Data Wrangling I: Enter the tidyverse

EDH7916 | Summer C 2020

Benjamin Skinner

We take our first full dive into R in this lesson. Though every data wrangling / analysis is unique, the best way to learn and practice R is to answer a question using data. By the end of this lesson, you will have read in a data set, lightly cleaned it, produced results, and saved your findings in a new file. As part of this process, you will have practiced translating a research question into data analysis steps, which is a skill every bit as important as technical sophistication with a statistical language like R.

Core processes in a data analysis

Large or small, a typical data analysis will involve most — if not all — of the following steps:

1. **Read** in data
2. **Select** variables (columns)
3. **Mutate** data into new forms
4. **Filter** observations (rows)
5. **Summarize** data
6. **Arrange** data
7. **Write** out updated data

Returning to our cooking metaphor from the organizing lesson: if the first and last steps represent our raw ingredients and finished dish, respectively, then the middle steps are the core processes we use to prepare the meal.

As you can see, there aren't that many core processes to use. Their power comes from the infinite ways they can be ordered and combined. There are, of course, many specialized tools for specialized data wrangling tasks — too many to cover in this course (Google is your friend here!). But I call these processes "core processes" for a reason — they will be at the center of most of your data analytic work.

Tidyverse

The tidyverse is shorthand for a number of packages that are built to work well together and can be used in place of base R functions. A few of the tidyverse packages that you will often use are:

- dplyr for data manipulation
- tidyr for making data tidy
- readr for flat file I/O
- readxl for Excel file I/O
- haven for other file format I/O

- ggplot2 for making graphics
- stringr for working with strings
- lubridate for working with dates
- purrr for working with functions

Many R users find functions from these libraries to be more intuitive than base R functions. In some cases, tidyverse functions are faster than base R, which is an added benefit when working with large data sets.

Today we will primarily use functions from the **dplyr** and **readr** libraries. We'll also use some common base R functions as necessary.

```
## ---------------------------
## libraries
## ---------------------------

library(tidyverse)
```

```
── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──
✔ ggplot2 3.3.0     ✔ purrr   0.3.4
✔ tibble  3.0.1     ✔ dplyr   0.8.5
✔ tidyr   1.0.3     ✔ stringr 1.4.0
✔ readr   1.3.1     ✔ forcats 0.5.0
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
```

Check working directory

This script — like the one from the organizing lesson — assumes that the `scripts` subdirectory is the working directory, that the required data file is in the `data` subdirectory, and that both subdirectories are at the same level in the course directory. Like this:

```
edh7916/                <--- Top-level
|
|__/data                <--- Sub-level 1
|    |--+ hsls_small.csv
|
|__/scripts             <--- Sub-level 1 (Working directory)
     |--+ dw_one.R
```

If you need a refresher on setting the working directory, see the prior lesson.

Notice that I'm not setting (*i.e.* hard coding) the working directory in the script. That would not work well for sharing the code. Instead, I tell you where you need to be (a common landmark), let you get there, and then rely on relative paths afterwards.

```
## ---------------------------
## directory paths
## ---------------------------

## assume we're running this script from the ./scripts subdirectory
dat_dir <- file.path("..", "data")
```

Example data analysis task

Let's imagine we've been given the following data analysis task with the HSLS09 data:

> *Figure out average differences in college degree expectations across census regions; for a first pass, ignore missing values and use the higher of student and parental expectations if an observation has both.*

A primary skill (often unremarked upon) in data analytic work is translation. Your advisor, IR director, funding agency director — even collaborator — won't speak to you in the language of R. Instead, it's up to you to (1) translate a research question into the discrete steps coding steps necessary to provide an answer, and then (2) translate the answer such that everyone understands what you've found.

What we need to do is some combination of the following:

1. **Read** in the data
2. **Select** the variables we need
3. **Mutate** a new value that's the higher of student and parental degree expectations
4. **Filter** out observations with missing degree expectation values
5. **Summarize** the data within region to get average degree expectation values
6. **Arrange** in order so it's easier to rank and share
7. **Write** out the results to a file so we have it for later

Let's do it!

Read in data

For this lesson, we'll use a subset of the High School Longitudinal Study of 2009 (HSLS09), an IES / NCES data set that features:

- Nationally representative, longitudinal study of 23,000+ 9th graders from 944 schools in 2009, with a first follow-up in 2012 and a second follow-up in 2016
- Students followed throughout secondary and postsecondary years
- Surveys of students, their parents, math and science teachers, school administrators, and school counselors
- A new student assessment in algebraic skills, reasoning, and problem solving for 9th and 11th grades
- 10 state representative data sets

If you are interested in using HSLS09 for future projects, **DO NOT** rely on this subset. Be sure to download the full data set with all relevant variables and weights if that's the case. But for our purposes in this lesson, it will work just fine.

Throughout, we'll need to consult the code book. An online version can be found at this link (after a little navigation).

> **Quick exercise** Follow the code book link above in your browser and navigate to the HSLS09 code book.

```
## ----------------------------
## input
## ----------------------------
```

```
## data are CSV, so we use read_csv() from the readr library
df <- read_csv(file.path(dat_dir, "hsls_small.csv"))
```

```
Parsed with column specification:
cols(
  stu_id = col_double(),
  x1sex = col_double(),
  x1race = col_double(),
  x1stdob = col_double(),
  x1txmtscor = col_double(),
  x1paredu = col_double(),
  x1hhnumber = col_double(),
  x1famincome = col_double(),
  x1poverty185 = col_double(),
  x1ses = col_double(),
  x1stuedexpct = col_double(),
  x1paredexpct = col_double(),
  x1region = col_double(),
  x4hscompstat = col_double(),
  x4evratndclg = col_double(),
  x4hs2psmos = col_double()
)
```

Unlike the `readRDS()` function we've used before, `read_csv()` prints out information about the data just read in. Nothing is wrong! The `read_csv()` function, like many other functions in the tidyverse, assumes you'd rather have more rather than less information and acts accordingly. We can see that all the columns were read in as doubles (`col_double()`), which is just a type of number that the computer understands in a special way (a distinction that's not important for us in this case). For other data (or if we had told `read_csv()` how to parse the columns), we might see other column types like:

- `col_integer()`: another type of number (again, an important distinction for the computer, but not usually for us)
- `col_character()`: strings (*e.g.*, `"Ben"` or `"1"` [notice the quotes])
- `col_logical()`: Boolean values of `TRUE` or `FALSE`

> **Quick exercise** `read_csv()` is special version of `read_delim()`, which can read various *delimited* file types, that is, tabular data in which data cells are separated by a special character. What's the special character used to separate *CSV* files? Once you figure it out, re-read in the data using `read_delim()`, being sure to set the `delim` argument to the correct character.

Select variables (columns)

To choose variables, either when making a new data frame or dropping them, use `select()`. Like the other **dplyr** functions we'll use, the first argument `select()` takes is the data frame (or tibble) object. After that, we list the column names we want to keep.

Some pseudocode for using `select()` is:

```
## pseudocode (not to be run)
select(< df object >, column_1_name, column_2_name)
```

Because we don't want to overwrite our original data in memory, we'll assign (`<-`) the output to a new object called `df_tmp`.

```r
## -----------------
## select
## -----------------

## select columns we need and assign to new object
df_tmp <- select(df, stu_id, x1stuedexpct, x1paredexpct, x1region)

## show
df_tmp
```

```
# A tibble: 23,503 x 4
   stu_id x1stuedexpct x1paredexpct x1region
    <dbl>        <dbl>        <dbl>    <dbl>
 1  10001            8            6        2
 2  10002           11            6        1
 3  10003           10           10        4
 4  10004           10           10        3
 5  10005            6           10        3
 6  10006           10            8        3
 7  10007            8           11        1
 8  10008            8            6        1
 9  10009           11           11        3
10  10010            8            6        1
# … with 23,493 more rows
```

Mutate data into new forms

To add variables and change existing ones, use the `mutate()` function.

Just like `select()`, the `mutate()` function takes the data frame as the first argument, followed by variable name, new or old, that is created/modified by some function:

```r
## pseudocode (not to be run)
mutate(< df object >, column_1_name = function(...))
```

In this case, the `function(...)` (or, *stuff we want to do*) is add a new column that is the larger of `x1stuedexpct` and `x1paredexpct`.

Understanding our data

First things first, however, we need to check the code book to see what the numerical values for our two education expectation variables represent. To save time, I've copied them here:

`x1stuedexpct` *How far in school 9th grader thinks he/she will get*

value	label
1	Less than high school
2	High school diploma or GED
3	Start an Associate's degree
4	Complete an Associate's degree
5	Start a Bachelor's degree
6	Complete a Bachelor's degree
7	Start a Master's degree
8	Complete a Master's degree

value	label
9	Start Ph.D/M.D/Law/other prof degree
10	Complete Ph.D/M.D/Law/other prof degree
11	Don't know
-8	Unit non-response

x1paredexpct *How far in school parent thinks 9th grader will go*

value	label
1	Less than high school
2	High school diploma or GED
3	Start an Associate's degree
4	Complete an Associate's degree
5	Start a Bachelor's degree
6	Complete a Bachelor's degree
7	Start a Master's degree
8	Complete a Master's degree
9	Start Ph.D/M.D/Law/other prof degree
10	Complete Ph.D/M.D/Law/other prof degree
11	Don't know
-8	Unit non-response
-9	Missing

The good news is that the categorical values are the same for both variables (meaning we can make an easy comparison) and move in a logical progression. The bad news is that we have three values — `-8`, `-9`, and `11` — that we need to deal with so that the averages we compute later represent what we mean.

Let's see how many observations are affected by these values using `count()` (notice that we don't assign to a new object; this means we'll see the result in the console, but nothing in our data or object will change):

```
## -----------------
## mutate
## -----------------

## see unique values for student expectation
count(df_tmp, x1stuedexpct)
```

```
# A tibble: 12 x 2
   x1stuedexpct     n
          <dbl> <int>
 1           -8  2059
 2            1    93
 3            2  2619
 4            3   140
 5            4  1195
 6            5   115
 7            6  3505
 8            7   231
 9            8  4278
10            9   176
11           10  4461
12           11  4631
```

```r
## see unique values for parental expectation
count(df_tmp, x1paredexpct)
```

```
# A tibble: 13 x 2
   x1paredexpct     n
          <dbl> <int>
 1           -9    32
 2           -8  6715
 3            1    55
 4            2  1293
 5            3   149
 6            4  1199
 7            5   133
 8            6  4952
 9            7    76
10            8  3355
11            9    37
12           10  3782
13           11  1725
```

Dealing with `-8` and `-9` is straightforward — we'll convert it missing. In R, missing values are technically stored as `NA`. Not all statistical software uses the same values to represent missing values (for example, Stata uses a dot `.`). Likely because they want to be software agnostic, NCES has decided to represent missing values as a limited number of negative values. In this case, `-8` and `-9` represent missing values.

How to handle missing values is a **very** important topic, one we could spend all semester discussing. For now, we are just going to drop observations with missing values; but be forewarned that how you handle missing values can have real ramifications for the quality of your final results.

Deciding what to do with `11` is a little trickier. While it's not a missing value *per se*, it also doesn't make much sense in its current ordering, that is, to be "higher" than completing a professional degree. We'll make a decision to convert these to `NA` as well, effectively deciding that an answer of "I don't know" is the same as missing an answer.

Changing existing variables (columns)

So first step: convert `-8`, `-9`, and `11` in both variables to `NA`. We can do this by overwriting cells in each variable with `NA` when they equal one of these two values. For this, we'll use the `ifelse()` function, which has three parts:

```r
## pseudo code (not to be run)
ifelse(< test >, < return this if TRUE >, < return this if FALSE >)
```

`ifelse()` works by asking: `if` the `< test >` is `TRUE` do **this** `else` (*i.e.* the `< test >` is `FALSE`) do **that**. By *test*, I mean a code statement that evaluates to either `TRUE` or `FALSE`. For example:

- `1 == 1` (`TRUE`)
- `1 == 2` (`FALSE`)
- `1 + 1 == 2` (`TRUE`)

NOTE When checking whether something equals something else, use a double equals sign (`==`); a single equals sign typically means assignment `=` or the same thing as the arrow, `<-`.

What we want is to go row by row (that is, observation by observation) through both `x1stuedexpct` and `x1paredexpct` and test whether the value is either `-8`, `-9`, or `11` — if it is, then replace with `NA`, otherwise, just replace it with the value it found (*i.e.* leave it alone).

We could develop a sophisticated test that looked for any of these conditions, but we can also just do it with

multiple `ifelse()` functions. Notice, however, that we can test for both `-8` and `-9` in the same test since all the categories we want are positive.

```
## use case_when to overwrite -8 and 11 with NA in our two expectation variables
df_tmp <- mutate(df_tmp,
                 ## correct student expectations
                 x1stuedexpct = ifelse(x1stuedexpct < 0,   # is value < 0?
                                       NA,                 # T: replace with NA
                                       x1stuedexpct),      # F: replace with self
                 x1stuedexpct = ifelse(x1stuedexpct == 11, # is value == 11?
                                       NA,                 # T: replace with NA
                                       x1stuedexpct),      # F: replace with self
                 ## correct parental expectations
                 x1paredexpct = ifelse(x1paredexpct < 0,   # (same as above...)
                                       NA,
                                       x1paredexpct),
                 x1paredexpct = ifelse(x1paredexpct == 11,
                                       NA,
                                       x1paredexpct))
```

Notice that we used `df_tmp` rather than `df`. That's because we want to carry through the work we did with `select()` before. If we used `df` instead, then we'd be back to the original data object — not what we want.

Let's confirm that our code worked as we planned by using `count()` again.

```
## again see unique values for student expectation
count(df_tmp, x1stuedexpct)
```

```
# A tibble: 11 x 2
   x1stuedexpct     n
          <dbl> <int>
 1            1    93
 2            2  2619
 3            3   140
 4            4  1195
 5            5   115
 6            6  3505
 7            7   231
 8            8  4278
 9            9   176
10           10  4461
11           NA  6690
```

```
## again see unique values for parental expectation
count(df_tmp, x1paredexpct)
```

```
# A tibble: 11 x 2
   x1paredexpct     n
          <dbl> <int>
 1            1    55
 2            2  1293
 3            3   149
 4            4  1199
 5            5   133
 6            6  4952
 7            7    76
 8            8  3355
```

```
 9            9    37
10           10  3782
11           NA  8472
```

Adding new variables (columns)

Adding a new variable to our data frame is just like modifying an existing column. The only difference is that instead of putting an existing column name on the LHS of the = sign in `mutate()`, we'll make up a new name. This tells R to make a new column in our data frame that contains the results from the the RHS function(s).

```
## pseudocode (not to be run)
mutate(< df object >, new_column_name = function(...))
```

Now that we've corrected our expectation variables, we create a new variable that is the higher of the two (per our initial instructions to choose the higher of the two if both existed).

Using `ifelse()` again, we can test whether each student's degree expectation is higher than that of their parent; if true, we'll put the student's value into the new variable — if false, we'll put the parent's value into the new variable.

```
## mutate (notice that we use df_tmp now)
df_tmp <- mutate(df_tmp,
                 high_expct = ifelse(x1stuedexpct > x1paredexpct, # test
                                     x1stuedexpct,                # if TRUE
                                     x1paredexpct))               # if FALSE

## show
df_tmp
```

```
# A tibble: 23,503 x 5
   stu_id x1stuedexpct x1paredexpct x1region high_expct
    <dbl>        <dbl>        <dbl>    <dbl>      <dbl>
 1  10001            8            6        2          8
 2  10002           NA            6        1         NA
 3  10003           10           10        4         10
 4  10004           10           10        3         10
 5  10005            6           10        3         10
 6  10006           10            8        3         10
 7  10007            8           NA        1         NA
 8  10008            8            6        1          8
 9  10009           NA           NA        3         NA
10  10010            8            6        1          8
# … with 23,493 more rows
```

Doing a quick "ocular test" of our first few rows, it seems like our new variable is correct…**EXCEPT**…it doesn't look like we handled `NA` values correctly. Look at student `10002` in the second row: while the student doesn't have an expectation (or said "I don't know"), the parent does. However, our new variable records `NA`. Let's fix it with this test:

> If `high_expct` **is missing and** `x1stuedexpct` **is not missing**, replace with that; otherwise replace with itself (leave alone). Repeat, but for `x1paredexpct`. If still `NA`, then we can assume both student and parent expectations were missing.

Translating the bold words to R code:

- **is missing**: `is.na()`
- **and**: `&`

- **is not missing**: `!is.na()` (`!` means **NOT**)

we get:

```
## correct for NA values
df_tmp <- mutate(df_tmp,
                 ## step 1: compare with student's expectations
                 high_expct = ifelse(is.na(high_expct) & !is.na(x1stuedexpct),
                                     x1stuedexpct,
                                     high_expct),
                 ## step 2: compare with parent's expectations
                 high_expct = ifelse(is.na(high_expct) & !is.na(x1paredexpct),
                                     x1paredexpct,
                                     high_expct))

## show
df_tmp
```

```
# A tibble: 23,503 x 5
   stu_id x1stuedexpct x1paredexpct x1region high_expct
    <dbl>        <dbl>        <dbl>    <dbl>      <dbl>
 1  10001            8            6        2          8
 2  10002           NA            6        1          6
 3  10003           10           10        4         10
 4  10004           10           10        3         10
 5  10005            6           10        3         10
 6  10006           10            8        3         10
 7  10007            8           NA        1          8
 8  10008            8            6        1          8
 9  10009           NA           NA        3         NA
10  10010            8            6        1          8
# … with 23,493 more rows
```

Looking at the second observation again, it looks like we've fixed our `NA` issue. Looking at rows 7 and 9, it seems like those situations are correctly handled as well.

To be clear, there were other ways we could have handled fixing our missing values and creating our new variable. For example, we could have left our missing values as negative numbers (and converting 11 to a negative value) so that our comparison would have worked the first time. We could have used more sophisticated tests in our `ifelse()` statements. However, these paths weren't clear until we'd already worked a bit. The point to keep in mind that the process is often iterative (two steps forward, one step back…) and that there's seldom an single *correct way*.

> **Quick exercise** What happens when the student and parent expectations are the same, either a value or `NA`? Does our `ifelse()` statement account for those situations? If so, how?

Filter observations (rows)

Let's check the counts of our new variable:

```
## -----------------
## filter
## -----------------
```

```
## get summary of our new variable
count(df_tmp, high_expct)
```

```
# A tibble: 11 x 2
   high_expct     n
        <dbl> <int>
 1          1    71
 2          2  2034
 3          3   163
 4          4  1282
 5          5   132
 6          6  4334
 7          7   191
 8          8  5087
 9          9   168
10         10  6578
11         NA  3463
```

Since we're not going to use the missing values (we really can't, even if we wanted to do so), we'll drop those observations from our data frame using `filter()`.

An important point about `filter()` that often trips people up at first: use it to *filter in what you* want. This is the opposite of the more common usage of filters, which are about removing things (*e.g.*, air filters, water filters, *etc*).

```
## filter out missing values
df_tmp <- filter(df_tmp, !is.na(high_expct))

## show
df_tmp
```

```
# A tibble: 20,040 x 5
   stu_id x1stuedexpct x1paredexpct x1region high_expct
    <dbl>        <dbl>        <dbl>    <dbl>      <dbl>
 1  10001            8            6        2          8
 2  10002           NA            6        1          6
 3  10003           10           10        4         10
 4  10004           10           10        3         10
 5  10005            6           10        3         10
 6  10006           10            8        3         10
 7  10007            8           NA        1          8
 8  10008            8            6        1          8
 9  10010            8            6        1          8
10  10011            8            6        3          8
# … with 20,030 more rows
```

It looks like we've dropped the rows with missing values in our new variable (or, more technically, *kept* those without missing values). Since we haven't removed rows until now, we can compare the number of rows in the original data frame, `df`, to what we have now.

```
## is the original # of rows - current # or rows == NA in count?
nrow(df) - nrow(df_tmp)
```

```
[1] 3463
```

Comparing the difference, we can see it's the same as the number of missing values in our new column. While not a formal test, it does support what we expected (in other words, if the number were different,

we'd definitely want to go back and investigate).

Summarize data

Now we're ready to get the average of expectations that we need. The `summarize()` command will allow us to apply a summary measure, like `mean()`, to a column of our data. (**NOTE** that if we wanted another summary measure, like the median or standard deviation, there are other functions like `median()` and `sd()`…if you need a particular stat, there's likely a function for it!).

```
## -----------------
## summarize
## -----------------

## get average (without storing)
summarize(df_tmp, high_expct_mean = mean(high_expct))
```

```
# A tibble: 1 x 1
  high_expct_mean
            <dbl>
1            7.27
```

Overall, we can see that students and parents have high postsecondary expectations on average: to earn some graduate credential beyond a bachelor's degree. However, this isn't what we want. We want the values across census regions.

```
## check our census regions
count(df_tmp, x1region)
```

```
# A tibble: 4 x 2
  x1region     n
     <dbl> <int>
1        1  3128
2        2  5312
3        3  8177
4        4  3423
```

We're not missing any census data, which is good. To calculate our average expectations, we need to use the `group_by()` function. This function allows to set groups and perform other **dplyr** operations *within* those groups. Right now, we'll use it to get our summary.

```
## get expectations average within region
df_tmp <- group_by(df_tmp, x1region)

## show grouping
df_tmp
```

```
# A tibble: 20,040 x 5
# Groups:   x1region [4]
   stu_id x1stuedexpct x1paredexpct x1region high_expct
    <dbl>        <dbl>        <dbl>    <dbl>      <dbl>
 1  10001            8            6        2          8
 2  10002           NA            6        1          6
 3  10003           10           10        4         10
 4  10004           10           10        3         10
 5  10005            6           10        3         10
 6  10006           10            8        3         10
 7  10007            8           NA        1          8
```

```
 8  10008            8            6        1          8
 9  10010            8            6        1          8
10  10011            8            6        3          8
# … with 20,030 more rows
```

Notice the extra row at the second line now? `Groups: x1region [4]` tells us that our data set is now grouped.

Quick exercise What does the `[4]` mean?

Now that our groups are set, we can get the summary we really wanted

```
## get average (assigning this time)
df_tmp <- summarize(df_tmp, high_expct_mean = mean(high_expct))

## show
df_tmp
```

```
# A tibble: 4 x 2
  x1region high_expct_mean
     <dbl>           <dbl>
1        1            7.39
2        2            7.17
3        3            7.36
4        4            7.13
```

Success! Expectations are similar across the country, but not the same by region.

NB: The reason we didn't assign the first ungrouped `summarize()` function back to `df_tmp` is that `summarize()` fundamentally changes our data frame, from one of many observations to one that represents their summary (as we asked!). Keep this in mind for your own analyses: the order of operations matters. If you select columns, then columns you didn't selection won't be available later; if you summarize your data, then you only have access to the summary.

Arrange data

As our final step, we'll arrange our data frame from highest to lowest (descending). For this, we'll use `arrange()` and a special operator, `desc()` which is short for *descending*.

```
## -----------------
## arrange
## -----------------

## arrange from highest expectations (first row) to lowest
df_tmp <- arrange(df_tmp, desc(high_expct_mean))

## show
df_tmp
```

```
# A tibble: 4 x 2
  x1region high_expct_mean
     <dbl>           <dbl>
1        1            7.39
2        3            7.36
3        2            7.17
4        4            7.13
```

Quick exercise What happens when you don't include `desc()` around `high_expct_mean`?

Write out updated data

We can use this new data frame as a table in its own right or to make a figure. For now, however, we'll simply save it using the opposite of `read_csv()` — `write_csv()` — which works like `writeRDS()` we've used before.

```
## write with useful name
write_csv(df_tmp, file.path(dat_dir, "high_expct_mean_region.csv"))
```

And with that, we've met our task: we can show average educational expectations by region. To be very precise, we can show the higher of student and parental educational expectations among those who answered the question by region. This caveat doesn't necessarily make our analysis less useful, but rather sets its scope. Furthermore, we've kept our original data as is (we didn't overwrite it) for future analyses while saving the results of this analysis for quick reference.

Pipes (`%>%`)

Above, we've performed each step of our analysis piecemeal, saving new objects or overwriting along the way. This is fine, but a huge benefit of the **tidyverse** is that it allows users to chain commands together using pipes.

Tidyverse pipes, `%>%`, come from the magrittr package.

Pipes take output from the left side and pipe it to the input of the right side. So `mean(x)` can be rewritten as `x %>% mean`: `x` outputs itself and the pipe, `%>%`, makes it the input for `mean()`.

Quick exercise Store 1,000 random values in `x`: `x <- rnorm(1000)`. Now run `mean(x)` and `x %>% mean`. Do you get the same thing?

This may be a silly example (why would you do that?), but pipes are powerful because they allow data wrangling processes to be chained together.

Normally, functions (like `select()`, `mutate()`, *etc*), can be nested in R, but after too many, the code becomes difficult to parse since it has to be read from the inside out. For this reason, many analysts run one discrete function after another, saving output along the way. This is what we did above.

Pipes allow functions to come one after another in the order of the work being done, which is more legible. As a bonus, chaining functions together is sometimes faster due to behind-the-scenes processing.

Let's use Hadley's canonical example to make the readability comparison between nested functions and piped functions clearer:

```
## foo_foo is an instance of a little bunny function
foo_foo <- little_bunny()

## adventures in base R must be read from the middle up and backwards
bop_on(
    scoop_up(
        hop_through(foo_foo, forest),
        field_mouse
    ),
    head
)

## adventures w/ pipes start at the top and work down
foo_foo %>%
    hop_through(forest) %>%
    scoop_up(field_mouse) %>%
    bop_on(head)
```

In the first set, we have to read the story of little bunny foo foo from the inside out: "Little bunny foo_foo bopped on the head a field mouse that was scooped up while hopping through the forest."

With pipes, we can read it more like the original rhyme: "Little bunny foo foo hopped through the forest, scooped up a field mouse, and bopped it on the head."

The main thing to remember is with pipes and tidyverse is that because the output of the function goes into the next function, you don't need to include the first argument, that is, the data frame object name.

This:

```
df_example <- select(df, col1, col2)
```

becomes this:

```
df_example <- df %>% select(col1, cols2)
```

In the second example, the object `df` was piped into the first argument spot in `select()`. Since that's already accounted for, we were able to just start with the column names we want.

Rewriting our analysis using pipes

Returning to our analysis above, let's rewrite all our of our steps in on piped chain of commands.

```
## start with the original data frame...
df_tmp_chained <- df %>%
    ## (df is piped in): select the columns we want
    select(stu_id, x1stuedexpct, x1paredexpct, x1region) %>%
    ## (selected df is piped in): mutate our data, starting with missing
```

```
    mutate(x1stuedexpct = ifelse(x1stuedexpct < 0,
                                 NA,
                                 x1stuedexpct),
           x1stuedexpct = ifelse(x1stuedexpct == 11,
                                 NA,
                                 x1stuedexpct),
           x1paredexpct = ifelse(x1paredexpct < 0,
                                 NA,
                                 x1paredexpct),
           x1paredexpct = ifelse(x1paredexpct == 11,
                                 NA,
                                 x1paredexpct),
           ## create new column
           high_expct = ifelse(x1stuedexpct > x1paredexpct,
                               x1stuedexpct,
                               x1paredexpct),
           ## fix new column NAs
           high_expct = ifelse(is.na(high_expct) & !is.na(x1stuedexpct),
                               x1stuedexpct,
                               high_expct),
           high_expct = ifelse(is.na(high_expct) & !is.na(x1paredexpct),
                               x1paredexpct,
                               high_expct)) %>%
    ## (mutated df is piped in): filter in non-missing rows
    filter(!is.na(high_expct)) %>%
    ## (filtered df is piped in): group by region
    group_by(x1region) %>%
    ## (grouped df is piped in): summarize mean expectations
    summarize(high_expct_mean = mean(high_expct)) %>%
    ## (summarized df is piped in): arrange average expectations hi --> lo
    arrange(desc(high_expct_mean))
```

Notice how I included comments along the way? Since R ignores commented lines (it's as if they don't exist), you can include within your piped chain of commands. This is a good habit that collaborators and future you will appreciate.

To be sure, let's check: is the result the same as before?

```
## show
df_tmp_chained
```

```
# A tibble: 4 x 2
  x1region high_expct_mean
     <dbl>           <dbl>
1        1            7.39
2        3            7.36
3        2            7.17
4        4            7.13
```

```
## test using identical()
identical(df_tmp, df_tmp_chained)
```

```
[1] TRUE
```

Success!

Final notes

In this lesson, you've converted a research question into a data analysis that started with reading in raw data and ended with your summary table. You've seen how to wrangle your data in multiple discrete steps as well as chained together using pipes.

As you start to apply these tools to your own analyses, the first questions should always be:

1. What I am trying to do?
2. What would my data or results need to look like in order to do that?
3. What are the discrete steps I need to get from what I have (raw data) to what I need (wrangled data)?

Remember, these are iterative questions, meaning you will almost certainly need to revisit and adjust during your analysis. But becoming a better quantitative researcher mostly means becoming a better translator: question –> data/coding –> answer.