variables."" So if you wanted to move the `Sugars` column to the front, you can just type that column name and then `everything()`:

```
menu %>%
    select(Sugars, everything())
```

```
## # A tibble: 260 x 6
##    Sugars  Category                     Item    Oz Calories   Fat
##     <int>     <chr>                    <chr> <dbl>    <int> <dbl>
##  1      3 Breakfast             Egg McMuffin   4.8      300    13
##  2      3 Breakfast        Egg White Delight   4.8      250     8
##  3      2 Breakfast         Sausage McMuffin   3.9      370    23
##  4      2 Breakfast Sausage McMuffin with Eg   5.7      450    28
##  5      2 Breakfast Sausage McMuffin with Eg   5.7      400    23
##  6      3 Breakfast      Steak & Egg McMuffin   6.5      430    23
##  7      3 Breakfast Bacon, Egg & Cheese Bisc   5.3      460    26
##  8      4 Breakfast Bacon, Egg & Cheese Bisc   5.8      520    30
##  9      3 Breakfast Bacon, Egg & Cheese Bisc   5.4      410    20
## 10      4 Breakfast Bacon, Egg & Cheese Bisc   5.9      470    25
## # ... with 250 more rows
```

Again, this is super useful for when you have tons of columns and only need to move a couple up towards the front.

So the `select` function is great because it can take care of a lot of stuff like removing and reordering columns all at once. The syntax is intuitive and clear. And, as it turns out, the function "returns" your modified dataframe, so you can then pipe it onto other tidyverse functions for additional processing. Pretty cool.

3.4 Adding columns

We've seen how to remove and reorder columns with `select`, so how do we add columns? Let's say we want to take our McDonalds data and create a new column. We have information about the weight of the item (in ounces) and both the fat and sugar content. What if we wanted to look at the most sugar per ounce? For this, we can use the `mutate` function, which creates a new column and tags it onto the end.

```
menu %>%
    mutate(sugar_per_oz = Sugars / Oz) %>%
    select(sugar_per_oz, everything())
```

```
## # A tibble: 260 x 7
##    sugar_per_oz  Category                     Item    Oz Calories   Fat
##           <dbl>     <chr>                    <chr> <dbl>    <int> <dbl>
## 1     0.6250000 Breakfast             Egg McMuffin   4.8      300    13
## 2     0.6250000 Breakfast        Egg White Delight   4.8      250     8
## 3     0.5128205 Breakfast         Sausage McMuffin   3.9      370    23
## 4     0.3508772 Breakfast Sausage McMuffin with Eg   5.7      450    28
## 5     0.3508772 Breakfast Sausage McMuffin with Eg   5.7      400    23
## 6     0.4615385 Breakfast      Steak & Egg McMuffin   6.5      430    23
## 7     0.5660377 Breakfast Bacon, Egg & Cheese Bisc   5.3      460    26
## 8     0.6896552 Breakfast Bacon, Egg & Cheese Bisc   5.8      520    30
## 9     0.5555556 Breakfast Bacon, Egg & Cheese Bisc   5.4      410    20
## 10    0.6779661 Breakfast Bacon, Egg & Cheese Bisc   5.9      470    25
## # ... with 250 more rows, and 1 more variables: Sugars <int>
```

The syntax here is that you type the name of the new column you want to create (not in quotes), an equals sign, and then do whatever calculations or functions you want on existing columns (also not in quotes). So here, we're taking the `Sugars` and dividing it by the `Oz` column. This creates a new column called `sugar_per_oz`, tagged on at the end of our dataframe. I then piped it to a `select` function, which reorders the columns so that this new one is at the front (so we can see it).

If you wanted to do the same thing in base R, it's possible and not too much of a headache. But you'd need to create some intermediate dataframe for each step along the way.

```
# Base R equivalent
menu_temp <- menu
menu_temp$sugar_per_oz <- menu_temp$Sugars / menu_temp$Oz
menu_temp <- subset(menu_temp, select=c(sugar_per_oz, Category:Sugars))
```

When adding columns using `mutate`, you can add as many additional columns as you want into a single function as long as they're separated by commas. Here, I create the `sugar_per_oz` column as well as `fat_per_oz` and `calories_per_oz`.

```
menu %>%
    mutate(sugar_per_oz = Sugars / Oz,
           fat_per_oz   = Fat / Oz,
           calories_per_oz = Calories / Oz) %>%
    select(sugar_per_oz, fat_per_oz, calories_per_oz, Item, everything())
```

```
## # A tibble: 260 x 9
##    sugar_per_oz fat_per_oz calories_per_oz                     Item
##           <dbl>      <dbl>           <dbl>                    <chr>
## 1     0.6250000   2.708333        62.50000             Egg McMuffin
## 2     0.6250000   1.666667        52.08333        Egg White Delight
## 3     0.5128205   5.897436        94.87179         Sausage McMuffin
## 4     0.3508772   4.912281        78.94737 Sausage McMuffin with Eg
## 5     0.3508772   4.035088        70.17544 Sausage McMuffin with Eg
## 6     0.4615385   3.538462        66.15385      Steak & Egg McMuffin
## 7     0.5660377   4.905660        86.79245 Bacon, Egg & Cheese Bisc
## 8     0.6896552   5.172414        89.65517 Bacon, Egg & Cheese Bisc
##  9    0.5555556   3.703704        75.92593 Bacon, Egg & Cheese Bisc
## 10    0.6779661   4.237288        79.66102 Bacon, Egg & Cheese Bisc
## # ... with 250 more rows, and 5 more variables: Category <chr>, Oz <dbl>,
## #   Calories <int>, Fat <dbl>, Sugars <int>
```

So the `mutate` function is really handy for adding new columns, again with easy-to-understand syntax that makes it easy to read and write.

3.5 Global changes to columns

With `mutate` you can actually make some substantial changes to columns. Let's say you like the `Oz` column, but you want it rounded to the nearest integer, rather than with decimals. We can do this with the base R `round` funtion, specifiying that we want it to round to zero decimal places. (I'll rearrange the columns so you can more easily see the changes.)

```
menu %>%
    mutate(Oz_rounded = round(Oz, 0)) %>%
    select(Category:Oz, Oz_rounded, everything())

## # A tibble: 260 x 7
##     Category                     Item    Oz Oz_rounded Calories   Fat
##        <chr>                    <chr> <dbl>      <dbl>    <int> <dbl>
##  1 Breakfast              Egg McMuffin   4.8          5      300    13
##  2 Breakfast         Egg White Delight   4.8          5      250     8
##  3 Breakfast          Sausage McMuffin   3.9          4      370    23
##  4 Breakfast Sausage McMuffin with Eg   5.7          6      450    28
##  5 Breakfast Sausage McMuffin with Eg   5.7          6      400    23
##  6 Breakfast      Steak & Egg McMuffin   6.5          6      430    23
##  7 Breakfast Bacon, Egg & Cheese Bisc   5.3          5      460    26
##  8 Breakfast Bacon, Egg & Cheese Bisc   5.8          6      520    30
##  9 Breakfast Bacon, Egg & Cheese Bisc   5.4          5      410    20
## 10 Breakfast Bacon, Egg & Cheese Bisc   5.9          6      470    25
## # ... with 250 more rows, and 1 more variables: Sugars <int>
```

In fact, if we know that's what we want to do with the Oz column, we can just overwrite it by creating a "new column with the same name as an existing one.

```
menu %>%
    mutate(Oz = round(Oz, 0))

## # A tibble: 260 x 6
##     Category                     Item    Oz Calories   Fat Sugars
##        <chr>                    <chr> <dbl>    <int> <dbl>  <int>
##  1 Breakfast              Egg McMuffin     5      300    13      3
##  2 Breakfast         Egg White Delight     5      250     8      3
##  3 Breakfast          Sausage McMuffin     4      370    23      2
##  4 Breakfast Sausage McMuffin with Eg     6      450    28      2
##  5 Breakfast Sausage McMuffin with Eg     6      400    23      2
##  6 Breakfast      Steak & Egg McMuffin     6      430    23      3
##  7 Breakfast Bacon, Egg & Cheese Bisc     5      460    26      3
##  8 Breakfast Bacon, Egg & Cheese Bisc     6      520    30      4
##  9 Breakfast Bacon, Egg & Cheese Bisc     5      410    20       3
## 10 Breakfast Bacon, Egg & Cheese Bisc     6      470    25       4
## # ... with 250 more rows
```

Normally, I wouldn't recommend doing this unless you're sure you know that you don't need the rounded data. You're of course only modifying the data in R only and your original dataset (the one that's saved as a file on your computer) is unmodifed.

However, there is one very handy way that you can use this overwriting trick. If your data isn't formatted the right way (for example, R thinks a particular column is a categorical variable when in reality it's a number), you can modify it using this `mutate` function. In our McDonald's data, we probably want to treat Category as a factor rather than as a character vector. We can do that by creating a "new" column named `Category` that is defined as just the existing `Category` column wrapped up in the `as.factor` function. Because the "new" column has the same name as an existing one, it'll just overwrite it.

```
menu <- menu %>%
    mutate(Category = as.factor(Category)) %>%
    print()
```

```
## # A tibble: 260 x 6
##     Category                         Item    Oz Calories   Fat Sugars
##       <fctr>                        <chr> <dbl>    <int> <dbl>  <int>
##  1 Breakfast                 Egg McMuffin   4.8      300    13      3
##  2 Breakfast            Egg White Delight   4.8      250     8      3
##  3 Breakfast             Sausage McMuffin   3.9      370    23      2
##  4 Breakfast Sausage McMuffin with Eg   5.7      450    28      2
##  5 Breakfast Sausage McMuffin with Eg   5.7      400    23      2
##  6 Breakfast          Steak & Egg McMuffin   6.5      430    23      3
##  7 Breakfast Bacon, Egg & Cheese Bisc   5.3      460    26      3
##  8 Breakfast Bacon, Egg & Cheese Bisc   5.8      520    30      4
##  9 Breakfast Bacon, Egg & Cheese Bisc   5.4      410    20      3
## 10 Breakfast Bacon, Egg & Cheese Bisc   5.9      470    25      4
## # ... with 250 more rows
```

Comparing the output to the previous one, we see that on the second line, right underneath the `Catefory` column header, it has changed from `<chr>` to `<fctr>`. Note that we'll be using this `Category` column as a factor in the next section, so I'm overwriting the `menu` dataframe entirely with this modified version. In other words, that top line is `menu <- menu %>%` rather than just `menu %>%`. The less than sign followed by the dash (which looks like a leftward-pointing arrow) is the *assignment* operator and is what you use to create new variable names in R. Up until now, I've only made temporary changes to the menu data, but because I want to keep this one for this workshop, I'm going to overwrite it. Because assigning data to a variable doesn't automatically print it out for me, I then pipe it to the `print` function. In fact, pretty much all of the tidyverse commands in my code all end in the `print` function so I can make sure things look good every step of the way.

So with just a couple of functions, namely `mutate` and `select`, we can easily add, remove, reorder, and change the columns in our dataset. This modified dataset can then be piped into additional functions that'll clean up your data even more. In the next section we'll see what other kinds of changes we can do to your data.

4 Cleaning your data

In the previous section, we've seen how to make relatively large changes to your dataframe, changes that involve entire columns at the very least. In this section, we'll be able to zoom in a little bit and make small tweaks to individual columns or sometimes just a few rows within a single column. Specifically, we'll look at how to rename columns, rename specific factors within a column, and then how to filter your data based on what's in the columns.

In base R, these are the kinds of things that are a lot harder than I feel like they should be. In fact, I used to do most of this stuff in Excel because it was just too annoying to do in R. With tidyverse, it's pretty easy to make these changes, so now I hardly work in Excel at all to be honest. Let's start with renaming columns.

4.1 Renaming columns

If you want to rename a column in `tidyverse`, you can just use the `rename` function. Inside of it, you put the new name (without quotes), an equals sign, and then the old name in quotes.

```
menu %>%
    rename(name_of_food = "Item")

## # A tibble: 260 x 6
##     Category             name_of_food    Oz Calories   Fat Sugars
##       <fctr>                    <chr> <dbl>    <int> <dbl>  <int>
##  1 Breakfast             Egg McMuffin   4.8      300    13      3
##  2 Breakfast        Egg White Delight   4.8      250     8      3
##  3 Breakfast         Sausage McMuffin   3.9      370    23      2
##  4 Breakfast Sausage McMuffin with Eg   5.7      450    28      2
##  5 Breakfast Sausage McMuffin with Eg   5.7      400    23      2
##  6 Breakfast     Steak & Egg McMuffin   6.5      430    23      3
##  7 Breakfast Bacon, Egg & Cheese Bisc   5.3      460    26      3
##  8 Breakfast Bacon, Egg & Cheese Bisc   5.8      520    30      4
##  9 Breakfast Bacon, Egg & Cheese Bisc   5.4      410    20      3
## 10 Breakfast Bacon, Egg & Cheese Bisc   5.9      470    25      4
## # ... with 250 more rows
```

If you want to rename multiple columns, just do the same thing but with commas between each pair:

```
menu %>%
    rename(name_of_food = "Item",
           group = "Category",
           serving_size = "Oz",
           Sugar = "Sugars")
```

```
## # A tibble: 260 x 6
##        group             name_of_food serving_size Calories   Fat Sugar
##       <fctr>                    <chr>        <dbl>    <int> <dbl> <int>
##  1 Breakfast             Egg McMuffin          4.8      300    13     3
##  2 Breakfast        Egg White Delight          4.8      250     8     3
##  3 Breakfast         Sausage McMuffin          3.9      370    23     2
##  4 Breakfast Sausage McMuffin with Eg          5.7      450    28     2
##  5 Breakfast Sausage McMuffin with Eg          5.7      400    23     2
##  6 Breakfast      Steak & Egg McMuffin          6.5      430    23     3
##  7 Breakfast Bacon, Egg & Cheese Bisc          5.3      460    26     3
##  8 Breakfast Bacon, Egg & Cheese Bisc          5.8      520    30     4
##  9 Breakfast Bacon, Egg & Cheese Bisc          5.4      410    20     3
## 10 Breakfast Bacon, Egg & Cheese Bisc          5.9      470    25     4
## # ... with 250 more rows
```

It's really that simple. The code is clean, consise, easy to read, and easy and type. Just as pretty much all tidyverse functions are.

To emphasize how handy this is, here's how I would have done the samething in base R:

```
menu$name_of_food <- menu$Item
menu$group        <- menu$Category
menu$serving_size <- menu$Oz
menu$Sugar        <- menu$Sugars
menu <- menu[c(8,7,9,4,5,10)]
```

First, I have to run a separate line of code for each column and it's not clear that I'm essentially doing the same thing each time. Then, I need to fix the order of the columns again to get it back to the original order, which like I mentioned before, is error-prone and hard to read. Plus I had to type the name of my dataframe, `menu`, *ten times*! If I had had a longer name (like `mcdonalds_menu_items`), it would have been crazy. Also, if I wanted to apply this to a different dataset, I'd have to copy and paste the new name ten time, where as with `rename`, I just need to do it once. It's crazy how much easier tidyverse makes things!

4.2 Releveling factors

Renaming columns is one thing, but sometimes you'll need to change the data itself. With numbers, it's not too hard because calculations are relatively straightforward, but with categorical data (or basically anything with text), it gets tricky (See *R for Data Science* Chapter 15 on "Factors" (http://r4ds.had.co.nz/factors.html)). Let's look at the `Category` column in our McDonalds data.

```
levels(menu$Category)
```

```
## [1] "Beef & Pork"          "Beverages"          "Breakfast"
## [4] "Chicken & Fish"       "Coffee & Tea"       "Desserts"
## [7] "Salads"               "Smoothies & Shakes" "Snacks & Sides"
```

So we have nine different categories. Looking through them, perhaps you want to rename "Beef & Pork" to "Red Meat" and "Chicken & Fish" into "White Meat". Do to this we'll use a function in the `forcats` package, which, like `readxl` is part of the tidyverse but is not automatically installed when you load the `tidyverse` package. The `forcats` package is full of functions that work well for categorical data.

```
library(forcats)
```

After you've got it installed and loaded, use the `fct_recode`, which has similar syntax to `rename` to change certain values. Because we're using `mutate`, technically what we're doing is creating a "new"" column called `Category` (thus overwriting it). The values of this column is what was previously in the `Category` column only we're changing "Beef & Pork" to "Red Meat".

```
menu_temp <- menu %>%
    mutate(Category = fct_recode(Category, "Red Meat" = "Beef & Pork"))
levels(menu_temp$Category)

## [1] "Red Meat"           "Beverages"          "Breakfast"
## [4] "Chicken & Fish"     "Coffee & Tea"       "Desserts"
## [7] "Salads"             "Smoothies & Shakes" "Snacks & Sides"
```

Just like `rename`, you can do multiple changes in the same function:

```
menu_temp <- menu %>%
    mutate(Category = fct_recode(Category,
                                 "Red Meat" = "Beef & Pork",
                                 "White Meat" = "Chicken & Fish"))
levels(menu_temp$Category)

## [1] "Red Meat"           "Beverages"          "Breakfast"
## [4] "White Meat"         "Coffee & Tea"       "Desserts"
## [7] "Salads"             "Smoothies & Shakes" "Snacks & Sides"
```

With this function, it's easy to collapse factors down into fewer categories, if that's what you need to do. Here, we're collapsing "Beef & Pork" and "Chicken & Fish" into a "Meats" category, we'll add "Coffee & Tea" to a Beverages category, and we'll collapse "Desserts" and "Smoothies & Shakes" into "Sweets".

```
menu_temp <- menu %>%
    mutate(Category = fct_recode(Category,
                                 "Meats" = "Beef & Pork",
                                 "Meats" = "Chicken & Fish",
                                 "Beverages" = "Coffee & Tea",
                                 "Sweets" = "Desserts",
                                 "Sweets" = "Smoothies & Shakes"))
levels(menu_temp$Category)

## [1] "Meats"           "Beverages"       "Breakfast"       "Sweets"
## [5] "Salads"          "Snacks & Sides"
```

In base R, this is a lot more complicated: you have to first allow the possibility for that column to contain this new value (after changing it from a character vector to a factor), and then you need to programatically find all the matching categories and change them one at a time. Several of these steps are not intuitive and can cause great frustration to those who aren't super comfortable with R.

```
# Turn it into a factor and allow for the new categories
menu$Category <- factor(menu$Category,
                        levels=c(unique(menu$Category), "Meats", "Sweets
"))
# Change the values
menu[menu$Category %in% c("Beef & Pork", "Chicken & Fish"),]$Category <- "Mea
ts"
menu[menu$Category == "Coffee & Tea",]$Category <- "Beverages"
menu[menu$Category %in% c("Desserts", "Smoothies & Shakes"),]$Category <- "Sw
eets"
# Remove the old levels
menu <- droplevels(menu)
```

Notice how many times I had to type `menu` and how many times I had to type `Category`. I counted 11 and 8 respectively, not to mention typing `"Meats"` and `"Sweets"` twice. So much repetition obfuscates what's actually going on. (You could probably cut this down using the `with` function, but it seems to me like it complicates the code more rather than simplifying it.) With the `fct_recode` function, you save a lot of typing and frustration because it takes care of all that for you in a way that's so much easier in every way.

In fact, we can even make this even easier for you. If you're just collapsing a couple categories into one (combining the two different meat categories into one), then it's not too bad to type "Meats" twice in the tidyverse version. But if you have 30 different categories you want to collapse down, it'll get tedious to type "Meats" over and over. For this reason, there's the `fct_collapse` function (also from the `forcats` library), which will make it a bit easier to use. The syntax is pretty self-explanatory in this example:

```
menu_temp <- menu %>%
    mutate(Category = fct_collapse(Category,
                                   "Meats"     = c("Beef & Pork", "Chicken & Fi
sh"),
                                   "Beverages" = "Coffee & Tea",
                                   "Sweets"    = c("Desserts", "Smoothies & Sha
kes")))
levels(menu_temp$Category)

## [1] "Meats"          "Beverages"      "Breakfast"      "Sweets"
## [5] "Salads"         "Snacks & Sides"
```

As you can see, with `fct_collapse`, you can just pass a list of all the old values instead of listing them one at a time. It also works just fine with only one item ("Beverages", above), so you'd probably be okay exclusively using `fct_collapse`.

4.3 Filtering

Finally, it's important to know how to filter your data based on these columns. Admittedly, the base R function `subset` is just as succinct as the tidyverse equivalent (`filter`), so here they are for comparison.

```
# Base R version
menu_temp[menu_temp$Category == "Meats",]

# Base R version using subset()
subset(menu_temp, Category == "Meats")

# Tidyverse version
menu_temp %>%
    filter(Category == "Meats")

## # A tibble: 42 x 6
##    Category                     Item    Oz Calories   Fat Sugars
##      <fctr>                    <chr> <dbl>    <int> <dbl>  <int>
##  1    Meats                  Big Mac   7.4      530    27      9
##  2    Meats Quarter Pounder with Che   7.1      520    26     10
##  3    Meats Quarter Pounder with Bac   8.0      600    29     12
##  4    Meats Quarter Pounder with Bac   8.3      610    31     10
##  5    Meats   Quarter Pounder Deluxe   8.6      540    27      9
##  6    Meats Double Quarter Pounder w  10.0      750    43     10
##  7    Meats                Hamburger   3.5      240     8      6
##  8    Meats             Cheeseburger   4.0      290    11      7
##  9    Meats      Double Cheeseburger   5.7      430    21      7
## 10    Meats   Bacon Clubhouse Burger   9.5      720    40     14
## # ... with 32 more rows
```

Either way, it's pretty simple. Except for the first one, which is longer and harder to read, all you need to do is type the column you want to filter by, some sort of operator (==, <, >=, !=, etc.) and then whatever value you want to subset by, in quotes. The above code selects only the menu items that are in the newly created Meat category.

You can add additional filters by just putting a comma between them. The following code filters out everything but the meats that are also greater than or equal to 11 ounces.

```
menu_temp %>%
    filter(Category == "Meats",
           Oz >= 11)

## # A tibble: 5 x 6
##   Category                     Item    Oz Calories   Fat Sugars
##     <fctr>                    <chr> <dbl>    <int> <dbl>  <int>
## 1    Meats Premium McWrap Chicken &  11.1      630    32      7
## 2    Meats Premium McWrap Southwest  11.1      670    33     12
## 3    Meats Premium McWrap Southwest  11.2      520    20     10
## 4    Meats Chicken McNuggets (20 pi  11.4      940    59       0
## 5    Meats Chicken McNuggets (40 pi  22.8     1880   118       1
```

There are a couple more useful tricks when filtering data. What if you want to set a filter so that you can keep things if either condition matches, you can use the | operator, just like in base R. This code finds all Meat or Breakfast items that are greater than or equal to 11 ounces.

```
menu_temp %>%
    filter(Category == "Meats" | Category == "Breakfast",
           Oz >= 11)
```

```
## # A tibble: 9 x 6
##    Category                     Item    Oz Calories   Fat Sugars
##      <fctr>                    <chr> <dbl>    <int> <dbl>  <int>
## 1 Breakfast Big Breakfast with Hotca  14.8     1090    56     17
## 2 Breakfast Big Breakfast with Hotca  15.3     1150    60     17
## 3 Breakfast Big Breakfast with Hotca  14.9      990    46     17
## 4 Breakfast Big Breakfast with Hotca  15.4     1050    50     18
## 5     Meats Premium McWrap Chicken &  11.1      630    32      7
## 6     Meats Premium McWrap Southwest  11.1      670    33     12
## 7     Meats Premium McWrap Southwest  11.2      520    20     10
## 8     Meats Chicken McNuggets (20 pi  11.4      940    59      0
## 9     Meats Chicken McNuggets (40 pi  22.8     1880   118      1
```

Again, with just two, it's not so bad, but if you've got a different dataset with data from all 50 states and you want to filter out 30 of them, you'll have to type `state  ==` over and over. A more elegant solution that is easier to read and easer to type is to use the `%in%` function. This is a base function that essentially lets you use a list where a single string is normally used. It's a little bit unusual-looking since it's got the percent signs, but it works just fine. I can accomplish the exact same thing as the above code but with less typing if I use this:

```
menu_temp %>%
    filter(Category %in% c("Meats", "Breakfast"),
           Oz >= 11)
```

Note that we don't use the == sign anymore, and the list has to be wrapped up in `c()` with each item separated by commas.

What if you want it the opposite way? What if you want to find all food items that are over 11 ounces that are *not* meat and breakfast items?

I believe it was a page on StackOverflow where someone asked that very thing. Several of the responses showed that you can create a new function, let's call it "not in", which can be defined as follows.

```
'%ni%' <- Negate('%in%')
```

When you run this, you've just created a new custom function, `%ni%` that you can now use:

```
menu_temp %>%
    filter(Category %ni% c("Meats", "Breakfast"),
           Oz >= 11)
```

```
## # A tibble: 144 x 6
##     Category                        Item    Oz Calories   Fat Sugars
##       <fctr>                       <chr> <dbl>    <int> <dbl>  <int>
##  1    Salads Premium Southwest Salad  12.3      450    22     12
##  2    Salads Premium Southwest Salad  11.8      290     8     10
##  3 Beverages Coca-Cola Classic (Small  16.0      140     0     39
##  4 Beverages Coca-Cola Classic (Mediu  21.0      200     0     55
##  5 Beverages Coca-Cola Classic (Large  30.0      280     0     76
##  6 Beverages Coca-Cola Classic (Child  12.0      100     0     28
##  7 Beverages         Diet Coke (Small)  16.0        0     0      0
##  8 Beverages        Diet Coke (Medium)  21.0        0     0      0
##  9 Beverages         Diet Coke (Large)  30.0        0     0      0
## 10 Beverages         Diet Coke (Child)  12.0        0     0      0
## # ... with 134 more rows
```

I use this `%ni%` function so much that I sometimes forget it's not a base function. I define it at the top of all my R scripts and use it all the time. It's super handy.

There are many, many more things you can do to clean your data. In this section I've only shown the ones I end up using the most. But hopefully this will take care of a lot of the work that you need to do.

5 Reshaping your data

So far in this workshop, we've seen the basics of tidyverse functions. Some of them make for more elegant code than base R, but others are about teh same. In this section we'll completely reshape how our dataframe is organized: something that as far as I know, is definitely not easy in R.

As I mentioned in my tweet at the beginning of this workshop, I have experienced a lot of frustration trying to reshape my data. By this, I mean going from "tall" to "wide". What do I mean by this?

In this section we'll look at a different dataset. This one is a very small one that contains acoustic measurements of my own speech. Specifically, I said the words *feel*, *fall*, and *fool* and extracted acoustic measurements at 5 points along each vowel's duration. They're saved as csv files on my website, so you can load them directly.

I have the data organized in two different ways. Let's look at each one and see what those differences are. This is what's called a "wide" format:

```
vowels_wide <- read_csv("http://joeystanley.com/data/vowels_wide.csv") %>%
    print()
```

```
## # A tibble: 3 x 6
##    word  t_0.2 t_0.35  t_0.5 t_0.65  t_0.8
##   <chr>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1  fall 515.54 479.96 491.71 459.01 479.61
## 2  feel 320.29 317.59 355.08 483.47 514.34
## 3  fool 296.76 303.73 299.21 299.48 296.11
```

As you can see, we have a table that is 6 columns wide and with only 3 rows. Each row represents one word, with five columns representing those measurements across the five times in which those measurements were taken. Here's the same dataset but in a "tall" format:

```
vowels_tall <- read_csv("http://joeystanley.com/data/vowels_tall.csv") %>%
    print()
```

```
## # A tibble: 15 x 3
##      word  time     Hz
##     <chr> <dbl>  <dbl>
##   1  fall  0.20 515.54
##   2  fall  0.35 479.96
##   3  fall  0.50 491.71
##   4  fall  0.65 459.01
##   5  fall  0.80 479.61
##   6  feel  0.20 320.29
##   7  feel  0.35 317.59
##   8  feel  0.50 355.08
##   9  feel  0.65 483.47
##  10  feel  0.80 514.34
##  11  fool  0.20 296.76
##  12  fool  0.35 303.73
##  13  fool  0.50 299.21
##  14  fool  0.65 299.48
##  15  fool  0.80 296.11
```

Here, there's one row for each acoustic measurement so that each word is spread acros five rows. This dataset is considered "tall" because there are many more rows than columns, giving the impression of a tall rectangle when you view it. (Note that depending on the size of your dataset, you may have many more rows than columns, regardless of how wide you spread out your data. But the principle applies that you can make your table relatively wider and taller when you spread out certain variables.)

Why are there different ways of storing the data? It depends on the research question you're asking. If you're interested in each *word*, you might want to keep the extra-wide format so that each row represents a word. If you're interested in each acoustic measurement individually, you might want to use the tall version.

It's good to know how to convert your data from wide to tall and vice versa. Not only because it lets you look at it in different ways, but because certain kinds of visualizations require one or the other.

5.1 Going from wide to tall

So our first step is going from the wide format to the tall format using `gather`.

The way `gather` works is you need three arguments. First, is the `key`, which is some arbitrary name that'll be given to the new column we're going to create. Next is the `value` argument, which is the name of the column you're creating that contains the information. Finally, you just name the columns that you want to be combined into one.

In this case, we'll create a column called "time" that will contain the time values (0.2, 0.35, 0.5, 0.65, and 0.8). We'll create a new column called "Hz" since these measurements are measured as Hertz (a unit of frequency). And to specify the columns, we could do `t_0.2, t_0.35, t_0.5, t_0.65, t_0.8` or better yet `t_0.2:t_0.8`, but since we just want everything *except* the `word` column, I'll just specify that I *don't* want the `word` column used.

```
vowels_wide %>%
    gather(key = "time", value = "Hz", -word)

## # A tibble: 15 x 3
##     word   time     Hz
##    <chr>  <chr>  <dbl>
##  1  fall  t_0.2 515.54
##  2  feel  t_0.2 320.29
##  3  fool  t_0.2 296.76
##  4  fall t_0.35 479.96
##  5  feel t_0.35 317.59
##  6  fool t_0.35 303.73
##  7  fall  t_0.5 491.71
##  8  feel  t_0.5 355.08
##  9  fool  t_0.5 299.21
## 10  fall t_0.65 459.01
## 11  feel t_0.65 483.47
## 12  fool t_0.65 299.48
## 13  fall  t_0.8 479.61
## 14  feel  t_0.8 514.34
## 15  fool  t_0.8 296.11
```

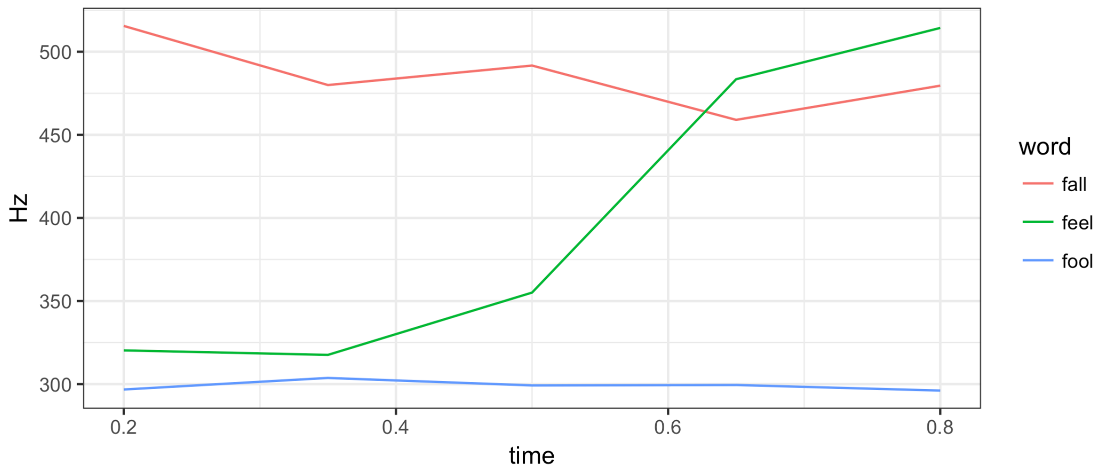
The result is essentially the same as the tall format. The only difference is the order and the fact that the `time` column has that `t_` before each value. Here's the code for doing that, but I'll let you figure out how that's done.

```
vowels_wide %>%
    gather(key = "time", value = "Hz", -word) %>%
    separate(time, into = c("throw_away", "time"), sep="_") %>%
    select(-throw_away)

## # A tibble: 15 x 3
##     word  time     Hz
##   * <chr> <chr>  <dbl>
##   1  fall   0.2 515.54
##  2  feel   0.2 320.29
##  3  fool   0.2 296.76
##  4  fall  0.35 479.96
##  5  feel  0.35 317.59
##  6  fool  0.35 303.73
##  7  fall   0.5 491.71
##  8  feel   0.5 355.08
##  9  fool   0.5 299.21
## 10  fall  0.65 459.01
## 11  feel  0.65 483.47
## 12  fool  0.65 299.48
## 13  fall   0.8 479.61
## 14  feel   0.8 514.34
## 15  fool   0.8 296.11
```

So converting something into a tall format is useful because it allows you to make line plots in `ggplot`, which are otherwise impossible. The way line plots work is that you need one column in your table that represents a unique value for each line you want to draw.

```
vowels_tall %>%
    ggplot(aes(x=time, y=Hz, group=word, color=word)) +
    geom_line() +
    theme_bw()
```



Since we just have three words and five points we want to connect for each one, we can use the `word` column as for the `group` argument. If we had additional lines from any one word or if we had multiple observations of the same word (with 5 points each), we'd have to generate a unique ID column. Ask me for details if you would like do know how.

These line plots are really useful for comparing change across some variable like time. As fantastic as they are, they take some work because your data has to be in a tall format. So converting from wide to tall is a really useful skill to have.

5.2 Going from tall to wide

So what if you have some data that is already tall and you need to convert it into a wide format? You can use `spread`.

With `spread`, the syntax is similar to `gather`. The `key` argument is where you specify which column contains the values that will become the colum *names*. In this case it's the `time` column since we want to spread each point in time into its own column. The `value` argument contains the column in the wide format that contains what you want to fill those cells. In this case, it's the `Hz` column.

```
vowels_tall %>%
    spread(key = time, value = Hz, sep = "_") %>%
    print()

## # A tibble: 3 x 6
##    word time_0.2 time_0.35 time_0.5 time_0.65 time_0.8
## * <chr>    <dbl>     <dbl>    <dbl>     <dbl>    <dbl>
## 1  fall   515.54    479.96   491.71    459.01   479.61
## 2  feel   320.29    317.59   355.08    483.47   514.34
## 3  fool   296.76    303.73   299.21    299.48   296.11
```

The result is a table strikingly similar to the wide table we saw earlier. The addition of the `sep` argument makes it so that the column name "time" is appended at the start of each column followed by an underscore (which I specified). This makes it so that the resulting table doesn't have column names that are numbers, which are tricky to work with.

This section has been a quick run-through of how to convert things from wide to tall. In my experience, it has been frustrating to learn how it's done, but once you get the hang of it, it's incredibly useful and powerful. For more information on spreading and gathering data, see Chapter 12 of *R for Data Science* (http://r4ds.had.co.nz/tidy-data.html).

6 Merging

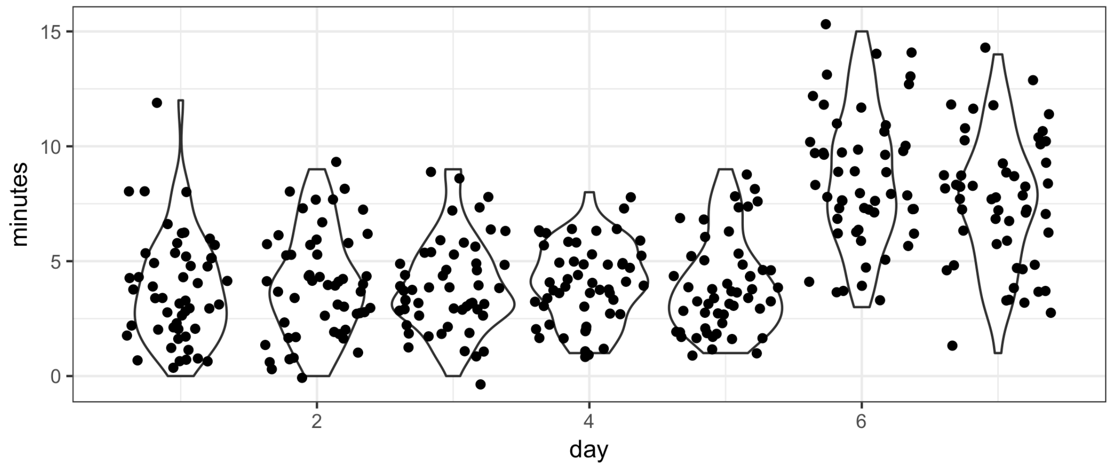
The last thing we'll cover in this workshop is merging. Sometimes you have two datasets that are related in some way and you want to combine them together. For this example, we'll use a simple dataset, the `snooze` one from before which should still be on your computer. As a reminder, this is a simulated dataset that contains information about how many minutes it took someone to get out of bed for an entire year. There are just two columns, the minutes and the day of the week (which is represented by the numbers 1–7).

```
snooze <- read_excel("/Users/joeystanley/Desktop/snoozing.xlsx") %>%
    print()

## # A tibble: 364 x 2
##    minutes   day
##       <dbl> <dbl>
##  1        0     1
##  2        0     3
##  3        0     2
##  4        0     2
##  5        1     5
##  6        1     3
##  7        1     3
##  8        1     5
##  9        1     2
## 10        1     4
## # ... with 354 more rows
```

We can view the general sleep patterns here by creating a simple scatterplot. I'm going to add some *jitter* to the plot which will randomly nudge the points around from left to right a little bit within their column. I'll also add a *violin plot* to see the overall shape of the distribution (which was covered in the previous workshop).

```
ggplot(snooze, aes(x=day, y=minutes, group=day)) +
    geom_violin() +
    geom_jitter() +
    theme_bw()
```



As we can see, during most days, this person slept in maybe 4 or 5 minutes a day. But on days 6 and 7 of each week (presumably Satuday and Sunday), they slept in a little longer, maybe around 8 or so minutes.

Our task is to convert the 1–7 into "Monday", "Tuesday", "Wednesday", etc. How can we do this? One solution is to use the `fct_recode` that was introduced earlier:

```
snooze %>%
    mutate(day = as.factor(day),
        day = fct_recode(day, "Monday" = "1", "Tuesday" = "2", "Wednesday" =
"3",
                          "Thursday" = "4", "Friday" = "5", "Saturday" = "6",
"Sunday" = "7"))

## # A tibble: 364 x 2
##    minutes       day
##      <dbl>    <fctr>
##  1       0    Monday
##  2       0 Wednesday
##  3       0   Tuesday
##  4       0   Tuesday
##  5       1    Friday
##  6       1 Wednesday
##  7       1 Wednesday
##  8       1    Friday
##  9       1   Tuesday
## 10       1  Thursday
## # ... with 354 more rows
```

With just seven possible levels, this works out fine. But imagine a scenario where we have a very long list of unique values, like country codes, and we're trying to convert it into the full country name. That would turn out to be a really big chunk of code and would be a pain to debug if there were typos or something.

Instead, we can create a *lookup table* like we have in Excel. This table will just be 2 columns, the number and the day of the week, and one row for each day.

```
days <- tibble(
    id_number = 1:7,
    spelled = c("Monday", "Tuesday", "Wednesday", "Thursday",
                "Friday", "Saturday", "Sunday")) %>%
    print()

## # A tibble: 7 x 2
##   id_number   spelled
##       <int>     <chr>
## 1         1    Monday
## 2         2   Tuesday
## 3         3 Wednesday
## 4         4  Thursday
## 5         5    Friday
## 6         6  Saturday
## 7         7    Sunday
```

We can merge these two toegher using base R `merge`, or in tidyverse with `left_join`. Here, we supply the two tables we want to merge (the first one is usually piped in), and the crucial part is that we include the `by` argument which contains the name of the columns we want to match up, in quotes. If both have the exact same name, great, but in our case we have to specify both like so:

```
# Base R
merge(snooze, days, by.x="day", by.y="id_number")
```

```
# Tidyverse
snooze %>%
    left_join(days, by=c("day" = "id_number"))
```

```
## # A tibble: 364 x 3
##    minutes   day   spelled
##      <dbl> <dbl>     <chr>
##  1       0     1    Monday
##  2       0     3 Wednesday
##  3       0     2   Tuesday
##  4       0     2   Tuesday
##  5       1     5    Friday
##  6       1     3 Wednesday
##  7       1     3 Wednesday
##  8       1     5    Friday
##  9       1     2   Tuesday
## 10       1     4  Thursday
## # ... with 354 more rows
```

Now we have a new dataset that contains the spelled out versions of the days of the week, which is a lot easier to read.

I've only scratched the surface when it comes to merges and joins. I encourage you to learn more about them (with the help of some useful visualizations) by reading the Chapter 13 of *R for Data Science*, which is called "Relational Data" and can be found here: http://r4ds.had.co.nz/relational-data.html#mutating-joins.

7 Wrapping Up

This workshop has covered a lot of information. As a review, here're the kinds of things we've covered:

1. We saw how to get your data into R using `read_csv` and `read_excel` and how these functions are a bit more efficient than the base R equivalents.
2. We looked at how to tranform your data. First we started off by going over the pipe (`%>%`) and how useful it can be. We then saw how to remove and reorder columns using `select` and to add and modify columns using `mutate`.
3. Next was some tricks on how to clean up your data. Using `rename` you can change the column name; using `relevel` you can change individual levels in your categorical data; andusing `filter` you can quickly make subsets of your data.
4. Then we looked at how to reshape your data from wide to tall or from tall to wide formats.

5. We ended on how to merge two datasets together and in particular how to think of merging as a lookup table in Excel.

This is a lot of information that I've thrown at you at once. But I've provided relevant reading so you can go in and learn more at your own pace. Hopefully this workshop has given you some new skills that will be useful for you and your research, but more importantly I hope it has opened your eyes a little bit to the kinds of ways you can transform your data, which ultimately will let you use your data in different ways and to lead you to new research questions.