Week 8 Walkthrough: Column Manipulations with `dplyr`

2025-02-21

Functions Covered

In this walkthrough, we'll be focusing on the following functions:

- `select()`: **Selects** which columns to keep in (or drop from) the dataset
  - We'll generally pick them out by column **name** or column **number**. The order we specify them will be the order they come back in!
- `rename()`: **Renames** columns
  - We've seen other ways to **rename** columns, but the convenient thing here is we *don't* have to know the order of the columns, we can go entirely by names if we want to!
- `%>%`: Our **chaining** or **piping** operator, lets us keep pusing results through a series of functions
  - The best way to read this is as something like "**and then**". You *can* just read it as `percent greater than percent`, but seeing as that means nothing to anyone, just looking at it as being the same as saying "**and then**" will probably be a little more convenient.

`dplyr`

In case you didn't read over the walkthrough on row manipulations, I'll talk again (briefly) about the `dplyr` package (pronounced DEE-ply-er)!

The `dplyr` package is an important part of the `tidyverse` and lets us do a lot of convenient **data manipulations**. It also gives us `as_tibble()`, a way to make our **data frames** a little prettier. Using square brackets to subset our data works just the same as ever, but we'll teach some other (more easily understandable) tools for these manipulations.

For everything we're doing in this walkthrough, we will use the `dplyr` package, so unless you want to start every function call with `dplyr::` (hint: you don't), make sure you open the `dplyr` library with `library(dplyr)`!

```
library(dplyr)
```

`select()`

When modifying data (*especially* modifying it to share!), controlling which columns are in the data can be a pretty big deal! If we've collected a bunch of data full of personally identifiable information (PII) such as addresses, phone numbers, full names, place of business, etc. then the people who graciously provided us with this information would be kind of upset if we were handing it out all willy-nilly! The `select()` function helps us with that. With `select()` we can knock off all of the columns with PII and I can cover my `liability` when sharing the data.

In a less liability-fueled vision of column **selecting**, sometimes there's are a bunch of columns that we quite frankly don't care about. If you have medical data and want to look at differences in blood tests based on blood type, you don't really care about columns listing parents' blood types, marital status, favorite ice cream, etc. These things just don't factor in and clutter up our dataset, so we can filter down to just what we *do* care about!

Let's look at some examples. I'll use a food dataset I found on Kaggle[1]. Let's start off by reading it in and looking at the sheer size of it all. For the sake of brevity, I'll just print the number of columns in `yum`.

```
library(readr)
yum<-read_csv('food_coded.csv')
ncol(yum)
```

```
## [1] 61
```

61 dang columns (that's **inline code** by the way)! That's quite a few, with names ranging in length from 3 to 31 (more inline code!), so this dataset leaves a lot to be desired in terms of ease-of-use. If some of my columns are 31 characters long, I do *not* want to type that! And if I rely on tab-completion, I similarly do *not* want to scroll through 61 options to find the one I'm looking for. We'll get to the length problem in a second, but with `select()` we can at least pare it down to only what we're interested in. With that said, there's a lot going on in this dataset, so what *am* I interested in? Well, as a student myself I'll focus on the things that come into my daily life:

- `GPA`: Grade Point Average
  - We're students so why not?
- `breakfast`: Whether the respondent eats breakfast
  - I'm often in a rush in the morning, so it's my most-skipped meal by a long shot.
- `coffee`: Frequency of coffee consumption
  - I just got some new coffee beans[2] and they're *so* good.
- `cook`: Whether the respondent cooks for themselves
  - The busier I get, the less I make time to cook for myself. Maybe that matters?
- `exercise`: Frequency of exercise
  - Diet and exercise are two sides of the same coin when it comes to health! If we just focus on food (yum...), we'll miss this other important aspect.

So now let's use `select()` to focus in on *only* these columns and knock out the rest. Before we do that, we'll learn a *little* more about the syntax of `select()` so we can understand what we're seeing. The `select()` function can take any number of **arguments**, but really only takes two *kinds* of **arguments**: the dataset and column names. The first argument should always be the dataset (unless using the chaining operator `%>%`, more on this later), then separated by commas are all of our column names. Interestingly, you can specify the column names with **or without** quotation marks! You actually don't even have to be consistent, you can alternate back and forth if you want[3]. So if we wanted to **select** only these listed columns from `yum` then it would look like the below. If you haven't already, make sure you open `dplyr` here!

```
head(select(yum,GPA,"breakfast",coffee,"cook",exercise))
```

```
## # A tibble: 6 x 5
##   GPA   breakfast coffee  cook exercise
##   <chr>     <dbl>  <dbl> <dbl>    <dbl>
## 1 2.4           1      1     2        1
## 2 3.654         1      2     3        1
## 3 3.3           1      2     1        2
## 4 3.2           1      2     2        3
## 5 3.5           1      2     1        1
## 6 2.25          1      2     3        2
```

And interestingly if I change the order of the variables in `select()` I change their order in the output data! A very convenient way to reorder data.

[1] https://www.kaggle.com/datasets/markmedhat/food-dataset

[2] Cup-A-Joe on Hillsborough St. has this Ethiopian coffee that I can't get enough of, just bought my mom another pound of beans too.

[3] This might make your TA upset because of how strange it looks, but you decide on your own style!

```
head(select(yum,GPA,exercise,coffee,"breakfast","cook"))
```

```
## # A tibble: 6 x 5
##   GPA   exercise coffee breakfast  cook
##   <chr>    <dbl>  <dbl>     <dbl> <dbl>
## 1 2.4          1      1         1     2
## 2 3.654        1      2         1     3
## 3 3.3          2      2         1     1
## 4 3.2          3      2         1     2
## 5 3.5          1      2         1     1
## 6 2.25         2      2         1     3
```

As another point, if I wanted to keep *all* the variables but just wanted to make these variables of interest the first ones in my dataset, there's a convenient way to do that, too! If I was feeling particularly Sisyphean I could list out every column one-by-one until I cry, give up, or reach the end but on a normal day where I prefer doing *less* unnecessary work, we have the `everything()` function that says simply "everything else"!

```
head(select(yum,GPA,exercise,coffee,"breakfast","cook",everything()))
```

```
## # A tibble: 6 x 61
##   GPA   exercise coffee breakfast  cook Gender calories_chicken calories_day
##   <chr>    <dbl>  <dbl>     <dbl> <dbl>  <dbl>            <dbl>        <dbl>
## 1 2.4          1      1         1     2      2              430          NaN
## 2 3.654        1      2         1     3      1              610            3
## 3 3.3          2      2         1     1      1              720            4
## 4 3.2          3      2         1     2      1              430            3
## 5 3.5          1      2         1     1      1              720            2
## 6 2.25         2      2         1     3      1              610            3
## # i 53 more variables: calories_scone <dbl>, comfort_food <chr>,
## #   comfort_food_reasons <chr>, comfort_food_reasons_coded...10 <dbl>,
## #   comfort_food_reasons_coded...12 <dbl>, cuisine <dbl>, diet_current <chr>,
## #   diet_current_coded <dbl>, drink <dbl>, eating_changes <chr>,
## #   eating_changes_coded <dbl>, eating_changes_coded1 <dbl>, eating_out <dbl>,
## #   employment <dbl>, ethnic_food <dbl>, father_education <dbl>,
## #   father_profession <chr>, fav_cuisine <chr>, fav_cuisine_coded <dbl>, ...
```

Neat! Now if I want all of the data *except* some specific columns, I can use the "not" operator, `!`, to tell `R` which to leave off. If I've decided I no longer care about `GPA` because I'm tired of worrying about grades and they are *not* yummy, I can still have everything else and have it in the order I want with one little change, a `!` in front of `GPA`. **Note here that order becomes very important!** If I leave `!GPA` at the front, the `!` carries through and removes all those other variables I wanted leaving only the `everything()`. If I put `!GPA` at the end, `R` will have already included `everything()` and it will be too late to knock it out! So when specifying which columns we *do not* want alongside those which we *do*, we need to put the "not"s (`!`) between the "do"s and the `everything()`.

```
head(select(yum,exercise,coffee,"breakfast","cook",!GPA,everything()))
```

```
## # A tibble: 6 x 61
##   exercise coffee breakfast  cook Gender calories_chicken calories_day
##      <dbl>  <dbl>     <dbl> <dbl>  <dbl>            <dbl>        <dbl>
## 1        1      1         1     2      2              430          NaN
## 2        1      2         1     3      1              610            3
## 3        2      2         1     1      1              720            4
## 4        3      2         1     2      1              430            3
## 5        1      2         1     1      1              720            2
## 6        2      2         1     3      1              610            3
```

```
## # i 54 more variables: calories_scone <dbl>, comfort_food <chr>,
## #   comfort_food_reasons <chr>, comfort_food_reasons_coded...10 <dbl>,
## #   comfort_food_reasons_coded...12 <dbl>, cuisine <dbl>, diet_current <chr>,
## #   diet_current_coded <dbl>, drink <dbl>, eating_changes <chr>,
## #   eating_changes_coded <dbl>, eating_changes_coded1 <dbl>, eating_out <dbl>,
## #   employment <dbl>, ethnic_food <dbl>, father_education <dbl>,
## #   father_profession <chr>, fav_cuisine <chr>, fav_cuisine_coded <dbl>, ...
```

Now before we move on let's hit on a *really* common error. Pretend I only want to remove `fav_cuisine_coded` because it feels kind of redundant. If we choose to overwrite `yum` when dropping this column, it would look like the below.

```
yum<-select(yum,!fav_cuisine_coded)
```

Easy-peezy, no problems there. But if as I'm toying around with my code I were to run that line again, what would happen?

```
yum<-select(yum,!fav_cuisine_coded)
```

```
## Error in `select()`:
## ! Can't select columns that don't exist.
## x Column `fav_cuisine_coded` doesn't exist.
```

Out comes the error! The error says that `fav_cuisine_coded` is not a column in our data! This code ran just fine a second ago, but now it's hitting an error because *we already removed that column*. We can't remove it twice and so when we try `R` throws an error, giving us the chance to maybe say "Oh, I meant a different column" or just understand the work has already been done. Watch out for this error! It *will* come for you or someone you know, just give it time.

`rename()`

So we've addressed how to handle the 60 columns problem, but what about the long variable names? This is kind of a good-news-bad-news situation. The good news is, we can make those names shorter and really make them anything we want! The bad news is we still have to type our ugly column names one last time (or get really fancy with it).

First off, let's change `GPA` and `Gender` to a lowercase version because I don't like hitting `Shift` (or `Caps Lock`). Now when we're doing the renames I *am* going to overwrite our existing `yum` so I can call our new names later. The syntax is similar to `select()`, just a bunch of arguments. In this case instead of column names to keep/drop, we have arguments of the form `newName=oldName`. The `dplyr` functions are really flexible, so you can put `newName` or `oldName` in quotation marks if you want, or leave them off! See the code below where I make it look really weird but it still works!

```
yum<-rename(yum,gpa='GPA',
                'gender'=Gender)
```

Well, that was easy. What else is there to do with `rename()`? All the syntax really follows the same way, so I'll just set out explicitly the way this syntax flows and then we'll move on to another topic. Remember, it is always the *new* name on the left of the equals sign `=`!

```
rename(DATA_FRAME,
       new_Name_1='old_Name_1',
       new_Name_2='old_Name_2',
       easyName='compLicAted_old_naME',
       fun.Name_123_='boring.name',
       with_underscores='camelCase') #I do like camelCase, but other people don't!
```

Now, on to chaining!