Some Data Wrangling Basics

Scott Jackson

October 21, 2019

What's this?

This document is the first set of materials for an informal workshop on mixed-effects models (MEMs) sponsored by the University of Maryland Language Science Center. The goal of this document is to give a kind of "crash course" in several tools that are especially handy for general data wrangling and programming in R, since the workshop participants may not be familiar with all of these.

The intended use of these documents is as follows:

1. Read through the PDF version of this document, and follow along by typing in the example code yourself. I do believe that going through the act of typing out stuff is more useful than simply reading through example code, or even copy-pasting. The act of literally typing out stuff helps with recalling things later. If you get stuck and the code isn't working for you, you can always look at the `.Rmd` version of this document to double-check for typos, etc.
2. Complete the Practice Exercises. These are examples intended to help you practice and consolidate what you are learning. Solutions are given in a separate document, but I encourage you to resist the urge to peek at those until you are well and truly stuck, and have spent some time trying to figure things out. Also, as with most programming tasks, the "solutions" are not the only possible solutions, and may not even be the best solutions I could imagine, so take them with a grain of salt.
3. Attempt the Extension Exercises. These will often ask you to explore either the same tasks with data of your own, or some extension of what is covered in the tutorial materials, or both. This is where you should try to make sure you can apply the material to something that is close to your real-life data analysis goals.
4. When you are done, if you have any comments, submit them via an "issue" on GitHub (Classroom). This will help me track and compile what people are having trouble with, so we can address those things in the in-person workshop sessions. Don't just submit problems; if you do something cool with your own data, tell me!

So with that prologue out of the way, let's get started.

Always look at your data

Initial quick checks

I'm not going to discuss issues of initial reading and cleaning data because that's a bit too much to bite off here, and we're going to start with the assumption that you can read data into R more or less successfully. I'll try to sprinkle tips around throughout the course, but if you run into issues, of course let me know. For the initial examples here, let's use the `sleep` data set, which comes pre-loaded in R, so we don't need to deal with reading from a file. However, let's go ahead and make a copy of the `sleep` data frame in the current workspace (aka Global Environment). When we are in "mucking around" mode, it's sometimes convenient to make a copy of your data so that if you accidentally mess something up, you can easily start with a "fresh" copy, either to start over or to compare what you've done. So by creating an object called `mysleep` with the same value as the `sleep` data frame, no matter what we to to `mysleep`, we can always pull `sleep` back up, without have to re-start our session or anything drastic like that.

```
mysleep <- sleep
```

Once you have data loaded, almost always the very first thing you should do is to look at your data, and start to test your own assumptions about what it should look like. This usually means plotting data, but even before plotting, there are a couple of things that are even quicker, that can help you get started. First is just to make sure it looks like the data has been read in correctly, and you know more or less what the columns are and what's in them. I like to use `head()` for this purpose.

```
head(mysleep)
```

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

This gives us a quick peek at the top of the data frame, sort of like opening the data in a spreadsheet program and just looking at the top of the file. We can see that there are three columns, `extra`, `group`, and `ID`, and they all look numeric, though the latter two look like integers. A next quick thing to do is `summary()`, because this gives us a quick summary of *all* the data in each column, not just the first few rows.

```
summary(mysleep)
```

```
##      extra        group        ID
##  Min.   :-1.600   1:10   1      :2
##  1st Qu.:-0.025   2:10   2      :2
##  Median : 0.950          3      :2
##  Mean   : 1.540          4      :2
##  3rd Qu.: 3.400          5      :2
##  Max.   : 5.500          6      :2
##                          (Other):8
```

This gives us a bit more information, but the general idea is to just use this to check basic assumptions. When calling `summary()` in this way, R gives quartiles (including minimum and maximum values) and a mean for any numeric columns, which we see for the `extra` column. This fits our expectations, because `extra` represents the extra sleep time a participant had in this experiment, which is essentially a continuous measurement. However, the fact that we don't get the quartile/mean summary for `group` and `ID` tells us that despite the values *looking* like numbers (from when we peeked at the `head`), they are in fact stored in the data frame as factors. But again, this is in line with expectations, because the `group` column represents an experimental condition, and the `ID` column represents a nominal subject identifier.

So far so good. And that's the point of checking things out with `head()` and `summary()` – just a quick check that we've got the values we expect in the columns we expect. But especially when you have columns with continuous variables, plotting is the next step to start getting a handle on your data. As the classic Anscombe's Quartet illustrates, numerical summaries such as means, standard deviations, and even fitted model results often just do not tell you critical things that a visualization might easily reveal.

Plotting simple histograms

So let's plot! While we're in the process of getting a descriptive handle on the data, plots of distributions such as histograms are a great place to start. For plotting, we will use the (rightfully) popular `ggplot2` package, which is conveniently also loaded with the `tidyverse` package. As we will be using other functions from `tidyverse` packages, we will go ahead and load that.

```
library(tidyverse)
```

```
## -- Attaching packages ------------------------------------------------- tidyverse 1.2.1 --
```

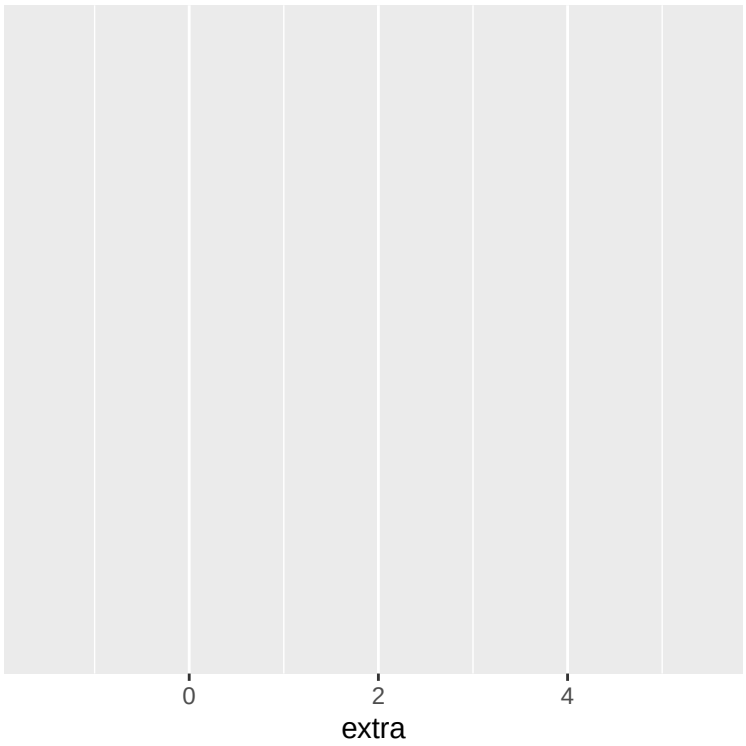
```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ---------------------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Side note: whenever you load a package, it's possible that some of the objects in the package (functions, data) may have the same names as other objects you have loaded. This is called "masking", and it essentially means that if you refer to an object, it will by default use the object with that name that was loaded most recently, and so we say that the other version of the object has been "masked." R helpfully warns you about this every time you load a package, specifically telling you which objects from which packages have been masked. The `tidyverse` package is a bit fancier about how it gives this information, which is what you see in the output when you load this package the first time. For example, it's telling us that if we use the function `filter()`, it's going to default to the function from the `dplyr` package, which is masking the function called `filter()` from the `stats` package. So if we actually want to use the `stats` version, we need to call it with `stats::filter()` instead of just `filter()`. And so on. **End side note**

The `ggplot2` package is extremely powerful, and it has grown over the years to have a *lot* of different functionality and extensions. So again, this is a "crash course" in some basics to get you started, not even a thorough introduction to the package. That said, the basic structure of a plot is as follows. First, there's a call to `ggplot()`, which establishes the basic mapping of data to plot dimensions. For a histogram, we really only have one variable, mapped to the `x` dimension.

```
ggplot(data = mysleep, aes(x = extra))
```

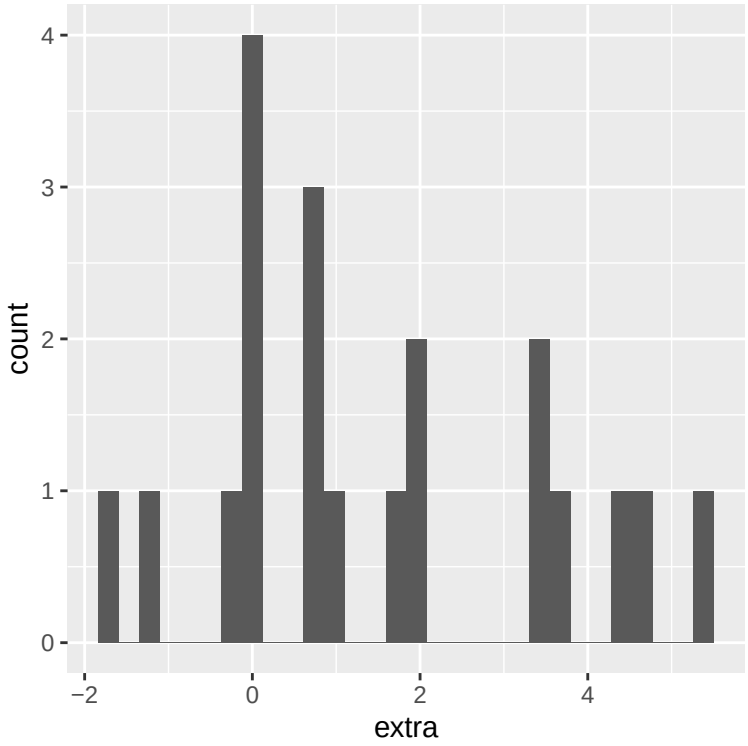


Hmm, just a blank plot, not exactly what we are aiming for. But notice that the x-axis is in fact labeled

with the name of the variable we mapped to it, and the range of values seems appropriate (recall the results of `summary()` above, which already gave us an indication of the data range). The `aes()` function stands for "aesthetics," and this is the crux of the design of `ggplot2`: a plot is a **mapping between data and aesthetics**, which are physical properties of a plot. But in order to actually plot something, we need to not only map variables to aesthetics, but we need to specify what kinds of shapes – or "geoms" – we want to use to display the data. In this case, there is a convenient histogram geom. The code below also illustrates how to be a bit lazier – you don't have to explicitly name the `data` and `x` arguments, since both of those are default arguments (for the first arguments of `ggplot()` and `aes()`, respectively). So the following looks like a typical `ggplot` call:

```
ggplot(mysleep, aes(extra)) + geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



That's better! But we also get a warning about `stat_bin()`. In general, the authors of `tidyverse` functions are pretty liberal with warnings, because they like giving users various heads-ups about default decisions. In this case, the warning is letting us know that a default value for the bin width of the histogram is set in order to produce 30 bins, but that may or may not be a good choice. In the spirit of data exploration, playing around a little with bin widths can actually be very informative. For example, there might be a "spike" of data around a specific value for some reason that hadn't occurred to us, but that may not be visible if the bins are set too wide, because that spike essentially gets smoothed over within the bin. So this bring us to…

Practice Exercise #1

Many of the options for plots are controlled by arguments within geoms. The `geom_histogram()` function has an argument called `binwidth`. Continue looking at histograms of the `extra` variable from `mysleep`, and use the `binwidth` argument inside `geom_histogram` to try out the following bin widths:

1. 2
2. 1.5
3. 1

4. 0.5
5. 0.05

What does the value of bin width correspond to? In other words, what is the scale of the bin width value? Do these different views seem to tell us anything different?

As a reminder, you can look for the "solutions" to this exercise in `data_wrangling_solutions.Rmd`. But also remember that those may not be the only solutions or even necesarily the best ones. But take a peek after you have your own solution, or if you get *really* stuck and are starting to get really frustrated.

Extension Exercise #1

At this point, take a minute to load your own data into R, then do the following:

1. Use `head()` to check that the column names (if there were any in the original data) are what you expect, and that the values visible in the top few rows look right.
2. Use `summary()` to check your assumptions about the ranges and distributions of values, the types of data, amount of missing values, and so on.
3. Pick one or two columns of numeric data, and plot histograms for each of them. Play around with bin widths. Find anything interesting?

Transforming data

After inspecting some distributions, you might decide that some transformations are in order. We will frequently transform our data in the process of fitting models, so it's good to know a few basic transformation techniques.

One fundamental way to transform your data in R is to simply create a vector of values that represents your transformation, and put it into a column in your data. One of the most common transformations we'll use is *standardizing*, by which I mean converting a vector of values to a vector with a mean of 0 and a standard deviation of 1. In base R, we can use the `scale()` function for this. For example:

```
mean(scale(mysleep$extra))
```

```
## [1] -3.729655e-17
```

```
sd(scale(mysleep$extra))
```

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

As we can see, the result of using `scale()` on this variable has a mean of 0 and a standard deviation of 1. Technically, the mean isn't precisely zero, due to the limits of computational precision, but it's extremely close, so for most practical purposes it's close enough to count. While we're on the topic, here's a side note about `scale()`. Check out the following:

```
summary(mysleep$extra)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -1.600  -0.025   0.950   1.540   3.400   5.500
```

```
summary(scale(mysleep$extra))
```

```
##        V1
##  Min.   :-1.5561
##  1st Qu.:-0.7756
##  Median :-0.2924
##  Mean   : 0.0000
```

```
##  3rd Qu.: 0.9217
##  Max.   : 1.9624
```

```
class(mysleep$extra)
```

```
## [1] "numeric"
```

```
class(scale(mysleep$extra))
```

```
## [1] "matrix"
```

```
str(scale(mysleep$extra))
```

```
##  num [1:20, 1] -0.416 -1.556 -0.862 -1.358 -0.813 ...
##  - attr(*, "scaled:center")= num 1.54
##  - attr(*, "scaled:scale")= num 2.02
```

What this tells us is that `scale()` doesn't actually return a vector, even if we give it a vector as input. What this means is that depending on what we're doing, we may need to explicitly convert the result of `scale()` to a vector. In some contexts it gets coerced into a vector structure, but to be safe, we will often want to wrap the result of `scale()` in `as.numeric()`. For example:

```
summary(as.numeric(scale(mysleep$extra)))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.5561 -0.7756 -0.2924  0.0000  0.9217  1.9624
```

```
class(as.numeric(scale(mysleep$extra)))
```

```
## [1] "numeric"
```

A second tip about `scale()`. Recall that standardizing means two things: a mean of 0 and a standard deviation of 1. Depending on what we are trying to do, we might want only one or the other instead of both. You can control that with the `center` and `scale` arguments, which are both `TRUE` by default. When we talk about a *centered* variable, we mean a variable with a mean of 0. Setting `scale = FALSE` but leaving `center = TRUE` will center the variable, but the standard deviation will remain the same as the untransformed variable. And so on.

With all of that out of the way, here's the base R way to make a new transformed (in this case, standardized) column of data:

```
mysleep$extra_std <- as.numeric(scale(mysleep$extra))
summary(mysleep)
```

```
##      extra        group        ID      extra_std
##  Min.   :-1.600   1:10   1      :2   Min.   :-1.5561
##  1st Qu.:-0.025   2:10   2      :2   1st Qu.:-0.7756
##  Median : 0.950          3      :2   Median :-0.2924
##  Mean   : 1.540          4      :2   Mean   : 0.0000
##  3rd Qu.: 3.400          5      :2   3rd Qu.: 0.9217
##  Max.   : 5.500          6      :2   Max.   : 1.9624
##                          (Other):8
```

However, there is also a way to do this in the `tidyverse`, which can be convenient, especially if we would like to put a transformation in a chain of `tidyverse` operations. The following has the same result as above:

```
mysleep <- mysleep %>% mutate(extra_std_tidy = as.numeric(scale(extra)))
```

I'll unpack this a bit, because I also threw in the `%>%` operator, which is a bit of syntactic sugar that is very commonly used in the `tidyverse`.

Many of the `tidyverse` functions, especially those from `dplyr` are designed with a common approach and structure. The idea is that in R we very frequently work with `data.frame` objects, so the `tidyverse` often focuses on manipulating and enhancing these objects. What's happening in this case is that we start with the `data.frame` called `mysleep` and then the `%>%` operator "passes" the value of the object on its left (i.e., `mysleep`) to the *first argument* of the function on its right. In other words, these two lines are doing the same thing:

```
mysleep <- mysleep %>% mutate(extra_std_tidy = as.numeric(scale(extra)))
mysleep <- mutate(mysleep, extra_std_tidy2 = as.numeric(scale(extra)))
identical(mysleep$extra_std_tidy, mysleep$extra_std_tidy2)
```

Side note: `identical()` is a handy function to double-check the equivalence of two objects, but be aware that it is *extremely* conservative. Meaning that if `identical()` returns `TRUE`, you can be confident that the two arguments are in fact equivalent, but if it returns `FALSE`, it may be because of a potentially trivial difference. **End side note**

What's nice about the `%>%` operator is that you can chain steps of evaluation together, from left to right, rather than having to always nest things, from inside out. We'll see many examples of this as we go.

Back to the topic of `mutate()`, it is a nice, convenient way to transform data, and can save a little typing and be a little easier to read. For example, in the above code we can just use `extra` as the argument to `scale()` instead of `mysleep$extra` because it assumes that `extra` comes from the `data.frame` we passed to `mutate()`. Recall that in the above code, the `%>%` operator passes the `mysleep` data frame to the first argument of `mutate()`.

Practice Exercise #2

1. Create a new column in `mysleep` that is the square root of the absolute value of `extra`.
2. Learn and practice how to use `ifelse()` to create or transform a variable conditional on another variable.
    a. Read the help for `ifelse()` to see how it works.
    b. Use `ifelse()` inside `mutate()` to create a new character column called `extra_cat`, which has the value "low" wherever `extra` is less than or equal to 0, "medium" where `extra` is between 0 and 3, and "high where `extra` is 3 or greater.
    c. Imagine the values for `extra` were all recorded wrong for subjects 1, 3, and 7, where those values should be 2 units higher. Use `ifelse()` to create an `extra_corrected` column that corrects these values.

Extension Exercise #2

1. Standardize one or more variables in data of your own. Plot a histogram and use `summary()` to confirm that it's scaled correctly.
2. Use `ifelse()` and `mutate()` to create a categorical "re-coding" of a numeric variable.
3. Create a log-transformed version of a numeric variable in your data. Plot histograms of both the raw variable and the log-transformed variable.

Calculating summary values from your data

Starting with looking at entire distributions with something like histograms is a good idea, because you can discover quirks and issues with your data that you might not notice otherwise. But at some point, we are

often more interested in drawing conclusions using summary statistics of various kinds. For example, in our `mysleep` data frame, we have a treatment condition, and we might want to know whether the `extra` values are higher on average in one condition vs. the other. In other words, we'd like to compute the condition cell means (here, just two, since there are only two groups).

In general, I will use the term *aggregation* to talk about boiling down multiple values to single values (like getting the mean of a vector of numbers). We will want to do this a lot, and not just for pre-defined functions like `mean()`, but for any arbitrary function.

Simple aggregation in the `tidyverse`

There are various ways of doing this, but one of the most convenient is using the `summarize()` function from the `dplyr` package (part of the `tidyverse`). For example, the following:

```
mysleep %>% summarize(extra_mean = mean(extra))
```

```
##   extra_mean
## 1       1.54
```

Here's how `summarize()` works (not to be confused with `summary()`, but also equivalent to `summarise()`, which may be preferred depending on what dialect of English you are happier using). The first argument of `summarize()` is simply the `data.frame` you are working with. (And recall that in the above code, we are using the `%>%` operator to pass the value of `mysleep` to that first argument.) Every argument after that is determined by the user, in the following way. The *name* of the argument is the name of the new column that you are putting the summary values in. The *value* of the argument is the value that you are putting in that column. In the code above, we are creating a column called `extra_mean` (this name is arbitrary, decided by us) and assigning that column the value of the mean of the values in the `extra` column from the `data.frame` we passed to `summarize()`.

One of the nice things about this function is that you can create multiple summary columns at once, and you have enormous flexibility because you can put literally anything in this function, as long as it returns a value that can fit in the resulting `data.frame`. In the simple case, the rule of thumb is that as long as the value you are returning is a single value (i.e., length of 1), it will work.

So we can easily spit out a summary table with the mean, median, and standard deviation as follows (side note: you can use whitespace to make your code a little easier to read, like adding line returns to keep it from it running on too long on a single line):

```
mysleep %>% summarize(extra_mean   = mean(extra),
                      extra_median = median(extra),
                      extra_sd     = sd(extra))
```

```
##   extra_mean extra_median extra_sd
## 1       1.54         0.95  2.01792
```

Practice Exercise #3

Another nice feature is that you can use whatever arguments you like in the summarizing functions, just like they are normally used. For example, you might have missing data in the form of NAs. Run the following code to see what happens:

```
mysleep_missing <- mysleep
mysleep_missing[c(1, 4, 7, 13), "extra"] <- NA
mysleep_missing %>% summarize(extra_mean = mean(extra))
```

Modify the final line in the above code so that the `mean()` function ignores NAs, and will therefore return the mean value of the observed values. (Hint: review the help for `mean()` if you get stuck.)

Extension Exercise #3

Use `summarize` to get summary values from a column in data of your own. Try the following:

1. mean
2. median
3. standard deviation
4. count (i.e., the number of values)
5. the maximum of natural log of the absolute value plus 1 (i.e., take the absolute value, add 1 – which insures there are no zeroes – take the log of that, and get the maximum of those log values)

Grouped aggregation

Getting summary values across your entire `data.frame` is fine, but it is often far more useful to get values broken down by various factors. Returning again to the `mysleep` data, we wanted to know whether the means are different in the two groups. Here's where the `group_by()` function shines.

```
mysleep %>% group_by(group) %>% summarize(extra_mean = mean(extra))
```

```
## # A tibble: 2 x 2
##   group extra_mean
##   <fct>      <dbl>
## 1 1           0.75
## 2 2           2.33
```

Here's how this works. The `group_by()` function (like many `tidyverse` functions) takes a `data.frame` as its first argument, and then following arguments are columns that are used to create a grouped structure. The result is a special structure that is still a `data.frame`, but "under the hood" it has the grouped structure, so that the next function applies to that `data.frame` in a grouped way. In this case, we tell it to group the `mysleep` data by the `group` column (which represents two different treatment conditions), and then when we apply `summarize()` to this grouped data frame, we get back results that have a different row for each value in `group`. That is, we get the mean value of `extra` separated out by group. This is the "cell means" we were going for.

Side note: You may have noticed that the output calls this a "tibble." This is like a cutesy pronunication of the `tbl` class that this object belongs to. Notice the difference:

```
class(mysleep)
```

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

```
class(as_tibble(mysleep))
```

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

Note that a tibble is still a `data.frame`, but that's not all it is. It's a `tbl` and a `tbl_df` as well. Not to go too deep into the semi-object-oriented nature of R, but a lot of functions in R change their behavior depending on the class of the object given as an argument. The `tidyverse` designers have built a lot of functionality around the `tbl` and related classes. But sometimes you really just don't want the fanciness and you want to use a plain 'ol data frame. You can always just end your line of `tidyverse` functions with `as.data.frame()` to do the trick. Notice the difference:

```
mysleep %>% group_by(group) %>% summarize(extra_mean = mean(extra))
```

```
## # A tibble: 2 x 2
##   group extra_mean
##   <fct>      <dbl>
## 1 1           0.75
```

```
## 2 2           2.33
```

```
mysleep %>% group_by(group) %>% summarize(extra_mean = mean(extra)) %>% as.data.frame()
```

```
##   group extra_mean
## 1     1       0.75
## 2     2       2.33
```

We will occasionally run into this issue, and I just wanted to point it out now, in cae you were wondering about the "tibble" output. **End side note**

Plotting interlude

We often want to visualize summaries like cell means with plots rather than tables of numbers. One common visualization is a bar plot. There are a few wrinkles to how bar plots work in `ggplot`, so let's go over those.

First, there is often a temptation to want to create a summary bar plot from the raw data. That is, it seems like it would be nice to feed in your entire data set to `ggplot` and get out a bar plot of cell means. There are some options you can use to do something like this, but I encourage you to stick with the principle of using `gpplot` to visualize the actual numbers in your data frame. If you want to visualize a summary like cell means, then create a new data frame of cell means and plot that. This will make things far more consistent and easier to track and troubleshoot than trying to cram your summary statistics inside of a `ggplot` call.

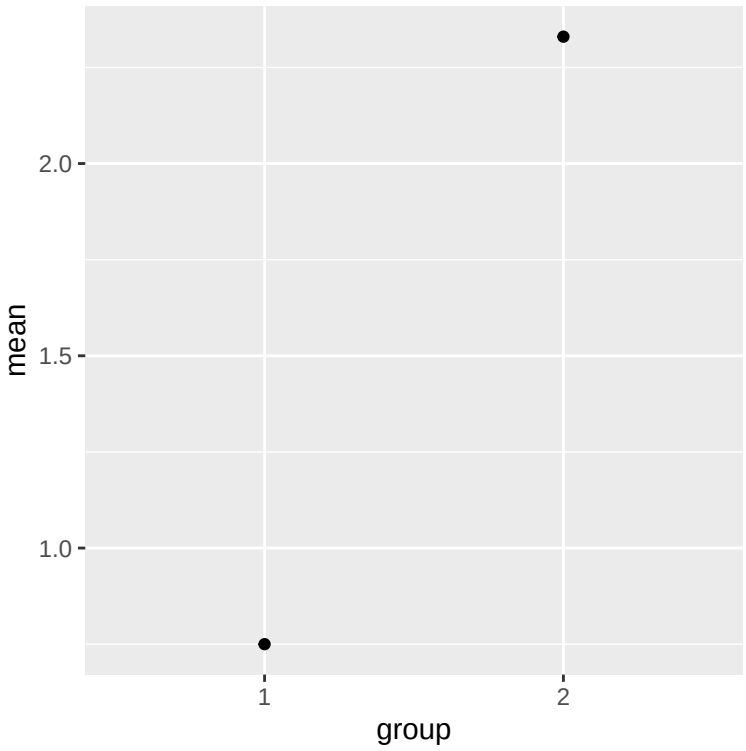
So that's what we'll do here. First, let's create the summary table as above, and print it out, just to be clear about what we're doing.

```
sleep_cellmeans <- mysleep %>% group_by(group) %>% summarize(mean = mean(extra))
sleep_cellmeans
```

```
## # A tibble: 2 x 2
##   group  mean
##   <fct> <dbl>
## 1 1      0.75
## 2 2      2.33
```

Now, we can plot those values directly. Before using `geom_bar`, let's use the simpler `geom_point`.

```
ggplot(sleep_cellmeans, aes(group, mean)) + geom_point()
```



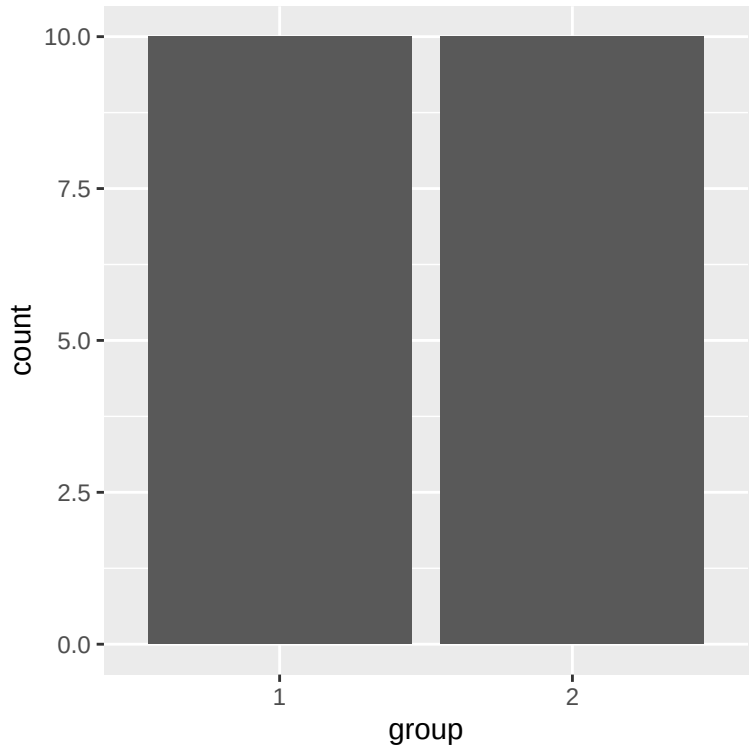
This is straightforward because we are telling `ggplot` to map the x-axis to `group` and the y-axis to `mean`, which was the name of the column we gave the cell means.

If we try to do the same with `geom_bar`, we run into a little trouble.

```
ggplot(sleep_cellmeans, aes(group, mean)) + geom_bar()
```

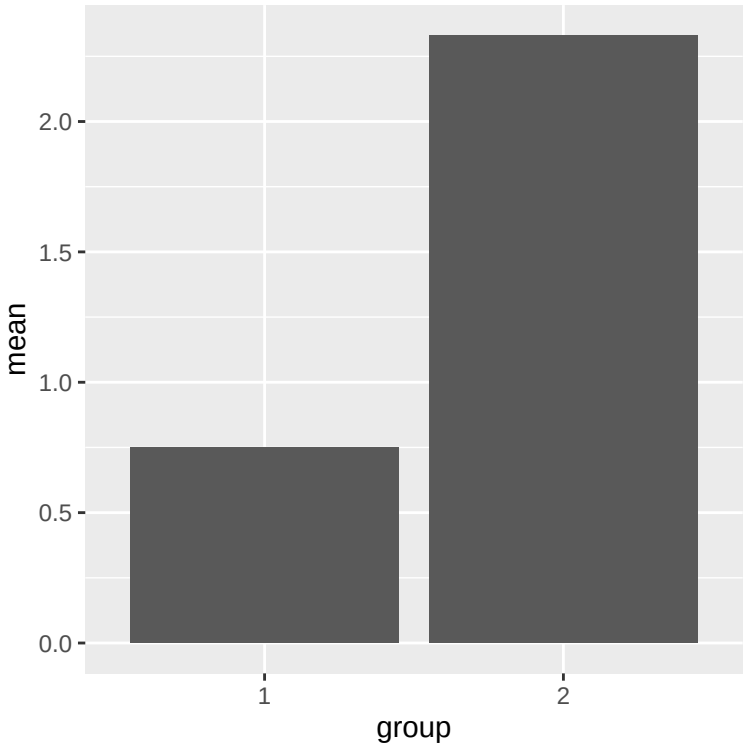
If you try the above, you will get an error! This is because `geom_bar` makes some different assumptions about how to map the data to the geom. Ultimately this is because the authors of the `ggplot2` package have fairly strong opinions about how bar plots should and shouldn't be used. So the default use of `geom_bar` is much like a histogram, where you only supply an `x`, and the plot generates bars for every value of x, where the height of the bar corresponds to the number of observations of that value. This is very handy when you have, for example, a response variable with only a few possible values, like a Likert scale response. For illustration, we can use the `group` variable itself in the raw `sleep` data:

```
ggplot(sleep, aes(group)) + geom_bar()
```



This is obviously a boring plot, because it's just telling us we have 10 values from group 1 and 10 from group 2, but you can see how this works. If we instead want bar plots that correspond to the actual numbers in our data and not counts of numbers, we need to use the `stat` argument with the value of `"identity"`. This basically tells `geom_bar` to "just plot the actual numbers."

```
ggplot(sleep_cellmeans, aes(group, mean)) + geom_bar(stat = "identity")
```



Now, notice a few differences between the bar plot and the first `geom_point` plot above. They are both essentially visualizing just two numbers, 0.75 and 2.33, the cell means of the two groups. However, one difference (beyond the shapes plotted) is the range of the y-axis. The y-axis in `geom_bar` always includes 0. The reason for this is connected to the rationale for why one might want to use a bar plot in the first place. Namely, you should only use a bar plot when part of what you are trying to communicate is the relative proportion of different quantities. For example, the difference between 2 and 4 is the same as the difference between 200 and 202, but the relative proportions are very different.

So here's the advice: when you are thinking about how to plot a summary like cell means, as yourself whether you care more about the difference between the two, or the relative proportions. For example, in this case of extra hours of sleep, being able to see that one condition is about 3 times the other may be meaningful. But if you have a behavioral experiment where you are looking at the difference between, say, 990 and 1000 milliseconds, you are probably more interested in looking at the magnitude of differences rather than the magnitude of proportions. The rule of thumb is that if you want to show the magnitude of differences, use points; if you want to show proportions, use bars.

Practice Exercise #4

1. Plot the mean value of `extra` for each subject ID in `mysleep`
   a. …using `geom_point()`
   b. …using `geom_bar()`
2. Try adding error bars with `geom_errorbar()` based on the standard deviations.
   a. Compute a summary table of both means and standard deviations, separately for each `group` in `mysleep`
   b. Specify the arguments of `geom_errorbar()` so that the error bars extend 1 standard deviation above and below the mean
   - `geom_errorbar()` has additional aesthetics (i.e., arguments inside `aes()` inside `geom_errorbar()`)
     - `ymin`: the lowest point of the error bar
     - `ymax`: the highest point of the error bar

   c. You may want to make the fill color of the bars lighter, in order to make the lower part of the error bar more visible. Try experimenting with a `fill` argument inside `geom_bar()`.
3. Now try the "point" version of the above, using `geom_pointrange()` to replace both `geom_bar()` and `geom_errorbar()`
   a. Use the same summary table of means and standard deviations
   b. Like `geom_errorbar()`, `geom_pointrange()` uses `ymin` and `ymax` arguments
   c. Any apparent advantages/disadvantages over the bars + error bars visualization?

Extension Exercise #4

Return to the summary table you created for Extension Exercise #3 (cell means of some quantity from your own data).

1. Re-run this table, but include standard deviations as well as means.
2. Does it make sense to use a bar plot or a dot plot for these summary values?

a. Make a bar plot with error bars
b. Make a dot + line plot with `geom_pointrange`

Functions and conditionals

One of the most powerful and useful things about R is not only using the functions that other people have written, but being able to write your own functions. There are a lot of benefits to doing this, the main ones

being that it makes your code much more re-usable, ueful, and easy to troubleshoot. Basically any time you find yourself copying and pasting code to change just a few little things, you should consider writing a function. Once you get the hang of this, it will make your R life much easier.

I already alluded to one example of this above, when I pointed out that the `summarize()` function can use any arbitrary function. That is, you don't need to stick to just plain `mean()`, `sd()` and so on, you can make any function you want and use that instead.

As an example, let's create a very simple function to take the geometric mean of a variable. A geometric mean is basically just the mean of the log-transformed variable, usually transformed back to the original scale. It is extremely easy to make functions in R, as they all follow the following format:

```
function_name <- function(arg1, arg2, arg3, ...) {
  # whatever code you want here
  return(value_you_want_returned)
}
```

Here's how this works. The `function_name` is defined by you, and this is just creating an object in your workspace, just like any other time you use the `<-` operator, but instead of creating a vector or data frame, you're making a function you can use later. The `function()` function is what actually turns your code into a function. Inside the `function()` call, you list whatever arguments you want your function to use, and name them whatever you want (not necessarily `arg1` and so on). Then you start a chunk of code with a curly brace, and end it with a closing curly brace later. Then you put *whatever code you want* inside those curly braces. That's the "body" of your function, and it can be as simple or as complex as you need, even defining other functions if you want.

But what does a function *do*? There are two answers. The basic idea comes from the mathematical definition of a function: you put stuff into a function and you get a single value out. That is, whatever else your function does, it must return a single value. You can be explicit about this and use the `return()` function inside your function, to make it obvious exactly what it's returning. Or you can be lazy, and R will basically return whatever the last value in your function code is. Or if there's no good candidate for a default return value, it will return NULL. Sometimes the value returned by a function is an extremely complex and/or large object, like the value returned by a `ggplot()` call or an `lmer()` call. But it's still a single object, a single returned value.

But that's not all. The second answer is that functions in R can also have "side effects." Because you can run whatever code you want in your function, you can make your function literally do anything, in addition to whatever value it returns. Base R plotting is an example of this, because the value returned from a plotting function might have some information or might be NULL, but the "side effect" is to draw the actual plot. So side effects can be extremely useful, or they can be annoying or even dangerous, if you get really sloppy (or intentionally evil – don't do that!). The point is that this flexibility is built into R.

So let's make our geometric mean function. Here's an initial version:

```
geomean <- function(x) {
  exp(mean(log(x)))
}
```

This is the laziest possible version (well, almost – we could also put the whole thing on one line and leave out the curly braces if we wanted). We don't have an explicit `return()` statement, but because the single line of code in our function returns the value we want (or think we want), this value is what gets returned when we run our function. Let's try this function on the `extra` variable in the `sleep` column.

```
geomean(sleep$extra)
```

```
## Warning in log(x): NaNs produced
```

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

And here we see what can often happen when you try to cram too much into a function at once. One of the good design principles of a function is that you should write and test them *incrementally*. So before we make a function, let's just run some code to see if we can get what we want.

```
x <- sleep$extra
log(x)
```

```
## Warning in log(x): NaNs produced
```

```
##  [1] -0.35667494         NaN         NaN         NaN         NaN
##  [6]  1.22377543  1.30833282 -0.22314355        -Inf  0.69314718
## [11]  0.64185389 -0.22314355  0.09531018 -2.30258509         NaN
## [16]  1.48160454  1.70474809  0.47000363  1.52605630  1.22377543
```

```
x
```

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

Aha! So because we have a zero and some negative numbers, `log()` runs into some issues. And of course if we try to take the mean of a vector with `NaN` or `Inf` values, we probably won't get a satisfying answer. So let's write a little code that handles this. In this case, just for illustration, let's perform two checks. First, if there are any zeros, let's add a little random noise to the whole variable, which will fudge any zeros to be non-zero. Second, if there are any negative numbers, let's take the absolute value. We can do this with conditional `if()` statements.

```
x <- sleep$extra
if(any(x == 0)) {
  x <- jitter(x)
}
x
```

```
##  [1]  0.69644459 -1.59395428 -0.19996047 -1.18993572 -0.10515080
##  [6]  3.40755257  3.71914379  0.78931282  0.01425149  1.99542484
## [11]  1.88712222  0.78221446  1.08512370  0.08706999 -0.08462278
## [16]  4.41107606  5.51490994  1.60060410  4.61274458  3.41089605
```

```
any(x == 0)
```

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

I've illustrated a couple of tricks here. First, the use of `if()`. This function checks its argument for whether it is `TRUE` or `FALSE`, and if it's `TRUE`, it runs all of the code in the following curly braces. If it's `FALSE`, then it just skips that code.

The second trick is that `if()` only expects a single value, and if you give it, say, a vector of values, then it will only check the first member of the vector for whether it is `TRUE` or `FALSE`. And in fact R will warn you when this happens. Usually that means you're giving something to `if()` that you didn't intend. What we want to do is look for any instances of a zero value in a vector. If we just say `x == 0`, we compare every value in `x` to zero, and get back a vector of logicals (try it out to see). The convenient `any()` function simply takes a vector of logicals and checks whether there is a `TRUE` value somewhere in the vector. If so, it returns a single `TRUE` value, if not, it returns a single `FALSE` value.

So let's use the same strategy to check for negatives.

```
x <- sleep$extra
if(any(x < 0)) {
  x <- abs(x)
}
x
```

```
##  [1] 0.7 1.6 0.2 1.2 0.1 3.4 3.7 0.8 0.0 2.0 1.9 0.8 1.1 0.1 0.1 4.4 5.5
## [18] 1.6 4.6 3.4
```

So far so good! Now, you might wonder why even check the conditional in this particular case, because if they're all positive, taking the absolute value won't change anything, so why not just do the absolute value every time to be safe and not bother with the `if()`? And that would be a very clever thing to wonder, nice job. In this case, you're probably right, it's not a big deal. But in general, the operation inside the `if()` could be computationally intensive, and if you can skip it, you might want to. That is, there's not much computation cost to checking this particular statement, and it might be worth it down the line, if we end up putting more code in the conditional block, or if we would like to control whether we take absolute values or not. This just illustrates the thought process of designing your functions and the choices you make in that process.

Practice Exercise #5

Previously we used standard deviations for error bars. This is not necessarily bad, but more typically we depend on standard errors. The standard error of the mean of a variable x with N values and standard deviation σ is typically defined as:

$SE = \frac{\sigma}{\sqrt{N}}$

In English, this means to take the standard deviation of the variable and divide it by the square root of the number of values in the variable.

1. Create a function called `std.err()` that takes a vector as an argument and returns the standard error of the mean of that vector.
2. Revisit the `sleep` data again (again saving a copy called `mysleep` in the workspace) and create a `sleep_cellmeans` summary table that has the mean and standard error for both groups.
3. 95% confidence intervals are usually associated with $\pm$ 1.96 standard errors (and some people round this up to 2). Using either `geom_bar() + geom_errorbar()` or `geom_pointrange()`, plot the means and 95% confidence intervals around those means using the table in #2.

Extension Exercise #5

1. Go through the steps for making a plot of means and 95% confidence intervals for some variable in your own data.
   a. Define a function (or re-use your previously defined function) to conveniently compute standard errors of a mean
   b. Use `group_by()` and `summarize()` to create a summary table of cell means and associated standard errors.
   c. Plot these with `ggplot()` using some combination of `geom_bar()`, `geom_errorbar()`, `geom_point`, `geom_pointrange()`, and/or `geom_linerange()`.
2. Go through the same process, but using two grouping variables.
   - Previously we mapped the grouping variable to the `x` aesthetic. When you have two grouping variables, you will need another aesthetic, to map the second variable to.
   - Color may be a good choice. If you are using a geom with "area", like `geom_bar()`, you should try mapping the `fill` aesthetic to a grouping variable. Otherwise trying mapping the `color` aesthetic (or `colour` if you like) to the grouping variable.
   - See if you can figure out how to get error bars to also behave appropriately.
   - Play around with which variable you're mapping to `x` and which to `fill`/`color`. Does one make more or less sense than another?

Reshaping

Up to this point, we have depended on our data being structured in a particular way. For example, `ggplot` uses a mapping between variables and aesthetics, which is another way of saying that if you want a dimension of your plot to depend on some dimension of your data, you need to put that dimension in its own column! But other times, we might want our data to be in a different shape, to accommodate different functions we might run. Because of this, the ability to easily reshape your data is extremely useful.

A little history

We will again turn to the `tidyverse`. As a bit of an aside, the `tidyverse` did not emerge fully formed, like some kind of R Athena. The primary author, Hadley Wickham, essentially started with two "killer apps" for R. One was `ggplot`, the other was a package called `reshape`. We have since moved on to `ggplot2`, and there is also a `reshape2`, but now there is also a package called `tidyr` that implements the newest and improvededest (sp?) versions of Hadley's vision of how to implement reshaping data in R.

The point here is that I will take you through the "bleeding edge" functions here, but in the wild of the internet, you may find many other alternatives, including `melt` vs. `cast`, `gather` vs. `spread`, etc. The good thing is that Hadley & co. are always searching for a better way to do things, but the bad thing is that they sometimes leave behind the old ways, even when they worked pretty well already.

`pivot_wider()`

The newest versions of reshaping functions in the `tidyverse` start with `pivot`. The idea here is that in most cases, when we talk about *reshaping* data instead of *aggregating* data, we are really just changing the shape of how values are arranged in a data frame, and not doing anything to those values themselves. (**Side note**: it's apparently possible to use `pivot_wider()` to also aggregate, like `cast()` or `dcast()` of old, but we'll leave that aside for now.)

Let's return to our `sleep` data. By now you are familiar with the data structure, and have probably noticed that each participant has data in both experimental conditions. Therefore, it might be of interest to get difference scores between the conditions, computed separately for each participant. In order to do that, we would essentially need to take the data in the `extra` column and put it into two different columns, one for group 1 and one for group 2. This is called making the data "wider," because we are essentially making more columns out of the columns that we have, so we use `pivot_wider`. Easy, right?

Sidebar on factors

Before we do this, one more sidetrack having to do with *factors* in R. First, the reason we are making this sidetrack: we want two new columns to replace the `extra` column, but right now our groups are called `1` and `2`, and it turns out that numbers are extremely annoying to use as column names in R. So we would like to change the values to `group1` and `group2`. This has the added benefit of making it much more obvious that these data represent distinct conditions, not actual integers.

We have already covered some tools you could use to do this, like `mutate()`, `ifelse()`, and so on. But there's another way that also gives me the opportunity to tell you an important fact: **factors in R behave like integers with labels**. That is, underlyingly, factors in R are represented by a set of factor levels, which are integers that start at 1 and go up to the number of levels, but these levels also have labels. And it turns out that you can manipulate these separately. Sometimes this can be convenient, but sometimes it can give you unexpected results. The point here is that if you can remember the mantra of **factors in R are integers with labels**, you will often steer yourself clear of a lot of frustration when working with factors.

In this particular case, we can access the levels of `group` directly:

```
levels(mysleep$group)
```

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

Note that the "1" and "2" are in quotes, indicating that the factor level *labels* are in fact strings, even though they look like numbers. It turns out that we can assign values to these labels directly, using the "assignment" version of that function. For example:

```
mysleep <- sleep
print(mysleep)
```

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

```
levels(mysleep$group) <- c("group1", "group2")
print(mysleep)
```

```
##    extra  group ID
## 1    0.7 group1  1
## 2   -1.6 group1  2
## 3   -0.2 group1  3
## 4   -1.2 group1  4
## 5   -0.1 group1  5
## 6    3.4 group1  6
## 7    3.7 group1  7
## 8    0.8 group1  8
## 9    0.0 group1  9
## 10   2.0 group1 10
## 11   1.9 group2  1
## 12   0.8 group2  2
## 13   1.1 group2  3
## 14   0.1 group2  4
## 15  -0.1 group2  5
## 16   4.4 group2  6
## 17   5.5 group2  7
## 18   1.6 group2  8
## 19   4.6 group2  9
```

```
## 20   3.4 group2 10
```

Note that this is extremely powerful and you need to be careful that you're doing this correctly, because if you did something like the following:

```
mysleep$group_wrong <- mysleep$group
levels(mysleep$group_wrong) <- c("group2", "group1")
print(mysleep)
```

```
##    extra  group ID group_wrong
## 1    0.7 group1  1      group2
## 2   -1.6 group1  2      group2
## 3   -0.2 group1  3      group2
## 4   -1.2 group1  4      group2
## 5   -0.1 group1  5      group2
## 6    3.4 group1  6      group2
## 7    3.7 group1  7      group2
## 8    0.8 group1  8      group2
## 9    0.0 group1  9      group2
## 10   2.0 group1 10      group2
## 11   1.9 group2  1      group1
## 12   0.8 group2  2      group1
## 13   1.1 group2  3      group1
## 14   0.1 group2  4      group1
## 15  -0.1 group2  5      group1
## 16   4.4 group2  6      group1
## 17   5.5 group2  7      group1
## 18   1.6 group2  8      group1
## 19   4.6 group2  9      group1
## 20   3.4 group2 10      group1
```

… you could end up with some very surprising results! But for now, we'll make use of this ability to easily change level labels so that when we make our data wider, it's easier to work with.

```
mysleep <- sleep
levels(mysleep$group) <- c("group1", "group2")
summary(mysleep)
```

```
##      extra           group          ID
##  Min.   :-1.600   group1:10   1      :2
##  1st Qu.:-0.025   group2:10   2      :2
##  Median : 0.950               3      :2
##  Mean   : 1.540               4      :2
##  3rd Qu.: 3.400               5      :2
##  Max.   : 5.500               6      :2
##                               (Other):8
```

Back to `pivot_wider()`

The basic use of `pivot_wider()` is very straightforward. Like many `tidyverse` functions, the first argument is a data frame, so we can "pass" it a data frame with `%>%` just like we did with `mutate()` or `summarize()`. And the idea is that you need to tell it where you're getting the names of your new columns, and where you're getting the values of your new columns. So for our present purposes, we can do the following:

```
mysleep_wide <- pivot_wider(mysleep, names_from = group, values_from = extra)
print(mysleep_wide)
```

```
## # A tibble: 10 x 3
##    ID    group1 group2
##    <fct>  <dbl>  <dbl>
##  1 1        0.7    1.9
##  2 2       -1.6    0.8
##  3 3       -0.2    1.1
##  4 4       -1.2    0.1
##  5 5       -0.1   -0.1
##  6 6        3.4    4.4
##  7 7        3.7    5.5
##  8 8        0.8    1.6
##  9 9        0      4.6
## 10 10       2      3.4
```

Et voilà! I'll just recap this again: in order to use `pivot_wider()`, what you need to do is imagine taking a column in your data and splitting it up across multiple new columns. All you need to tell `pivot_wider()` is the name of the current column that has the values that you want to put in the new columns (`values_from`) and the name of the current column that contains all of the names of the new columns (`names_from`). There are many more complex scenarios and `pivot_wider()` can handle a lot, but this is the basic usage, and it will get you very far.

'pivot_longer()

The opposite of making a data frame "wider" is making it "longer." This is perhaps a more frequent issue, since many data sources are arranged natively in wide format, as this can sometimes cut down on the size of the data in memory or storage. For example, note that when we made the `mysleep` data wider, we could represent it with the same set of 20 numbers, but now the IDs are not repeated twice, and we don't have a column that is a repetition of "group1" and "group2", so it is now more compact in a concrete sense of how many bits are needed to represent the data.

However, compact is not necessarily *convenient*, and we often need to make data longer. The `pivot_longer()` function gives us a nice way to do this. Like with `pivot_wider()`, we pass it a data frame, but now we need to specify three things: the set of columns that we wish to "combine" into a longer column, the name of a new column that will store the names of the columns we are combining, and the name of the column that will store the values we are taking from the columns we are combining. For example, we can make our `mysleep_wide` long again as follows:

```
mysleep_wide %>% pivot_longer(cols = c(group1, group2),
                              names_to = "condition",
                              values_to = "hours")
```

```
## # A tibble: 20 x 3
##    ID    condition hours
##    <fct> <chr>     <dbl>
##  1 1     group1      0.7
##  2 1     group2      1.9
##  3 2     group1     -1.6
##  4 2     group2      0.8
##  5 3     group1     -0.2
##  6 3     group2      1.1
##  7 4     group1     -1.2
##  8 4     group2      0.1
##  9 5     group1     -0.1
## 10 5     group2     -0.1
## 11 6     group1      3.4
```

```
## 12 6     group2      4.4
## 13 7     group1      3.7
## 14 7     group2      5.5
## 15 8     group1      0.8
## 16 8     group2      1.6
## 17 9     group1      0
## 18 9     group2      4.6
## 19 10    group1      2
## 20 10    group2      3.4
```

Notice that when we made it long again we decided on different names than the columns we started with in the original data. We have that flexibility. As an example of a few other handy functions, we can also re-create the original exactly:

```
mysleep
```

```
##    extra  group ID
## 1    0.7 group1  1
## 2   -1.6 group1  2
## 3   -0.2 group1  3
## 4   -1.2 group1  4
## 5   -0.1 group1  5
## 6    3.4 group1  6
## 7    3.7 group1  7
## 8    0.8 group1  8
## 9    0.0 group1  9
## 10   2.0 group1 10
## 11   1.9 group2  1
## 12   0.8 group2  2
## 13   1.1 group2  3
## 14   0.1 group2  4
## 15  -0.1 group2  5
## 16   4.4 group2  6
## 17   5.5 group2  7
## 18   1.6 group2  8
## 19   4.6 group2  9
## 20   3.4 group2 10
```

```
mysleep_wide %>% pivot_longer(cols = c(group1, group2),
                              names_to = "condition",
                              values_to = "hours") %>%
  select(hours, condition, ID) %>%
  arrange(condition, ID) %>% as.data.frame()
```

```
##    hours condition ID
## 1    0.7    group1  1
## 2   -1.6    group1  2
## 3   -0.2    group1  3
## 4   -1.2    group1  4
## 5   -0.1    group1  5
## 6    3.4    group1  6
## 7    3.7    group1  7
## 8    0.8    group1  8
## 9    0.0    group1  9
## 10   2.0    group1 10
## 11   1.9    group2  1
```

```
## 12   0.8    group2  2
## 13   1.1    group2  3
## 14   0.1    group2  4
## 15  -0.1    group2  5
## 16   4.4    group2  6
## 17   5.5    group2  7
## 18   1.6    group2  8
## 19   4.6    group2  9
## 20   3.4    group2 10
```

We will be doing this a LOT throughout this course, but let's practice a little more now.

Practice Exercise #6

Examine the `iris` data, which is another classic data set built in to R. Start by again making a copy in your workspace:

```
myiris <- iris
```

Follow these steps to make this data set tidier and longer:

1. Create a `plantID` column to identify each row as measurements from a different plant
2. Make the data set longer by putting all measurements (which are in cm) into a single column. The names of the columns you are collapsing together should end up in a new variable.
3. The variable where the columns names went still represents two different variables, the part of the plant, and the dimension being measured. Split those into two separate columns. You can do this by giving a vector of names for the `names_to` argument, and then using the `names_pattern` argument (see the help, try to figure it out). One more **hint**: in the pattern, use a single dot to represent "any character" and `\\.` to stand for a literal dot.
4. Now make the data set just a *little* wider, by putting the length and width measurements in different columns.
5. Now create a scatterplot of length by width (using `geom_point()`) and map the `color` aesthetic of `geom_point()` to the species variable.

Extension Exercise #6

1. Decide on a reason to make your data longer or wider. Putting variables in different columns makes it easier to look at differences or correlations or scatterplots. Putting variables in the same column and adding grouping columns makes it easier to do something that compares those groups, like getting cell means or breaking up a plot by color.
2. Change the shape of your data and do the thing you planned (plotting, summarizing, whatever).
3. Reshape the data back to the way it was when you started.

Loops

One of the great things about R is its interactive nature. You write a little code, then you run it, and get a result. Rinse and repeat. But sometimes you want to do something many, many times. This will be immediately useful when we start simulating data, but it can be useful in a wide variety of cases. One of the ways you can do this in R is to use loop structures.

There are a few different types of loops, but we will focus on the `for()` loop in R, which is simple but very flexible. The form of a `for()` loop is simple, and it looks a little like other structures we have seen, like an `if()` block or a function definition:

```
for(iterator in iterator_value_vector) {
  # code block
}
```

Inside the `for()` function, you specify a name of an "iterator", the special word `in`, and then a vector of values that you will loop over. Inside the curly braces that follow the `for()` function, you write the code that you want to loop over. The way it works is that R will run the code inside the block, as many times as you specify (or until it hits an error). Each time through the code block, the value of the object you named as the "iterator" changes, and it cycles through the vector of values that you set up, in order, all the way to the end of the vector.

For example:

```
for(counter in 1:10) {
  cat("here's the number:", counter, "\n")
}
```

```
## here's the number: 1
## here's the number: 2
## here's the number: 3
## here's the number: 4
## here's the number: 5
## here's the number: 6
## here's the number: 7
## here's the number: 8
## here's the number: 9
## here's the number: 10
```

So each time through the loop, we make use of the object called `counter`. The first time through the loop, it has the value of 1, the second time a value of 2, and so on, up to 10. But that's only because we specified the vector 1:10 in the `for()` function. Going in reverse or in an arbitrary order is trivial:

```
for(counter in 10:1) {
  cat("here's the number:", counter, "\n")
}
```

```
## here's the number: 10
## here's the number: 9
## here's the number: 8
## here's the number: 7
## here's the number: 6
## here's the number: 5
## here's the number: 4
## here's the number: 3
## here's the number: 2
## here's the number: 1
```

```
for(counter in c(2, 6, 4, 5, 8, 3, 1, 10, 9, 7)) {
  cat("here's the number:", counter, "\n")
}
```

```
## here's the number: 2
## here's the number: 6
## here's the number: 4
```

```
## here's the number: 5
## here's the number: 8
## here's the number: 3
## here's the number: 1
## here's the number: 10
## here's the number: 9
## here's the number: 7
```

This is all fine and good, but the real value is when you have a block of code that you would otherwise copy and paste, except for one little thing. Because if you can make a vector of values for that "one little thing" you change each time, then you can write the code once, put it inside a loop, and avoid the hazards of copy & paste. The idea here is to write code that does something once, then figure out how to minimally modify that code so that you can loop over it. Let's give this a try.

Subsetting with `filter()`

One more sidebar before the exercise. There are lots of ways to create subsets of data in R. The `tidyverse` uses a handy function called `filter()`. Besides the obligatory first argument (i.e., a data frame – see the pattern here?), you give it conditional expressions that return vectors of logicals, and the data is subsetted to only those rows where these vectors return `TRUE`.

For example, here's how to return only the subset of the `sleep` data where `extra` is greater than 4 and the group condition is "2":

```
sleep %>% filter(extra > 4, group %in% "2")
```

```
##   extra group ID
## 1   4.4     2  6
## 2   5.5     2  7
## 3   4.6     2  9
```

To be clear, the effect of multiple conditionals is the same as an explicit "and" (`&`) operator. If you want the effect of logical "or" (which returns `TRUE` if either thing is `TRUE`), you use the "pipe" symbol `|`.

Practice Exercise #7

1. Create a simple loop to print out a vector of strings, one at a time
2. Create a loop that will print out several different plots to a file:
    a. Use the "tidy" version of the `iris` data (re-run the code to get from the original data to the version where sepals and petals are in different columns).
    b. Create a scatterplot of sepal size by petal size for the whole data set, using color to distinguish length and width.
    c. Use subsetting to plot only the data from the "setosa" species.
    d. Use the `pdf()` and `dev.off()` functions to create a file with this plot in it
    e. Use a loop to create the sample plot for each of the three species, one at a time, writing each of them to the same PDF
    f. Modify this code to create a different PDF for each plot

Extension Exercise #7

1. Figure out some operation that you would like to perform many times over your data (or for many subsets, etc.). Write out what you would like to do in plain English.

2. Use a loop to carry out your idea!

The End

This wraps up this little crash course in data manipulation techniques. We will be using these a lot throughout the course, so I wanted to take the time to try to tackle them in a block, instead of being constantly sidetracked even more than we will be when we tackle real-world data problems. There is a lot more we could discuss (and probably will!) on each of these topics, but this should be enough to get us started so that we can be on the same page going forward.

That said, if not all of this sinks in immediately, that's okay. We'll be practicing a lot over the course. But if there are major confusions, now's the time to discuss them!

As a reminder, please submit an "issue" through GitHub for this assignment, noting any questions, problems, or suggestions you might have. That will help me track what we should try to focus on as a group in the Tuesday sessions.

Thanks, and good luck!