Data Wrangling with Base R

EDH7916

Benjamin Skinner

This supplemental lesson is a companion to the lesson on data wrangling with Tidyverse[1]. While the tidyverse has fundamentally changed the way many people work with R, you can still use base or "vanilla" R for your data wrangling tasks. You may find it useful to perform an example data analysis without the support of the tidyverse.

Because much of the material is otherwise the same, we won't go into the same depth in this supplemental lesson (you'll even see some repeated text). We'll simply work to answer the research question posed in the other lesson.

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!

NOTE: Since we're not using the **tidyverse**, we don't need to call it this time. Even with non-tidyverse R, you may need to call libraries. This analysis, however, does not require any.

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

## NONE
```

[1] https:/equant.github.io/edh7916/lessons/dw_one.html

Check working directory

This script — like the one from the organizing lesson[2] — 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:

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

If you need a refresher on setting the working directory, see the prior lesson[3].

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

For this lesson, we'll use a subset of the High School Longitudinal Study of 2009 (HSLS09)[4], 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)[5].

[2] https:/equant.github.io/edh7916/lessons/organizing.html
[3] https:/equant.github.io/edh7916/lessons/organizing.html
[4] https://nces.ed.gov/surveys/hsls09/
[5] https://nces.ed.gov/OnlineCodebook/

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(), which is base R function
df <- read.csv(file.path(dat_dir, "hsls_small.csv"))
```

Unlike the `read_csv()` function we've used before, `read.csv()` (notice the difference: a `.` instead of an _) doesn't print anything. So that we can see our data, well print to the console. **BUT** before we do that…

`read.csv()` returns a base R `data.frame()` rather than the special data frame or `tibble()` that the tidyverse uses. It's mostly the same, but one difference is that whereas R will only print the first 10 rows of a tibble, it will print the *entire* `data.frame`. We don't need to see the whole thing, so we'll use the `head()` function to print only the first 10 rows.

```
## show first 10 rows
head(df, n = 10)
```

```
##    stu_id x1sex x1race x1stdob x1txmtscor x1paredu x1hhnumber x1famincome
## 1   10001     1      8  199502    59.3710        5          3          10
## 2   10002     2      8  199511    47.6821        3          6           3
## 3   10003     2      3  199506    64.2431        7          3           6
## 4   10004     2      8  199505    49.2690        4          2           5
## 5   10005     1      8  199505    62.5897        4          4           9
## 6   10006     2      8  199504    58.1268        3          6           5
## 7   10007     2      8  199409    49.4960        2          2           4
## 8   10008     1      8  199410    54.6249        7          3           7
## 9   10009     1      8  199501    53.1875        2          3           4
## 10  10010     2      8  199503    63.7986        3          4           4
##    x1poverty185   x1ses x1stuedexpct x1paredexpct x1region x4hscompstat
## 1             0  1.5644            8            6        2            1
## 2             1 -0.3699           11            6        1            1
## 3             0  1.2741           10           10        4            1
## 4             0  0.5498           10           10        3            1
## 5             0  0.1495            6           10        3            1
## 6             0  1.0639           10            8        3           -8
## 7             0 -0.4300            8           11        1            1
## 8             0  1.5144            8            6        1            1
## 9             0 -0.3103           11           11        3            1
## 10            0  0.0451            8            6        1           -8
##    x4evratndclg x4hs2psmos
## 1             1          3
## 2             1          3
## 3             1          4
## 4             0         -7
## 5             0         -7
## 6            -8         -8
## 7             1          2
## 8             1          3
```

```
## 9             1          8
## 10           -8         -8
```

> **Quick exercise** *`read.csv()` is special version of `read.table()`, 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.table()`, being sure to set the `sep` argument to the correct character.*

You'll also notice that the data.frame doesn't tell use the types of our columns. If we really want to know, we can use the `class()` function. So that we can see all columns, we can use the `sapply()` function, which will let us *apply* the `class()` function across all columns.

```
## show column types
sapply(df, class)
```

```
##       stu_id        x1sex       x1race      x1stdob   x1txmtscor     x1paredu
##    "integer"    "integer"    "integer"    "integer"    "numeric"    "integer"
##   x1hhnumber  x1famincome x1poverty185        x1ses x1stuedexpct x1paredexpct
##    "integer"    "integer"    "integer"    "numeric"    "integer"    "integer"
##     x1region x4hscompstat x4evratndclg   x4hs2psmos
##    "integer"    "integer"    "integer"    "integer"
```

Select variables (columns)

Data frames are like special matrices. They have rows and columns. You can access these rows and columns using square bracket notation (`[]`). Because a matrix has two dimensions, you use a comma inside the square brackets to indicate what you mean (`[,]`):

- `df[<rows>,<cols>]`

At it's most basic, you can use numbers to represent the index of the cell or cells you're interested in. For example, if you want to access the value of the cell in row 1, column 4, you can use:

```
## show value at row 1, col 4
df[1, 4]
```

```
## [1] 199502
```

Because data frames have column names (the variable names in our data set), we can also refer to them by name. The fourth column is the student date of birth variable, `x1stdob`. Using that instead of `4` (notice the quotation marks `""`):

```
## show value at row 1, x1stdob column
df[1, "x1stdob"]
```

```
## [1] 199502
```

If we want to see more than one column, we can put the names in a concatenated vector using the `c()` function:

```
## show values at row 1, stu_id & x1stdob column
df[1, c("stu_id", "x1stdob")]
```

```
##   stu_id x1stdob
## 1  10001  199502
```

So far, we've not assigned these results to anything, so they've just printed to the console. However, we can assign them to a new object. If we want to slice our data so that we only have selected columns, we can leave the rows section blank (meaning we want all rows) and include all the columns we want to keep in our new data frame object.

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

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

## show 10 rows
head(df_tmp, n = 10)
```

```
##    stu_id x1stuedexpct x1paredexpct x1region
## 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
```

Mutate data into new forms

Changing existing variables (columns)

To conditionally change a variable, we'll once again use the bracket notation to target our changes. This time, however, we do a couple of things differently:

- include square brackets on the LHS of the assignment
- use conditions in the `<rows>` part of the bracket

As before, we need to account for the fact that our two expectation variables, `x1stuedexpct` and `x1paredexpct`, have values that need to be converted to `NA`: `-8`, `-9`, and `11`. See the first data wrangling lesson for the rationale behind these changes.[6]

First, let's look at the unique values using the `table()` function. So that we see any missing values, we'll include an extra argument:

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

## see unique values for student expectation
table(df_tmp$x1stuedexpct, useNA = "ifany")
```

```
##
##   -8    1    2    3    4    5    6    7    8    9   10   11
## 2059   93 2619  140 1195  115 3505  231 4278  176 4461 4631
```

[6] https:/equant.github.io/edh7916/lessons/dw_one.html

```
## see unique values for parental expectation
table(df_tmp$x1paredexpct, useNA = "ifany")
```

```
##
##   -9   -8    1    2    3    4    5    6    7    8    9   10   11
##   32 6715   55 1293  149 1199  133 4952   76 3355   37 3782 1725
```

Notice that we use a dollar sign, `$`, to call the column name directly. Unlike with the tidyverse, we cannot just use the column name. Base R will look for that column name not as a column in a data frame, but as its own object. It probably won't find it (or worse, you'll have another object in memory that it will find and you'll get the wrong thing!).

Remember how data.frames are special matrices? One of the special features is that you can use the `$` sign with the column name as a short cut or double square brackets without the comma.

```
## each version pulls the column of data for student expectations
## TRUE == 1, so if the mean of all values == 1, then all are TRUE
mean(df_tmp$x1stuedexpct == df_tmp[, "x1stuedexpct"]) == 1
```

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

```
mean(df_tmp$x1stuedexpct == df_tmp[["x1stuedexpct"]]) == 1
```

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

Back to replacing our missing values with `NA`…

The conditions we care about are when `df_tmp$x1stuedexpct == -8`, for example. Using that condition in the row section of the square bracket, we can replace only what we want.

```
## replace student expectation values
df_tmp$x1stuedexpct[df_tmp$x1stuedexpct == -8] <- NA
df_tmp$x1stuedexpct[df_tmp$x1stuedexpct == 11] <- NA

## replace parent expectation values
df_tmp$x1paredexpct[df_tmp$x1paredexpct == -8] <- NA
df_tmp$x1paredexpct[df_tmp$x1paredexpct == -9] <- NA
df_tmp$x1paredexpct[df_tmp$x1paredexpct == 11] <- NA
```

What each of these lines says is (first one for example): *In the data.frame `df_tmp`, replace the value in `x1stuedexpct` — where the value of `df_tmp$x1stuedexpct` is `-8` — with `NA`*

As less convoluted way of saying this might be (more generally stated): *Where the value in column A equals X, replace the value in column A with Y.*

Let's confirm using `table()` again. The values that were in `-8`, `-9`, and `11` should now be summed under `NA`.

```
## see unique values for student expectation (confirm changes)
table(df_tmp$x1stuedexpct, useNA = "ifany")
```

```
##
##    1    2    3    4    5    6    7    8    9   10 <NA>
##   93 2619  140 1195  115 3505  231 4278  176 4461 6690
```

```
## see unique values for parental expectation (confirm changes)
table(df_tmp$x1paredexpct, useNA = "ifany")
```

```
##
##    1    2    3    4    5    6    7    8    9   10 <NA>
##   55 1293  149 1199  133 4952   76 3355   37 3782 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 after the first `$` sign, we'll make up a new name. This tells R to add a new column to our data frame.

As with the tidyverse version, we'll use the `ifelse()` function to create a new variable that is the higher of student or parental expectations.

```
## add new column
df_tmp$high_expct <- ifelse(df_tmp$x1stuedexpct > df_tmp$x1paredexpct, # test
                            df_tmp$x1stuedexpct,                       # if TRUE
                            df_tmp$x1paredexpct)                       # if FALSE

## show first 10 rows
head(df_tmp, n = 10)
```

```
##    stu_id x1stuedexpct x1paredexpct x1region high_expct
## 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
```

Again, our "ocular test" shows that this doesn't handle `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

## NB: We have to include [is.na(df_tmp$high_expct)] each time so that
## everything lines up

## step 1 student
df_tmp$high_expct[is.na(df_tmp$high_expct)] <- ifelse(
    ## test
    !is.na(df_tmp$x1stuedexpct[is.na(df_tmp$high_expct)]),
    ## if TRUE do this...
    df_tmp$x1stuedexpct[is.na(df_tmp$high_expct)],
    ## ... else do that
```

```
    df_tmp$high_expct[is.na(df_tmp$high_expct)]
)

## step 2 parent
df_tmp$high_expct[is.na(df_tmp$high_expct)] <- ifelse(
    ## test
    !is.na(df_tmp$x1paredexpct[is.na(df_tmp$high_expct)]),
    ## if TRUE do this...
    df_tmp$x1paredexpct[is.na(df_tmp$high_expct)],
    ## ... else do that
    df_tmp$high_expct[is.na(df_tmp$high_expct)]
)
```

That's a lot of text! What's happening is that we are trying to replace a vector of values with another vector of values, which need to line up and be the same length. That's why we have

```
## what we'll use to replace when TRUE
df_tmp$x1stuedexpct[is.na(df_tmp$high_expct)]
```

When our `high_expct` column has missing values, we want to replace with non-missing `x1stuedexpct` values *in the same row*. That means we also need to subset that column to only include values in rows that have missing `high_expct` values. Because we must do this each time, our script gets pretty long and unwieldy.

Let's check to make sure it worked as intended.

```
## show first 10 rows
head(df_tmp, n = 10)
```

```
##    stu_id x1stuedexpct x1paredexpct x1region high_expct
## 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
```

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.

Filter observations (rows)

Let's check the counts of our new variable:

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

## get summary of our new variable
table(df_tmp$high_expct, useNA = "ifany")
```

```
##
##    1    2    3    4    5    6    7    8    9   10 <NA>
##   71 2034  163 1282  132 4334  191 5087  168 6578 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.

As when we selected columns above, we'll use the square brackets notation. As with dplyr's `filter()`, we want to *filter in* what we want. We set this condition before the comma in the square brackets. Because we want all the columns, we leave the space after the comma blank.

```
## filter in values that aren't missing
df_tmp <- df_tmp[!is.na(df_tmp$high_expct),]

## show first 10 rows
head(df_tmp, n = 10)
```

```
##    stu_id x1stuedexpct x1paredexpct x1region high_expct
## 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
## 10  10010            8            6        1          8
## 11  10011            8            6        3          8
```

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. For an overall average, we can just use the `mean()` function.

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

## get average (without storing)
mean(df_tmp$high_expct)
```

```
## [1] 7.272705
```

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, useNA = "ifany")
```

```
##
##    1    2    3    4
## 3128 5312 8177 3423
```

We're not missing any census data, which is good. To calculate our average expectations, we need to use the `aggregate` function. This function allows to compute a `FUN`ction `by` a group. We'll use it to get our summary.

```
## get average (assigning this time)
df_tmp <- aggregate(df_tmp["high_expct"],               # var of interest
                    by = list(region = df_tmp$x1region), # by group
                    FUN = mean)                          # function to run

## show
df_tmp
```

```
##   region high_expct
## 1      1   7.389066
## 2      2   7.168110
## 3      3   7.357833
## 4      4   7.125329
```

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

Arrange data

As our final step, we'll arrange our data frame from highest to lowest (descending). For this, we'll use `sort()` and the `decreasing` option.

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

## arrange from highest expectations (first row) to lowest
df_tmp$high_expct <- sort(df_tmp$high_expct, decreasing = TRUE)

## show
df_tmp
```

```
##   region high_expct
## 1      1   7.389066
## 2      2   7.357833
## 3      3   7.168110
## 4      4   7.125329
```

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.