Data Wrangling I: Enter the {tidyverse}

EDH7916 | Spring 2020

Benjamin Skinner

Large or small, a typical data analysis project 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 last lesson, if the first and last steps represent our raw ingredients and finished dish, respectively, then the middle steps are the core techniques we use to prepare the meal.

As you can see, there aren't that many. (There are, of course, many specialized tools for specialized tasks — too many to cover in this course.) But the power of the core techniques comes from the infinite ways they can be ordered and combined.

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 focus on functions from the **{dplyr}** and **{readr}** libraries.

```
## ---------------------------
## 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   1.0.2     ✔ 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 — unlike the others we've used so far — assumes that you are in the `scripts` directory, one down from top-level project directory. You can check by typing `getwd()` in the R console. If your path is anything other than `<path>/<to>/edh7916/scripts` (or whatever you've named your `student_<last_name>` repo), then use the R command `setwd()` with relative paths to get where you need to be.

For example, if I run `getwd()` and see `~/courses/edh7916`, that means my working directory is the top-level, not `scripts`. I can get to the right spot by typing `setwd("./scripts")` in the console: it tells R, go down one level into (`./scripts`) and make that the working directory. Were I to run `getwd()` afterwards, I should see `~/courses/edh7916/scripts`.

You may not need to do this. But if you open RStudio using the **Open Project** menu above the upper-right panel, you most likely will be in the top-level project directy — not quite right. Checking and setting your working directory at the start of a session is a pain, but is a useful skill for when you work with collaborators across multiple projects.

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")
```

Read in data

So far, we've only used small nonce data sets. 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 datasets

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 codebook link above in your browser and navigate to the HSLS09 codebook.

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

## data are CSV, so we use readr::read_csv()
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()`). For other data (or if we had told `read_csv()` how to parse the columns), we might see other column types like `col_integer()`, `col_character()`, or `col_logical()`.

Quick exercise `readr::read_csv()` is special version of `readr::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.

NOTE When a function comes from a specific library, I will sometimes signal that using `::` notation: `<package>::<function>`. Along as you've installed the package, you can actually call functions in R using this notation if you don't want to load the library at the top of the file (*e.g.* `library(tidyverse)`). We won't do that. I use this notation so it's clear to you which R library a function comes from. When see `readr::read_csv()`, that means the `read_csv()` function, which is part of the {readr} library.

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

Let's do it!

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.

Since we don't want to overwrite our original data in memory, we'll create an new object called `df_tmp`)

```
## select
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 *the stuff we want to do*. In this case, *the stuff we want to do* is add a new column that is the larger of `x1stuedexpct` and `x1paredexpct`. Some pseudocode for making a new column is:

```
## pseudo code (not to be run)
mutate(<dataframe>, <new_var> = function(<old_var>))
```

Sometimes, that `function()` could just be adding a value: `mutate(df, col_of_ones = 1)`.

> **Quick exercise** Without assigning to any object (no `<-`), type `mutate(df, col_of_ones = 1)`. What do you get?

Usually the function will be more interesting. In this case we need to choose the bigger of two values. This is an ideal use case for the `ifelse()` function, which works like this:

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

Using this logic within `mutate()`, 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           11            6        1         11
 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           11        1         11
 8  10008            8            6        1          8
 9  10009           11           11        3         11
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.

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

Filter observations (rows)

```
## get summary of our new variable
table(df_tmp$high_expct)
```

```
  -8    1    2    3    4    5    6    7    8    9   10   11
1732   47 1399  110  830   89 2982  169 4257  152 5810 5926
```

What do these values mean? What's up with the negative values.

Quick exercise Use the online codebook to figure out what these values mean.

In R, missing values are technically stored as `NA`. But not all statistical software uses the same values to represent missing values (for example, Stata uses a dot `.`) NCES has decided to represent missing values as a limited number of negative values. In this case, `-8` represents a missing value.

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.

```
## filter out missing values
df_tmp <- filter(df_tmp, high_expct != -8)

## notice the number of rows now:
## do they equal the first number minus the number of NAs from our table command?
df_tmp
```

```
# A tibble: 21,771 x 5
   stu_id x1stuedexpct x1paredexpct x1region high_expct
    <dbl>        <dbl>        <dbl>    <dbl>      <dbl>
 1  10001            8            6        2          8
 2  10002           11            6        1         11
 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           11        1         11
 8  10008            8            6        1          8
 9  10009           11           11        3         11
10  10010            8            6        1          8
# … with 21,761 more rows
```

Quick exercise What does the logical operator `!=` mean? How else could we have filtered our data?

Summarize data

Now we're ready to get the average. The `summarize()` command will allow us to apply a summary measure, like `mean()`, to a column of our data.

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

```
# A tibble: 1 x 1
  high_expct_mean
            <dbl>
1            8.48
```

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
table(df_tmp$x1region)
```

```
   1    2    3    4
3383 5779 8846 3763
```

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 **{dpylr}** 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: 21,771 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           11            6        1         11
 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           11        1         11
 8  10008            8            6        1          8
 9  10009           11           11        3         11
10  10010            8            6        1          8
# … with 21,761 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 (without storing)
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            8.62
2        2            8.44
3        3            8.49
4        4            8.42
```

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

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()`.

```
## 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            8.62
2        3            8.49
3        2            8.44
4        4            8.42
```

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"))
```

Pipes (`%>%`)

So far, we've performed each step of our analysis piecemeal. 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 `sum(x)` can be rewritten as `x %>% sum`: `x` outputs itself and the pipe, `%>%`, makes it the input for `sum()`.

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

This may be a silly example (why would you do that?), but pipes are powerful because they can be chained together. Functions 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, but in a connected fashion that is cleaner and, 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
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)
```

Rewriting our analysis using pipes

Returning to our example above, let's connect the steps using pipes. To start off, we can pipe our data to the `select()` command:

```
## pipe the original data frame into select
df_tmp_2 <- df %>%
    select(stu_id, x1stuedexpct, x1paredexpct, x1region)

## show
df_tmp_2
```

```
# 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
```

Notice how we didn't put `df` inside `select()`? That's because `df` is piped using `%>%` into the first argument of `select()`. Since that's already accounted for, we can just start with the column names we want.

We can keep the chain going to our next step:

```
## same as before, but add mutate()
df_tmp_2 <- df %>%
    ## vars to select
    select(stu_id, x1stuedexpct, x1paredexpct, x1region) %>%
    ## vars to add
    mutate(high_expct = ifelse(x1stuedexpct > x1paredexpct, # test
                               x1stuedexpct,                # if TRUE
                               x1paredexpct))               # if FALSE

## show
df_tmp_2
```

```
# 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           11            6        1         11
 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           11        1         11
 8  10008            8            6        1          8
 9  10009           11           11        3         11
10  10010            8            6        1          8
# … with 23,493 more rows
```

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.

> **Quick exercise** Redo our initial analysis using a single chain. You don't need to save the output as a data file with `write_csv()`, but you should get the same final four-row data frame we got before. If you want to formally test, you can use the function `identical(df_tmp, df_tmp_2)` to see if they are the same.