Data manipulation in R

April 18, 2018

Contents

Introduction and background

Today we are going to be learning how to perform basic data manipulation tasks in R. While there are many options for tackling data manipulation problems in R (e.g., `apply` family, `data.table` package, functions `ave` and `aggregate`), we will be working with the **dplyr** and **tidyr** packages today. I find that these packages are approachable for people without a lot of programming background but are still quite fast when working with large datasets.

In this workshop, we will cover the following:

- Making summary datasets by group
- Filtering the dataset to include only rows that satisfy certain conditions
- Selecting only some columns/variables in a dataset
- Adding new variables/columns
- Sorting datasets based on variables
- Reshaping datasets
- Merging or *joining* two datasets

The workshop is broken up into four parts:

In Part 1, we'll review functions from **dplyr** for basic data manipulation/munging/cleaning.

In Part 2, you'll be introduced to the concept of *reshaping* datasets via **tidyr** functions.

In Part 3 we'll practice joining datasets using the `join` functions from **dplyr**.

And finally, in Part 4, you'll have a chance to practice some of the functions you learned.

Where to find help

It is important to know where to go for help when you run into data manipulation problems. The first place to start is the help pages for the functions themselves; too often folks skip this step and end up in a time-consuming search that could have been avoided. Another place that I often go to find help is on the Stack Overflow website: http://stackoverflow.com/questions/tagged/r. I've given you the link to questions that are specifically R programming questions. You could also look for questions tagged with **dplyr** or **tidyr** or search all R-related questions using keywords or phrases.

Both of these packages are fairly young, and while they are stabilizing, some elements of the packages may still change. Functions we are using today, however, are functions that are already stable and likely won't change much through time.

Both packages have introductory vignettes that are useful.

The **Introduction to dplyr** vignette is updated as **dplyr** is updated, and is nice resource: https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html.

Also see the **Tidy data** vignette for some examples using **tidyr**: https://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html.

The RStudio cheat sheets may also be helpful: https://www.rstudio.com/resources/cheatsheets/

Getting started

Check package version

The current version of **dplyr** is 0.7.4 and the current version of **tidyr** is 0.8.0.
We can use `packageVersion` to check for the currently installed version of a package, to make sure we are using the current versions.

```
packageVersion("dplyr")
```

```
[1] '0.7.4'
```

```
packageVersion("tidyr")
```

```
[1] '0.8.0'
```

If one of these packages isn't up to date, you need to re-install it. You can install via coding using, e.g., `install.packages("tidyr")` or via the RStudio Packages pane Install button. Remember that you do not need to install a package every time you use it, so don't make this code part of a script.

In between version releases, bugs are fixed and new issues addressed in the *development version* of a package. For these two packages, you can see the changes, check for known issues, and download the current development version via their github repositories. For **dplyr** see https://github.com/tidyverse/dplyr and for **tidyr** see https://github.com/tidyverse/tidyr.

Load packages

If all packages are up-to-date, we can load **dplyr** and **tidyr** and get started.

```
library(dplyr)
library(tidyr)
```

The `mtcars` dataset

In the first part of the workshop we will be using the `mtcars` dataset to practice data manipulation. This dataset comes with R, and information about this dataset is available in the R help files for the dataset (`?mtcars`).

We will be using both categorical and continuous variables from this dataset, including, `mpg` (Miles per US gallon), `wt` (car weight in 1000 lbs), `cyl` (number of cylinders), `am` (type of transmission), `disp` (engine displacement), `qsec` (quarter mile time), and `hp` (horsepower).

Let's take a quick look at the first six lines (with `head`) and structure (with `str`) of this dataset. You should recognize that `cyl` and `am` (as well as others like `vs`) are categorical variables but that they are considered numeric variables in the dataset since the categories are expressed with numbers.

```
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
str(mtcars)
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Part 1: Functions for basic data manipulation

Calculating summary statistics by group

We're going to start out today by learning how to calculate summary statistics by group. I start here because this is common task that I see folks struggle with in R. The task of calculating summaries by groups in R is referred to as a *split-apply-combine* task because we want to split the dataset into groups, apply a function to each split, and then combine the results back into a single dataset.

There are a variety of ways to perform such tasks in R, including `tapply` and `by` from the `apply` family of functions, the `aggregate` and `ave` functions, the **data.table** package, and functions from **dplyr**. We will be using **dplyr** today but in the long run you may find you like the style of another method better.

Using the `group_by` function

With **dplyr**, the key to split-apply-combine tasks is *grouping*. We need to define which variable contains the groups we want separate summaries for. We create a grouped dataset using the `group_by` function.

Let's create a grouped dataset named `bycyl`, where we group `mtcars` by the `cyl` variable. The `cyl` variable is a categorical variable representing the number of cylinders a car has. This variable has 3 different levels, `4`, `6`, and `8`.

```
bycyl = group_by(mtcars, cyl)
```

We can see that the new object is a grouped dataset if we print the `head` of the dataset and see the `Groups` tag or see the class `grouped_df` in the object structure.

```
head(bycyl)
```

```
# A tibble: 6 x 11
# Groups: cyl [3]
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21.0  6.00   160 110    3.90  2.62  16.5  0     1.00  4.00  4.00
2  21.0  6.00   160 110    3.90  2.88  17.0  0     1.00  4.00  4.00
3  22.8  4.00   108  93.0  3.85  2.32  18.6  1.00  1.00  4.00  1.00
4  21.4  6.00   258 110    3.08  3.22  19.4  1.00  0     3.00  1.00
5  18.7  8.00   360 175    3.15  3.44  17.0  0     0     3.00  2.00
6  18.1  6.00   225 105    2.76  3.46  20.2  1.00  0     3.00  1.00
```

```
str(bycyl)
```

```
Classes 'grouped_df', 'tbl_df', 'tbl' and 'data.frame': 32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
 - attr(*, "vars")= chr "cyl"
 - attr(*, "drop")= logi TRUE
 - attr(*, "indices")=List of 3
  ..$ : int  2 7 8 17 18 19 20 25 26 27 ...
  ..$ : int  0 1 3 5 9 10 29
  ..$ : int  4 6 11 12 13 14 15 16 21 22 ...
 - attr(*, "group_sizes")= int  11 7 14
 - attr(*, "biggest_group_size")= int 14
 - attr(*, "labels")='data.frame':  3 obs. of  1 variable:
  ..$ cyl: num  4 6 8
  ..- attr(*, "vars")= chr "cyl"
  ..- attr(*, "drop")= logi TRUE
```

Using the summarise function

Now that we have a grouped dataset, we can use it with the `summarise` function to calculate summary statistics by group (`summarize` is an alternative spelling for the same function).

We'll start by calculating the mean engine displacement for each cylinder category. We will be working on the grouped dataset `bycyl` as we want summaries by groups.

Notice that the first argument of `summarise` is the dataset we want summarized. This is true for most of the **dplyr** functions. We list the summary function and variable we want summarized as the second argument.

```
summarise( bycyl, mean(disp) )
```

```
# A tibble: 3 x 2
    cyl `mean(disp)`
  <dbl>        <dbl>
1  4.00          105
2  6.00          183
```

```
3  8.00          353
```

Notice that we printed the summarized dataset to the R Console but did not name the resulting object. This is what we will be doing for most of the workshop, as my goal is to show you what happens to the dataset after we manipulate it. You certainly can (and likely will want to) name your final datasets. We'll see some examples of naming the new objects once we are doing multiple data manipulations tasks at one time.

Summarizing multiple variables in `summarise`

We can summarize multiple variables at once or use different summary functions on variables in `summarise` by simply using commas to separate each new function/variable.
For example, we can calculate the mean of engine displacement and horsepower by cylinder category in the same function call.

```
summarise( bycyl, mean(disp), mean(hp) )
```

```
# A tibble: 3 x 3
    cyl `mean(disp)` `mean(hp)`
  <dbl>        <dbl>      <dbl>
1  4.00          105       82.6
2  6.00          183      122
3  8.00          353      209
```

Naming the variables in `summarise`

The default names for the new variables we've been calculating are sufficient for a quick summary but are not particularly convenient if we wanted to use the result for anything further in R. We can set variable names as we summarize.

Let's calculate the mean and standard deviation of engine displacement by cylinder category and name the new variables `mdisp` and `sdisp`, respectively.

```
summarise( bycyl, mdisp = mean(disp), sdisp = sd(disp) )
```

```
# A tibble: 3 x 3
    cyl mdisp sdisp
  <dbl> <dbl> <dbl>
1  4.00   105  26.9
2  6.00   183  41.6
3  8.00   353  67.8
```

Grouping a dataset by multiple variables

Datasets can be grouped by multiple variables as well as by a single variable. This is common for studies with multiple factors of interest or with nested studies designs (e.g., plots nested in transects nested in sites).

Let's group `mtcars` by both `cyl` and `am` (transmission type) and then calculate the mean engine displacement. We get the summary statistic for every factor combination, for a total of six rows (3 `cyl` categories and 2 `am` categories).

```
byam.cyl = group_by(mtcars, cyl, am)

summarise( byam.cyl, mdisp = mean(disp) )
```

```
# A tibble: 6 x 3
# Groups: cyl [?]
    cyl    am mdisp
  <dbl> <dbl> <dbl>
1  4.00  0    136
2  4.00  1.00  93.6
3  6.00  0    205
4  6.00  1.00 155
5  8.00  0    358
6  8.00  1.00 326
```

Ungrouping a dataset

Looking at our last result, we can see the dataset is still grouped by the `cyl` variable (i.e., `cyl` is listed in "Groups"). If we are finished with any data manipulation we should *ungroup* the dataset. Trying to work with a dataset that is grouped when we don't want it to be can cause problems. It is "safest" to make sure the final version is ungrouped.

Ungrouping is done via the `ungroup` function. Notice we no longer have any `Groups` listed in the output once we do this, as the result is no longer grouped by any variables.

```
ungroup( summarise( byam.cyl, mdisp = mean(disp) ) )
```

```
# A tibble: 6 x 3
    cyl    am mdisp
  <dbl> <dbl> <dbl>
1  4.00  0    136
2  4.00  1.00  93.6
3  6.00  0    205
4  6.00  1.00 155
5  8.00  0    358
6  8.00  1.00 326
```

Summarizing multiple variables at once

When we want to summarize many variables in a dataset using the same function, we can use one of the `summarise_*` functions. The `summarise_*` functions are `summarise_all`, `summarise_at`, and `summarise_if`.

summarise_all

The `summarise_all` function is useful when we want to summarize every non-grouping variable in the dataset with the same function. We give the function we want to use for the summaries as the second argument.

Let's see how `summarise_all` works by calculating the mean of every variable in `mtcars` for each cylinder category.

```
summarise_all(bycyl, mean)
```

```
# A tibble: 3 x 11
    cyl   mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  4.00  26.7   105  82.6  4.07  2.29  19.1 0.909 0.727  4.09  1.55
2  6.00  19.7   183 122    3.59  3.12  18.0 0.571 0.429  3.86  3.43
3  8.00  15.1   353 209    3.23  4.00  16.8 0     0.143  3.29  3.50
```

Note that we need to be careful with `summarise_all`. We could have problems if trying to summarize both continuous and categorical variables in a single dataset and could end up with an error. All the variables in `mtcars` are currently numeric. What would happen if we made one of the variables a factor and tried to take the mean of every variable?

```
mtcars$vs = factor(mtcars$vs)
```

R still does the averaging, but returns `NA` and warning messages for the `vs` column.

```
summarise_all(bycyl, mean)
```

```
# A tibble: 3 x 11
    cyl   mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  4.00  26.7   105  82.6  4.07  2.29  19.1 0.909 0.727  4.09  1.55
2  6.00  19.7   183 122    3.59  3.12  18.0 0.571 0.429  3.86  3.43
3  8.00  15.1   353 209    3.23  4.00  16.8 0     0.143  3.29  3.50
```

summarise_at

We won't always want to summarize every column in a dataset, for reasons including having a mix of variable types. One option to only summarize some of the variables is to use `summarise_at`, where we can list a subset of columns to summarize by name.

If you look at the help page, you can see that we list the variable to summarize within `vars` as the second argument before defining the function we want summarize with.

```
summarise_at(bycyl, vars(disp, wt), mean)
```

```
# A tibble: 3 x 3
    cyl  disp    wt
  <dbl> <dbl> <dbl>
1  4.00   105  2.29
2  6.00   183  3.12
3  8.00   353  4.00
```

We can also drop out the variables we don't want summarized rather than writing out the ones we do want. For example, while all the variables in `mtcars` are read as numeric, some are actually categorical. If we don't want to treat them as continuous, we can drop them from the summary. Let's drop `am` and `vs` from our summary dataset. We can do this by using the minus sign with the variable names inside `vars`.

We will talk more about selecting and dropping specific variables later today when we talk about the `select` function.

```
summarise_at(bycyl, vars(-am, -vs), mean)
```

```
# A tibble: 3 x 9
    cyl   mpg  disp    hp  drat    wt  qsec  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  4.00  26.7   105  82.6  4.07  2.29  19.1  4.09  1.55
2  6.00  19.7   183 122    3.59  3.12  18.0  3.86  3.43
3  8.00  15.1   353 209    3.23  4.00  16.8  3.29  3.50
```

####summarise_if

If we want to choose the columns we want to summarize using a logical *predicate* function, we can use `summarise_if`. You can see on the help page that the predicate function is the second argument, followed by the summary functions.

Here, we'll only summarize the numeric variables by using the predicate function `is.numeric`. Essentially, R checks if a column is numeric with `is.numeric` and if the result is `TRUE` a summary of the column is made. If the result is `FALSE`, the variable is dropped from the output.

In this example, all variables except `vs` are numeric and will be summarized.

```
summarise_if(bycyl, is.numeric, mean)
```

```
# A tibble: 3 x 11
    cyl   mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  4.00  26.7   105  82.6  4.07  2.29  19.1 0.909 0.727  4.09  1.55
2  6.00  19.7   183 122    3.59  3.12  18.0 0.571 0.429  3.86  3.43
3  8.00  15.1   353 209    3.23  4.00  16.8 0     0.143  3.29  3.50
```

Summarizing many variables using multiple functions

If we want to summarize many variables with multiple functions, we list all the functions we want to use within `funs` with commas between them.

For example, maybe we want to calculate both the mean and the maximum for all numeric variables by group. The functions we use are `mean` and `max`.

While the only example we see today is using `summarise_if`, this can be done in any of the `summarise_*` functions.

```
summarise_if( bycyl, is.numeric, funs(mean, max) )
```

```
# A tibble: 3 x 21
    cyl mpg_m~ disp_~ hp_m~ drat~ wt_m~ qsec~ vs_m~ am_m~ gear~ carb~ mpg_~ disp~ hp_m~ drat~ wt_m~
  <dbl>  <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  4.00   26.7    105  82.6  4.07  2.29  19.1 0.909 0.727  4.09  1.55  33.9   147   113  4.93  3.19
2  6.00   19.7    183 122    3.59  3.12  18.0 0.571 0.429  3.86  3.43  21.4   258   175  3.92  3.46
3  8.00   15.1    353 209    3.23  4.00  16.8 0     0.143  3.29  3.50  19.2   472   335  4.22  5.42
# ... with 5 more variables: qsec_max <dbl>, vs_max <dbl>, am_max <dbl>, gear_max <dbl>, carb_max
#   <dbl>
```

Notice that the name of the function is appended to the variable name when using multiple functions. To control what name is appended, you can assign names to each function within `funs`.

```
summarise_if( bycyl, is.numeric, funs(mn = mean, mx = max) )
```

```
# A tibble: 3 x 21
    cyl mpg_mn disp_~ hp_mn drat~ wt_mn qsec~ vs_mn am_mn gear~ carb~ mpg_~ disp~ hp_mx drat~ wt_mx
  <dbl>  <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  4.00   26.7    105  82.6  4.07  2.29  19.1 0.909 0.727  4.09  1.55  33.9   147   113  4.93  3.19
2  6.00   19.7    183 122    3.59  3.12  18.0 0.571 0.429  3.86  3.43  21.4   258   175  3.92  3.46
3  8.00   15.1    353 209    3.23  4.00  16.8 0     0.143  3.29  3.50  19.2   472   335  4.22  5.42
# ... with 5 more variables: qsec_mx <dbl>, vs_mx <dbl>, am_mx <dbl>, gear_mx <dbl>, carb_mx <dbl>
```

The `glimpse` function for checking wide datasets

The **dplyr** package truncates how much of the dataset we see printed into the R Console. For very wide datasets like the one we just created, we can get a better idea of what the result looks like using `glimpse`.

```
glimpse( summarise_if( bycyl, is.numeric, funs(mean, max) ) )
```

```
Observations: 3
Variables: 21
$ cyl       <dbl> 4, 6, 8
$ mpg_mean  <dbl> 26.66364, 19.74286, 15.10000
$ disp_mean <dbl> 105.1364, 183.3143, 353.1000
$ hp_mean   <dbl> 82.63636, 122.28571, 209.21429
$ drat_mean <dbl> 4.070909, 3.585714, 3.229286
$ wt_mean   <dbl> 2.285727, 3.117143, 3.999214
$ qsec_mean <dbl> 19.13727, 17.97714, 16.77214
$ vs_mean   <dbl> 0.9090909, 0.5714286, 0.0000000
$ am_mean   <dbl> 0.7272727, 0.4285714, 0.1428571
$ gear_mean <dbl> 4.090909, 3.857143, 3.285714
$ carb_mean <dbl> 1.545455, 3.428571, 3.500000
$ mpg_max   <dbl> 33.9, 21.4, 19.2
$ disp_max  <dbl> 146.7, 258.0, 472.0
$ hp_max    <dbl> 113, 175, 335
$ drat_max  <dbl> 4.93, 3.92, 4.22
$ wt_max    <dbl> 3.190, 3.460, 5.424
$ qsec_max  <dbl> 22.90, 20.22, 18.00
$ vs_max    <dbl> 1, 1, 0
$ am_max    <dbl> 1, 1, 1
$ gear_max  <dbl> 5, 5, 5
$ carb_max  <dbl> 2, 6, 8
```

Filtering datasets wtih `filter`

Now we will cover functions for other common data manipulation tasks, starting with *filtering*. Filtering involves making specific subsets of interest by removing unwanted rows based on logical conditions. Filtering is about how many rows we want in the dataset, not about the number of columns.

For example, maybe we want to focus on a subset of the dataset that only involves cars with automatic transmissions. We can do this with the `filter` function to *filter* the `mtcars` dataset to only those rows where `am` is `0`.

Like other **dplyr** functions, the dataset is the first argument in `filter`. The subsequent arguments are the conditions that the filtered dataset should meet. Here, the condition is that cars must have automatic transmissions, or `am == 0` (note the *two* equals signs).

```
filter(mtcars, am == 0)
```

```
    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
2  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
3  18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
4  14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
5  24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
6  22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
7  19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
8  17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
9  16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
10 17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
11 15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
12 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
13 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
14 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
```

```
15 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
16 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
17 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
18 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
19 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
```

The `filter` function will always be used with logical operators such as `==` (testing for equality), `!=` (testing for inequality), `<` (less than), `is.na` (all `NA` values), `!is.na` (all values except `NA`), `>=` (greater than or equal to), etc.

If we wanted to filter out all cars that weigh more than 4000 lbs (i.e., 4 1000 lbs), we can keep only the rows where `wt <= 4`.

```
filter(mtcars, wt <= 4)
```

```
# A tibble: 28 x 11
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>
 1  21.0  6.00   160 110    3.90  2.62  16.5 0       1.00  4.00  4.00
 2  21.0  6.00   160 110    3.90  2.88  17.0 0       1.00  4.00  4.00
 3  22.8  4.00   108  93.0  3.85  2.32  18.6 1       1.00  4.00  1.00
 4  21.4  6.00   258 110    3.08  3.22  19.4 1       0     3.00  1.00
 5  18.7  8.00   360 175    3.15  3.44  17.0 0       0     3.00  2.00
 6  18.1  6.00   225 105    2.76  3.46  20.2 1       0     3.00  1.00
 7  14.3  8.00   360 245    3.21  3.57  15.8 0       0     3.00  4.00
 8  24.4  4.00   147  62.0  3.69  3.19  20.0 1       0     4.00  2.00
 9  22.8  4.00   141  95.0  3.92  3.15  22.9 1       0     4.00  2.00
10  19.2  6.00   168 123    3.92  3.44  18.3 1       0     4.00  4.00
# ... with 18 more rows
```

Alternatively, we could achieve the same thing by choosing everything that is *not* greater than 4, `!wt > 4`. The exclamation point, `!`, is the *not* operator.

```
filter(mtcars, !wt > 4)
```

```
# A tibble: 28 x 11
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>
 1  21.0  6.00   160 110    3.90  2.62  16.5 0       1.00  4.00  4.00
 2  21.0  6.00   160 110    3.90  2.88  17.0 0       1.00  4.00  4.00
 3  22.8  4.00   108  93.0  3.85  2.32  18.6 1       1.00  4.00  1.00
 4  21.4  6.00   258 110    3.08  3.22  19.4 1       0     3.00  1.00
 5  18.7  8.00   360 175    3.15  3.44  17.0 0       0     3.00  2.00
 6  18.1  6.00   225 105    2.76  3.46  20.2 1       0     3.00  1.00
 7  14.3  8.00   360 245    3.21  3.57  15.8 0       0     3.00  4.00
 8  24.4  4.00   147  62.0  3.69  3.19  20.0 1       0     4.00  2.00
 9  22.8  4.00   141  95.0  3.92  3.15  22.9 1       0     4.00  2.00
10  19.2  6.00   168 123    3.92  3.44  18.3 1       0     4.00  4.00
# ... with 18 more rows
```

Filtering grouped datasets

We can filter grouped datasets, and the condition will be applied separately to each group. For example, maybe we want to keep only the rows where `wt` is greater than its cylinder category group mean.

```
filter( bycyl, wt > mean(wt) )
```

```
# A tibble: 13 x 11
# Groups: cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  22.8  4.00   108  93.0  3.85  2.32  18.6  1.00  1.00  4.00  1.00
 2  21.4  6.00   258 110    3.08  3.22  19.4  1.00  0     3.00  1.00
 3  18.1  6.00   225 105    2.76  3.46  20.2  1.00  0     3.00  1.00
 4  24.4  4.00   147  62.0  3.69  3.19  20.0  1.00  0     4.00  2.00
 5  22.8  4.00   141  95.0  3.92  3.15  22.9  1.00  0     4.00  2.00
 6  19.2  6.00   168 123    3.92  3.44  18.3  1.00  0     4.00  4.00
 7  17.8  6.00   168 123    3.92  3.44  18.9  1.00  0     4.00  4.00
 8  16.4  8.00   276 180    3.07  4.07  17.4  0     0     3.00  3.00
 9  10.4  8.00   472 205    2.93  5.25  18.0  0     0     3.00  4.00
10  10.4  8.00   460 215    3.00  5.42  17.8  0     0     3.00  4.00
11  14.7  8.00   440 230    3.23  5.34  17.4  0     0     3.00  4.00
12  21.5  4.00   120  97.0  3.70  2.46  20.0  1.00  0     3.00  1.00
13  21.4  4.00   121 109    4.11  2.78  18.6  1.00  1.00  4.00  2.00
```

Filtering by multiple conditions

And, of course, we can filter datasets by multiple conditions at once. If we wanted to filter the dataset to only cars with automatic transmission (`am == 0`) *and* that have weights less than or equal to 4000 lbs (`wt <=4`), we can include both conditions in `filter` separated by a comma.

```
filter(mtcars, am == 0, wt <= 4)
```

```
    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
2  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
3  18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
4  14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
5  24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
6  22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
7  19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
8  17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
9  17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
10 15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
11 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
12 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
13 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
14 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
15 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
```

While we won't see it today, if you need a logical *OR* statement you will need the `|` symbol, found on the backslash key.

Additional `filter_*` verbs

The **dplyr** package has `filter_all`, `filter_at`, and `filter_if` verbs available. These would be useful if we, e.g., wanted to apply the same filter to many columns of data.

Selecting variables with `select`

Keeping only a subset of the columns of a dataset is referred to as *selecting variables*. Often this might be for organizational reasons, where an analysis is focused on only some of many variables and so we want to create a dataset that contains only the variables of interest. Selecting is about how many columns we want to keep, not how many rows we have.

The **dplyr** function `select` makes selecting columns very easy to do. We can keep or drop variables by name (although you can also use the index number) with straightforward code.

Let's *select* only the `cyl` variable from `mtcars` (printing just the first rows to save space in this document).

```
select(mtcars, cyl)
```

```
# A tibble: 32 x 1
     cyl
 * <dbl>
 1  6.00
 2  6.00
 3  4.00
 4  6.00
 5  8.00
 6  6.00
 7  8.00
 8  4.00
 9  4.00
10  6.00
# ... with 22 more rows
```

If we want to keep all variables between (and including) `cyl` and `vs`, we indicate that with the colon, `:`.

```
select(mtcars, cyl:vs)
```

```
# A tibble: 32 x 7
     cyl  disp    hp  drat    wt  qsec vs
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr>
 1  6.00   160 110    3.90  2.62  16.5 0
 2  6.00   160 110    3.90  2.88  17.0 0
 3  4.00   108  93.0  3.85  2.32  18.6 1
 4  6.00   258 110    3.08  3.22  19.4 1
 5  8.00   360 175    3.15  3.44  17.0 0
 6  6.00   225 105    2.76  3.46  20.2 1
 7  8.00   360 245    3.21  3.57  15.8 0
 8  4.00   147  62.0  3.69  3.19  20.0 1
 9  4.00   141  95.0  3.92  3.15  22.9 1
10  6.00   168 123    3.92  3.44  18.3 1
# ... with 22 more rows
```

If we want to keep only a few columns, we can separate the desired column names with a comma. Here we select only `cyl` and `vs`.

```
select(mtcars, cyl, vs)
```

```
# A tibble: 32 x 2
     cyl vs
 * <dbl> <fctr>
 1  6.00 0
 2  6.00 0
```

```
 3  4.00 1
 4  6.00 1
 5  8.00 0
 6  6.00 1
 7  8.00 0
 8  4.00 1
 9  4.00 1
10  6.00 1
# ... with 22 more rows
```

Using the special helper functions in `select`

The `select` function has several special functions to make variable selection even easier. See the help page for `select_helpers` for a list of all of these (`?select_helpers`).

These special functions include `starts_with`, `contains`, and `ends_with`, among others. These special functions can be very useful if you have coded your variables names so that groups of them contain the same letters or numbers.

We are going to start with `starts_with`, where we select all variables with names that *start with* a lowercase `d`. Remember that R is case sensitive, so an uppercase `D` is different than a lowercase `d`.

```
select( mtcars, starts_with("d") )
```

```
# A tibble: 32 x 2
    disp  drat
 * <dbl> <dbl>
 1   160  3.90
 2   160  3.90
 3   108  3.85
 4   258  3.08
 5   360  3.15
 6   225  2.76
 7   360  3.21
 8   147  3.69
 9   141  3.92
10   168  3.92
# ... with 22 more rows
```

Or we could keep all variables that *contain* a lowercase `a` anywhere in the variable name.

```
select( mtcars, contains("a") )
```

```
# A tibble: 32 x 4
    drat    am  gear  carb
 * <dbl> <dbl> <dbl> <dbl>
 1  3.90  1.00  4.00  4.00
 2  3.90  1.00  4.00  4.00
 3  3.85  1.00  4.00  1.00
 4  3.08  0     3.00  1.00
 5  3.15  0     3.00  2.00
 6  2.76  0     3.00  1.00
 7  3.21  0     3.00  4.00
 8  3.69  0     4.00  2.00
 9  3.92  0     4.00  2.00
10  3.92  0     4.00  4.00
```

```
# ... with 22 more rows
```

We've been choosing which variables we want to keep, but we could also choose which variables we want to drop like we did with `summarise_at` earlier. We drop variables using the minus sign (`-`).

Drop the `gear` variable.

```
select(mtcars, -gear)
```

```
# A tibble: 32 x 10
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  carb
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl>
 1  21.0  6.00   160 110    3.90  2.62  16.5 0       1.00  4.00
 2  21.0  6.00   160 110    3.90  2.88  17.0 0       1.00  4.00
 3  22.8  4.00   108  93.0  3.85  2.32  18.6 1       1.00  1.00
 4  21.4  6.00   258 110    3.08  3.22  19.4 1       0     1.00
 5  18.7  8.00   360 175    3.15  3.44  17.0 0       0     2.00
 6  18.1  6.00   225 105    2.76  3.46  20.2 1       0     1.00
 7  14.3  8.00   360 245    3.21  3.57  15.8 0       0     4.00
 8  24.4  4.00   147  62.0  3.69  3.19  20.0 1       0     2.00
 9  22.8  4.00   141  95.0  3.92  3.15  22.9 1       0     2.00
10  19.2  6.00   168 123    3.92  3.44  18.3 1       0     4.00
# ... with 22 more rows
```

Drop both the `gear` and `carb` variables.

```
select(mtcars, -gear, -carb)
```

```
# A tibble: 32 x 9
     mpg   cyl  disp    hp  drat    wt  qsec vs        am
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl>
 1  21.0  6.00   160 110    3.90  2.62  16.5 0       1.00
 2  21.0  6.00   160 110    3.90  2.88  17.0 0       1.00
 3  22.8  4.00   108  93.0  3.85  2.32  18.6 1       1.00
 4  21.4  6.00   258 110    3.08  3.22  19.4 1       0
 5  18.7  8.00   360 175    3.15  3.44  17.0 0       0
 6  18.1  6.00   225 105    2.76  3.46  20.2 1       0
 7  14.3  8.00   360 245    3.21  3.57  15.8 0       0
 8  24.4  4.00   147  62.0  3.69  3.19  20.0 1       0
 9  22.8  4.00   141  95.0  3.92  3.15  22.9 1       0
10  19.2  6.00   168 123    3.92  3.44  18.3 1       0
# ... with 22 more rows
```

Drop all variables between and including `am` and `carb`.

```
select( mtcars, -(am:carb) )
```

```
# A tibble: 32 x 8
     mpg   cyl  disp    hp  drat    wt  qsec vs
 * <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr>
 1  21.0  6.00   160 110    3.90  2.62  16.5 0
 2  21.0  6.00   160 110    3.90  2.88  17.0 0
 3  22.8  4.00   108  93.0  3.85  2.32  18.6 1
 4  21.4  6.00   258 110    3.08  3.22  19.4 1
 5  18.7  8.00   360 175    3.15  3.44  17.0 0
 6  18.1  6.00   225 105    2.76  3.46  20.2 1
 7  14.3  8.00   360 245    3.21  3.57  15.8 0
 8  24.4  4.00   147  62.0  3.69  3.19  20.0 1
```

```
 9  22.8  4.00   141  95.0  3.92  3.15  22.9 1
10  19.2  6.00   168 123    3.92  3.44  18.3 1
# ... with 22 more rows
```

Drop variables that end with the letter "t".

```
select( mtcars, -ends_with("t") )
```

```
# A tibble: 32 x 9
     mpg   cyl  disp    hp  qsec vs        am  gear  carb
 * <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>
 1  21.0  6.00   160 110    16.5 0       1.00  4.00  4.00
 2  21.0  6.00   160 110    17.0 0       1.00  4.00  4.00
 3  22.8  4.00   108  93.0  18.6 1       1.00  4.00  1.00
 4  21.4  6.00   258 110    19.4 1       0     3.00  1.00
 5  18.7  8.00   360 175    17.0 0       0     3.00  2.00
 6  18.1  6.00   225 105    20.2 1       0     3.00  1.00
 7  14.3  8.00   360 245    15.8 0       0     3.00  4.00
 8  24.4  4.00   147  62.0  20.0 1       0     4.00  2.00
 9  22.8  4.00   141  95.0  22.9 1       0     4.00  2.00
10  19.2  6.00   168 123    18.3 1       0     4.00  4.00
# ... with 22 more rows
```

The `select_helpers` can be used in other functions, as well. We would commonly use them in functions like `summarise_at`, among others, to help pick the variables to use within the function.

Creating new variables with `mutate`

In **dplyr**, we can use `mutate` to create new variables and add them to the dataset as new columns. The new variable is the same length as the current dataset (in other words, it has the same number of rows as the original dataset). We will be making some new variables and adding them to `mtcars` to illustrate how this works.

Let's start by making a new variable called `disp.hp`, which is the sum of engine displacement (`disp`) and horsepower (`hp`).

As with the other **dplyr** functions, the dataset is the first argument of `mutate`.

```
mutate(mtcars, disp.hp = disp + hp)
```

```
# A tibble: 32 x 12
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb disp.hp
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>   <dbl>
 1  21.0  6.00   160 110    3.90  2.62  16.5 0       1.00  4.00  4.00     270
 2  21.0  6.00   160 110    3.90  2.88  17.0 0       1.00  4.00  4.00     270
 3  22.8  4.00   108  93.0  3.85  2.32  18.6 1       1.00  4.00  1.00     201
 4  21.4  6.00   258 110    3.08  3.22  19.4 1       0     3.00  1.00     368
 5  18.7  8.00   360 175    3.15  3.44  17.0 0       0     3.00  2.00     535
 6  18.1  6.00   225 105    2.76  3.46  20.2 1       0     3.00  1.00     330
 7  14.3  8.00   360 245    3.21  3.57  15.8 0       0     3.00  4.00     605
 8  24.4  4.00   147  62.0  3.69  3.19  20.0 1       0     4.00  2.00     209
 9  22.8  4.00   141  95.0  3.92  3.15  22.9 1       0     4.00  2.00     236
10  19.2  6.00   168 123    3.92  3.44  18.3 1       0     4.00  4.00     291
# ... with 22 more rows
```

We can make multiple new variables at once, separating each new variable by a comma like we did in `summarise`. A handy feature of `mutate` is that we can work directly with the new variables we've made

within the same function call. For example, we could first calculate `disp.hp` and then calculate a second variable that is half of `disp.hp` (`disp.hp` divided by 2). We can create other variables, as well, so we'll create the ratio of `qsec` and `wt` while we're at it.

```
mutate(mtcars,
       disp.hp = disp + hp,
       halfdh = disp.hp/2,
       qw = qsec/wt)
```

```
# A tibble: 32 x 14
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb disp.hp halfdh    qw
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>   <dbl>  <dbl> <dbl>
 1  21.0  6.00   160 110    3.90  2.62  16.5 0       1.00  4.00  4.00     270    135  6.28
 2  21.0  6.00   160 110    3.90  2.88  17.0 0       1.00  4.00  4.00     270    135  5.92
 3  22.8  4.00   108  93.0  3.85  2.32  18.6 1       1.00  4.00  1.00     201    100  8.02
 4  21.4  6.00   258 110    3.08  3.22  19.4 1       0     3.00  1.00     368    184  6.05
 5  18.7  8.00   360 175    3.15  3.44  17.0 0       0     3.00  2.00     535    268  4.95
 6  18.1  6.00   225 105    2.76  3.46  20.2 1       0     3.00  1.00     330    165  5.84
 7  14.3  8.00   360 245    3.21  3.57  15.8 0       0     3.00  4.00     605    302  4.44
 8  24.4  4.00   147  62.0  3.69  3.19  20.0 1       0     4.00  2.00     209    104  6.27
 9  22.8  4.00   141  95.0  3.92  3.15  22.9 1       0     4.00  2.00     236    118  7.27
10  19.2  6.00   168 123    3.92  3.44  18.3 1       0     4.00  4.00     291    145  5.32
# ... with 22 more rows
```

Using `mutate` with grouped datasets

We can work with grouped datasets when using `mutate`. This is useful when we want to add a column of a summary statistic for each group to the existing dataset rather than making a summary dataset.

Let's create and add a new variable that is the mean horsepower for each cylinder category. Each car within a cylinder category will have the same value of mean horsepower, as `mutate` always returns a new dataset that is the same length as the original.

```
mutate( bycyl, mhp = mean(hp) )
```

```
# A tibble: 32 x 12
# Groups: cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb   mhp
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21.0  6.00   160 110    3.90  2.62  16.5  0     1.00  4.00  4.00 122
 2  21.0  6.00   160 110    3.90  2.88  17.0  0     1.00  4.00  4.00 122
 3  22.8  4.00   108  93.0  3.85  2.32  18.6  1.00  1.00  4.00  1.00  82.6
 4  21.4  6.00   258 110    3.08  3.22  19.4  1.00  0     3.00  1.00 122
 5  18.7  8.00   360 175    3.15  3.44  17.0  0     0     3.00  2.00 209
 6  18.1  6.00   225 105    2.76  3.46  20.2  1.00  0     3.00  1.00 122
 7  14.3  8.00   360 245    3.21  3.57  15.8  0     0     3.00  4.00 209
 8  24.4  4.00   147  62.0  3.69  3.19  20.0  1.00  0     4.00  2.00  82.6
 9  22.8  4.00   141  95.0  3.92  3.15  22.9  1.00  0     4.00  2.00  82.6
10  19.2  6.00   168 123    3.92  3.44  18.3  1.00  0     4.00  4.00 122
# ... with 22 more rows
```

As you can see, the code for `mutate` resembles the code for `summarise`. While we will not see it today, there are `mutate_all`/`mutate_at`/`mutate_if` functions that works much like the `summarise_*` functions we learned earlier.

There is also a function called `transmute`, which creates new variables that are the same length as the current

dataset like `mutate` but only returns the new variables like `summarise`.

Sorting

There are some situations where you might want to sort your dataset by variables within the dataset. For example, if we want to pull out the first observation in each group from a time series we might sort the dataset first by time within group prior to filtering. We can sort datasets with **dplyr** using `arrange`.

Here we'll start by sorting `mtcars` by `cyl`. By default we sort whatever variable we are sorting on from low to high (ascending order).

```
arrange(mtcars, cyl)
```

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>
 1  22.8  4.00 108    93.0  3.85  2.32  18.6 1       1.00  4.00  1.00
 2  24.4  4.00 147    62.0  3.69  3.19  20.0 1       0     4.00  2.00
 3  22.8  4.00 141    95.0  3.92  3.15  22.9 1       0     4.00  2.00
 4  32.4  4.00  78.7  66.0  4.08  2.20  19.5 1       1.00  4.00  1.00
 5  30.4  4.00  75.7  52.0  4.93  1.62  18.5 1       1.00  4.00  2.00
 6  33.9  4.00  71.1  65.0  4.22  1.84  19.9 1       1.00  4.00  1.00
 7  21.5  4.00 120    97.0  3.70  2.46  20.0 1       0     3.00  1.00
 8  27.3  4.00  79.0  66.0  4.08  1.94  18.9 1       1.00  4.00  1.00
 9  26.0  4.00 120    91.0  4.43  2.14  16.7 0       1.00  5.00  2.00
10  30.4  4.00  95.1 113    3.77  1.51  16.9 1       1.00  5.00  2.00
# ... with 22 more rows
```

To sort datasets by variables in descending order (highest to lowest), we can use the minus sign (`-`) or the function `desc` (which is from **dplyr**).

```
arrange(mtcars, -cyl)
```

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>
 1  18.7  8.00   360   175  3.15  3.44  17.0 0          0  3.00  2.00
 2  14.3  8.00   360   245  3.21  3.57  15.8 0          0  3.00  4.00
 3  16.4  8.00   276   180  3.07  4.07  17.4 0          0  3.00  3.00
 4  17.3  8.00   276   180  3.07  3.73  17.6 0          0  3.00  3.00
 5  15.2  8.00   276   180  3.07  3.78  18.0 0          0  3.00  3.00
 6  10.4  8.00   472   205  2.93  5.25  18.0 0          0  3.00  4.00
 7  10.4  8.00   460   215  3.00  5.42  17.8 0          0  3.00  4.00
 8  14.7  8.00   440   230  3.23  5.34  17.4 0          0  3.00  4.00
 9  15.5  8.00   318   150  2.76  3.52  16.9 0          0  3.00  2.00
10  15.2  8.00   304   150  3.15  3.44  17.3 0          0  3.00  2.00
# ... with 22 more rows
```

```
arrange( mtcars, desc(cyl) )
```

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>
 1  18.7  8.00   360   175  3.15  3.44  17.0 0          0  3.00  2.00
 2  14.3  8.00   360   245  3.21  3.57  15.8 0          0  3.00  4.00
 3  16.4  8.00   276   180  3.07  4.07  17.4 0          0  3.00  3.00
```

```
 4  17.3  8.00   276   180  3.07  3.73  17.6 0          0  3.00  3.00
 5  15.2  8.00   276   180  3.07  3.78  18.0 0          0  3.00  3.00
 6  10.4  8.00   472   205  2.93  5.25  18.0 0          0  3.00  4.00
 7  10.4  8.00   460   215  3.00  5.42  17.8 0          0  3.00  4.00
 8  14.7  8.00   440   230  3.23  5.34  17.4 0          0  3.00  4.00
 9  15.5  8.00   318   150  2.76  3.52  16.9 0          0  3.00  2.00
10  15.2  8.00   304   150  3.15  3.44  17.3 0          0  3.00  2.00
# ... with 22 more rows
```

To sort variables only within groups, we sort by the grouping variable first and then the other sorting variables. The `arrange` function ignores `group_by`, which is important to remember because it's different than the other **dplyr** verbs we've learned today.

Here's an example of within-group sorting, sorting each cylinder category from lowest to highest `wt`.

```
arrange(mtcars, cyl, wt)
```

```
    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1  30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
2  30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
3  33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
4  27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
5  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
6  32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
7  22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
8  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
9  21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
10 22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
11 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
12 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
13 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
14 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
15 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
16 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
17 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
18 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
19 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
20 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
21 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
22 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
23 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
24 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
25 17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
26 15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
27 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
28 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
29 16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
30 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
31 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
32 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
```

To sort by more variables, just keep adding them in `arrange`.

Combining data manipulation tasks

When working with our own datasets we'll often want to do multiple data manipulation tasks in a row. Now that we've learned how to do different kinds of data manipulation, let's string multiple manipulations together.

We are going to:

1. Filter the `mtcars` dataset to just those cars with automatic transmissions;
2. Create a new variable that is the ratio of engine displacement and horsepower;
3. Calculate the mean of this new variable separately for each cylinder category.

Using temporary objects

First we'll do this one step at a time, creating a new named object for each step. As a reminder, we haven't been naming objects as we practiced the functions but instead were only printing results to the R Console. Now we're actually naming each object. The extra pair of parentheses prints the object to the Console so we can see what happens at each step.

```
# Filter by automatic transmission
( filtcars = filter(mtcars, am == 0) )
```

```
# Create new variable in the filtered dataset
( ratio.cars = mutate( filtcars, hd.ratio = hp/disp) )
```

```
# A tibble: 19 x 12
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb hd.ratio
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>    <dbl>
 1  21.4  6.00   258 110    3.08  3.22  19.4 1          0  3.00  1.00    0.426
 2  18.7  8.00   360 175    3.15  3.44  17.0 0          0  3.00  2.00    0.486
 3  18.1  6.00   225 105    2.76  3.46  20.2 1          0  3.00  1.00    0.467
 4  14.3  8.00   360 245    3.21  3.57  15.8 0          0  3.00  4.00    0.681
 5  24.4  4.00   147  62.0  3.69  3.19  20.0 1          0  4.00  2.00    0.423
 6  22.8  4.00   141  95.0  3.92  3.15  22.9 1          0  4.00  2.00    0.675
 7  19.2  6.00   168 123    3.92  3.44  18.3 1          0  4.00  4.00    0.734
 8  17.8  6.00   168 123    3.92  3.44  18.9 1          0  4.00  4.00    0.734
 9  16.4  8.00   276 180    3.07  4.07  17.4 0          0  3.00  3.00    0.653
10  17.3  8.00   276 180    3.07  3.73  17.6 0          0  3.00  3.00    0.653
11  15.2  8.00   276 180    3.07  3.78  18.0 0          0  3.00  3.00    0.653
12  10.4  8.00   472 205    2.93  5.25  18.0 0          0  3.00  4.00    0.434
13  10.4  8.00   460 215    3.00  5.42  17.8 0          0  3.00  4.00    0.467
14  14.7  8.00   440 230    3.23  5.34  17.4 0          0  3.00  4.00    0.523
15  21.5  4.00   120  97.0  3.70  2.46  20.0 1          0  3.00  1.00    0.808
16  15.5  8.00   318 150    2.76  3.52  16.9 0          0  3.00  2.00    0.472
17  15.2  8.00   304 150    3.15  3.44  17.3 0          0  3.00  2.00    0.493
18  13.3  8.00   350 245    3.73  3.84  15.4 0          0  3.00  4.00    0.700
19  19.2  8.00   400 175    3.08  3.84  17.0 0          0  3.00  2.00    0.438
```

```
# Group by number of cylinders
grp.ratio = group_by(ratio.cars, cyl)

# Calculate mean of the new ratio variable by cylinder category
( sum.ratio = summarise( grp.ratio, mratio = mean(hd.ratio) ) )
```

```
# A tibble: 3 x 2
    cyl mratio
```

```
  <dbl>  <dbl>
1  4.00  0.635
2  6.00  0.590
3  8.00  0.554
```

The downside of this method is that we made four objects when we really just wanted the final `sum.ratio` object. We have to think of names for each object at each step and we end up with a bunch of temporary objects in our R Environment.

Nesting functions to avoid temporary objects

An alternative to temporary objects is to *nest* all the functions together. This means we put one function call within the next function call. Nesting allows us to avoid making any temporary objects but the resulting code is a bit hard to read. The code from nested functions is read inside out, where the first thing we do is also the most nested.

First, a simple example of nesting functions from work we did earlier, where we want to group the dataset by `cyl` and `am` and then calculate the mean of `disp`. Here's the same task via nesting. We put the `group_by` function call within `summarise`.

```
summarise( group_by(mtcars, cyl, am), mdisp = mean(disp) )
```

```
# A tibble: 6 x 3
# Groups: cyl [?]
    cyl    am mdisp
  <dbl> <dbl> <dbl>
1  4.00  0    136
2  4.00  1.00  93.6
3  6.00  0    205
4  6.00  1.00 155
5  8.00  0    358
6  8.00  1.00 326
```

Now the more complicated example, where we combined the series of data manipulation tasks. Note how the `filter` is four functions deep in the code below.

```
( sum.ratio = summarise( group_by( mutate( filter(mtcars, am == 0),
                                   hd.ratio = hp/disp),
                           cyl),
                  mratio = mean(hd.ratio) ) )
```

```
# A tibble: 3 x 2
    cyl mratio
  <dbl>  <dbl>
1  4.00  0.635
2  6.00  0.590
3  8.00  0.554
```

The pipe operator

Now that we are combining multiple data manipulation functions from **dplyr**, it's time to talk about the pipe operator. The pipe operator (%>%) represents a different coding style that is fairly new in R. The pipe allows us to perform a series of data manipulation steps in a long *chain* while avoiding all those temporary objects or difficult-to-read nested code.

In essence, the pipe operator *pipes* a dataset into a function as the first argument. One reason I've been pointing out to you that the **dplyr** functions have the dataset as the first argument is that this is one of the things that makes piping so easy with these functions.

You can think of the pipe as being pronounced "then", which we'll talk more about as we see some examples. Using the pipe is a bit hard to picture when you are first introduced to it, but things will get clearer once we see some code.

Let's start with a simple example. Remember when we grouped `mtcars` by `cyl` earlier?

```
bycyl = group_by(mtcars, cyl)
```

We read even that simple code inside out. We see that we are grouping with `group_by` and then if we read inside the function we see the dataset we are going to group. Let's write this same code using the pipe.

```
bycyl = mtcars %>% group_by(cyl)
```

The code with the pipe is read from left to right. We see we are working with the `mtcars` dataset and *then* that we are grouping that dataset by `cyl`. The result is the same, but the code itself looks quite different.

Handily, we can keep piping through multiple functions in one long chain. Let's group `mtcars` by `cyl` and then calculate the mean `disp` of each group.

When working with pipes in a chain, it is standard to use a line break after each pipe with an indent for each subsequent function.

Aside: Stylistically, including white space in your code improves code readability. Think of writing a sentence without white space; it would be hard to read! Newer R users sometimes need to be reminded that white space rationing is not in effect. :-D It might seem clunky at first, but including white space quickly becomes natural and your code becomes much easier to read and understand.

```
mtcars %>%
    group_by(cyl) %>%
    summarise( mdisp = mean(disp) )
```

```
# A tibble: 3 x 2
    cyl mdisp
  <dbl> <dbl>
1  4.00   105
2  6.00   183
3  8.00   353
```

Again, the above code is read from left to right. We see we are going to work with `mtcars`, then we group it by `cyl`, and then we calculate the mean `disp` of the grouped dataset. When you read it like this you can see why we might pronounce `%>%` as *then*.

Combining data manipulation tasks using the pipe operator

Let's go back to our combined data manipulation task we did a few minutes ago on `mtcars` and use piping instead of temporary objects or nesting.

```
mtcars %>%
    filter(am == 0) %>% # filter out the manual transmission cars
    mutate(hd.ratio = hp/disp) %>% # make new ratio variable
    group_by(cyl) %>% # group by number of cylinders
    summarise(mratio = mean(hd.ratio) ) # calculate mean hd.ratio per cylinder category
```

```
# A tibble: 3 x 2
    cyl mratio
  <dbl>  <dbl>
```

```
1  4.00  0.635
2  6.00  0.590
3  8.00  0.554
```

We didn't assign a name to the final object. Let's do that.

```
sum.ratio = mtcars %>%
    filter(am == 0) %>% # filter out the manual transmission cars
    mutate(hd.ratio = hp/disp) %>% # make new ratio variable
    group_by(cyl) %>% # group by number of cylinders
    summarise(mratio = mean(hd.ratio) ) # calculate mean hd.ratio per cylinder category
```

Using the pipe operator with non-dplyr functions

The pipe operator can be used with functions outside the **dplyr** package, as well. If the first argument of the function is the dataset, the code looks just like what we've been doing. For example, we can use the pipe with the `head` function from base R and get the first 10 rows of `mtcars`. The first argument of the `head` function is the dataset.

```
mtcars %>% head(n = 10)
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

If the first argument of a function is *not* the dataset, we have to use the dot, `.`, to represent the dataset name in the function we are piping into. We can see this if we use the pipe operator with the `t.test` function, which doesn't have `data` as the first argument.

Here we test for a difference in mean horsepower among transmission types based on the `mtcars` dataset.

```
mtcars %>% t.test(hp ~ am, data = .)
```

```

    Welch Two Sample t-test

data:  hp by am
t = 1.2662, df = 18.715, p-value = 0.221
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -21.87858  88.71259
sample estimates:
mean in group 0 mean in group 1
       160.2632        126.8462
```

We generally wouldn't use piping in that simple case, though, as we would just use the data argument in `t.test` directly. A more realistic example is if we wanted to filter the dataset before doing the test. Let's filter `mtcars` to cars weighing less than or equal to 4000 lbs and then test if mean horsepower is different between transmission types.

```
mtcars %>%
    filter(wt <= 4) %>%
    t.test(hp ~ am, data = .)
```

```
    Welch Two Sample t-test

data:  hp by am
t = 0.76927, df = 19.747, p-value = 0.4508
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -35.68283  77.32385
sample estimates:
mean in group 0 mean in group 1
       147.6667        126.8462
```

A few more dplyr functions

We'll end Part 1 of the workshop by quickly running through a few additional **dplyr** functions.

The `n` function

The **dplyr** package has a built-in function, `n`, for counting up the unique rows in a group. This is useful when making tables of summary statistics.

```
mtcars %>%
    group_by(cyl) %>%
    summarise( n = n() )
```

```
# A tibble: 3 x 2
    cyl     n
  <dbl> <int>
1  4.00    11
2  6.00     7
3  8.00    14
```

This function can be used directly inside other functions, such as `filter`, for removing rows based on the group total count.

```
mtcars %>%
    group_by(cyl) %>%
    filter(n() < 10)
```

```
# A tibble: 7 x 11
# Groups: cyl [1]
    mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>
1  21.0  6.00   160   110  3.90  2.62  16.5 0       1.00  4.00  4.00
2  21.0  6.00   160   110  3.90  2.88  17.0 0       1.00  4.00  4.00
3  21.4  6.00   258   110  3.08  3.22  19.4 1       0     3.00  1.00
4  18.1  6.00   225   105  2.76  3.46  20.2 1       0     3.00  1.00
5  19.2  6.00   168   123  3.92  3.44  18.3 1       0     4.00  4.00
6  17.8  6.00   168   123  3.92  3.44  18.9 1       0     4.00  4.00
7  19.7  6.00   145   175  3.62  2.77  15.5 0       1.00  5.00  6.00
```

It can also be used when assigning index numbers within groups when rows are not uniquely identified. It is especially useful if the group sizes aren't known or might vary. In this example we'll also `select` just the first three columns so we can easily see the new `index` column that we create.

```
mtcars %>%
    group_by(cyl) %>%
    select(1:3) %>%
    mutate( index = 1:n() )
```

```
# A tibble: 32 x 4
# Groups: cyl [3]
     mpg   cyl  disp index
   <dbl> <dbl> <dbl> <int>
 1  21.0  6.00   160     1
 2  21.0  6.00   160     2
 3  22.8  4.00   108     1
 4  21.4  6.00   258     3
 5  18.7  8.00   360     1
 6  18.1  6.00   225     4
 7  14.3  8.00   360     2
 8  24.4  4.00   147     2
 9  22.8  4.00   141     3
10  19.2  6.00   168     5
# ... with 22 more rows
```

Sometimes we might want to add this index in based on the order of some variable in the dataset, not on the order the dataset is when we read it in. This is a case for `arrange`. Let's add the index based on the order of `disp` within each `cyl` category.

```
mtcars %>%
    arrange(cyl, disp) %>%
    group_by(cyl) %>%
    select(1:3) %>%
    mutate( index = 1:n() )
```

```
# A tibble: 32 x 4
# Groups: cyl [3]
     mpg   cyl  disp index
   <dbl> <dbl> <dbl> <int>
 1  33.9  4.00  71.1     1
 2  30.4  4.00  75.7     2
 3  32.4  4.00  78.7     3
 4  27.3  4.00  79.0     4
 5  30.4  4.00  95.1     5
 6  22.8  4.00 108       6
 7  21.5  4.00 120       7
 8  26.0  4.00 120       8
 9  21.4  4.00 121       9
10  22.8  4.00 141      10
# ... with 22 more rows
```

The `n_distinct` function

Another useful function is `n_distinct`, which we can use for counting up the number of unique values of a variable. I use this most when I'm learning about a dataset that I don't know well and am checking things

out.

I also use `n_distinct` when I think I have mistakes in a variable, such as a value of a categorical variable being misentered. For example, if we know our dataset should only have 3 values for number of cylinders we could check to make sure our variable doesn't contain more than that.

```
mtcars %>%
    summarise( ncyl = n_distinct(cyl) )
```

```
  ncyl
1    3
```

Another example is checking how many unique values of one variable is in each group. Here we'll see how many unique values of `mpg` there are in each group.

```
mtcars %>%
    group_by(cyl) %>%
    summarise( n = n(), nmpg = n_distinct(mpg) )
```

```
# A tibble: 3 x 3
    cyl     n  nmpg
  <dbl> <int> <int>
1  4.00    11     9
2  6.00     7     6
3  8.00    14    12
```

There are fewer unique `mpg` values (only 27) than there are rows in the dataset.

The `distinct` function

The last of the basic **dplyr** functions we will see is the `distinct` function. This is the function we can use if we need to remove duplicate-value rows from our dataset.

For example, we saw with `n_distinct` that we had less unique values of `mpg` in each `cyl` group than we had rows in the dataset. Let's pull out only the `distinct` values of `mpg` per `cyl` group. While this example is made up, there are times we could have duplicated rows that need to be removed before analysis.

The resulting dataset has 27 rows that have unique values of `mpg` within each cylinder category instead of 32 rows like the original dataset.

```
mtcars %>%
    group_by(cyl) %>%
    distinct(mpg)
```

```
# A tibble: 27 x 2
# Groups: cyl [3]
     mpg   cyl
   <dbl> <dbl>
 1  21.0  6.00
 2  22.8  4.00
 3  21.4  6.00
 4  18.7  8.00
 5  18.1  6.00
 6  14.3  8.00
 7  24.4  4.00
 8  19.2  6.00
 9  17.8  6.00
10  16.4  8.00
```

```
# ... with 17 more rows
```

Above we only kept the grouping variables and the variable we used to determine uniqueness. If we want to keep all the variables with `distinct`, we use the `.keep_all` argument.

```
mtcars %>%
    group_by(cyl) %>%
    distinct(mpg, .keep_all = TRUE)
```

```
# A tibble: 27 x 11
# Groups: cyl [3]
     mpg   cyl  disp    hp  drat    wt  qsec vs        am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <fctr> <dbl> <dbl> <dbl>
 1  21.0  6.00   160 110    3.90  2.62  16.5 0       1.00  4.00  4.00
 2  22.8  4.00   108  93.0  3.85  2.32  18.6 1       1.00  4.00  1.00
 3  21.4  6.00   258 110    3.08  3.22  19.4 1       0     3.00  1.00
 4  18.7  8.00   360 175    3.15  3.44  17.0 0       0     3.00  2.00
 5  18.1  6.00   225 105    2.76  3.46  20.2 1       0     3.00  1.00
 6  14.3  8.00   360 245    3.21  3.57  15.8 0       0     3.00  4.00
 7  24.4  4.00   147  62.0  3.69  3.19  20.0 1       0     4.00  2.00
 8  19.2  6.00   168 123    3.92  3.44  18.3 1       0     4.00  4.00
 9  17.8  6.00   168 123    3.92  3.44  18.9 1       0     4.00  4.00
10  16.4  8.00   276 180    3.07  4.07  17.4 0       0     3.00  3.00
# ... with 17 more rows
```

Part 2: Reshaping datasets

We are going to switch gears now and talk about how to *reshape* datasets.

In this section, we will learn to take the information from the columns of a dataset and put that information on rows instead. This is an example of taking a *wide* dataset and making it *long*. We will also learn to take information from the rows of a dataset and put that information into columns instead. In other words, reshape a dataset from *long* to *wide*. None of this changes how much information we have, it just changes how the information is stored.

We will be learning to reshape using the **tidyr** package. This is a relatively new package. It's predecessor is package **reshape2**, which does the same tasks but is no longer being actively developed.

The language of the **tidyr** package involves *gathering* and *spreading*. To *gather* a dataset means to take a wide dataset and transform it into a long dataset. To *spread* a dataset means to take a long dataset and make it wide. We'll see examples of these as we go along, which should help clear up any confusion about this new terminology.

We'll learn the basics of reshaping on what I call a *toy* dataset. A toy dataset is a set of fake data that we make to practice functions on. Small toy datasets are handy when you are learning a new function or trying to troubleshoot a data manipulation technique. We could use built-in datasets like `mtcars`, as well, but toy datasets are conveniently very small.

The dataset `toy1` that we will create will have six rows and five columns. One column contains the levels of some treatment (`trt`), one contains the individuals the treatment was applied to (`indiv`), and the last three are some quantitative measurement taken at three different times (`time1`, `time2`, and `time3`). Notice that the combination of treatment and individual is the unique identifier for each row. The shape of this toy dataset is one I commonly see for data from studies that take measurements through time.

```
# Make 2 treatments, a and b
( trt = rep( c("a", "b"), each = 3) )
```

```
[1] "a" "a" "a" "b" "b" "b"
```

```
# Make 6 individuals, named 1 through 3 within each treatment
( indiv = rep(1:3, times = 2) )
```

```
[1] 1 2 3 1 2 3
```

```
# Make values for the quantitative measurement
    # taken at 3 different time periods for the 6 individuals
    # by drawing from a normal(0,1) distribution 6 times
# I didn't set the seed (see "set.seed"),
    # so our datasets will all be slightly different
time1 = rnorm(n = 6)
time2 = rnorm(n = 6)
time3 = rnorm(n = 6)

# Put these five vectors into a data.frame
( toy1 = data.frame(indiv, trt, time1, time2, time3) )
```

```
  indiv trt      time1      time2       time3
1     1   a -1.9353909 -0.6730491  0.47536551
2     2   a  1.3593814  1.2380950 -0.66051331
3     3   a -1.2629170 -1.2364464  0.75585872
4     1   b -0.6843409 -0.1661094  0.68641699
5     2   b -2.2434534 -1.5773098 -0.14257312
6     3   b -1.2527772 -1.6543452 -0.07109041
```

This dataset `toy1` is in a *wide* format. If we were going to analyze this dataset in R, we would likely need it to be in a long format. We want to have a single column containing the information about the time of measurement (time1, time2, or time3) and a single column containing the values of the quantitative measurement in addition to the columns containing the identifying information, `indiv` and `trt`. To go from wide to long, we'll use the `gather` function.

Reshaping from wide to long with `gather`

In `gather`, the first thing we do after defining the dataset we want to reshape is to name the two new columns we are making. The first name, `key`, is the name of the new categorical variable that we will create that is based on the variable names of the columns we are gathering. The second, `value`, is the name of the new column that will contain all the values that used to be in multiple columns.

Once we name our new columns, all we have to do is list which columns we want to collapse into one. In this case, we want to gather up the three `time` variables.

Notice that we have the same amount of information in the long dataset as we did in the wide dataset. We changed the shape, not the data.

```
gather(toy1, key = time,
       value = measurement,
       time1, time2, time3)
```

```
   indiv trt  time measurement
1      1   a time1 -1.93539092
2      2   a time1  1.35938138
3      3   a time1 -1.26291696
4      1   b time1 -0.68434091
5      2   b time1 -2.24345344
6      3   b time1 -1.25277716
```

```
7      1   a time2 -0.67304909
8      2   a time2  1.23809500
9      3   a time2 -1.23644637
10     1   b time2 -0.16610939
11     2   b time2 -1.57730976
12     3   b time2 -1.65434522
13     1   a time3  0.47536551
14     2   a time3 -0.66051331
15     3   a time3  0.75585872
16     1   b time3  0.68641699
17     2   b time3 -0.14257312
18     3   b time3 -0.07109041
```

The **tidyr** package was built to be used with **dplyr**. The dataset is always the first argument so the functions can easily be used with pipes. In addition, we can use the **dplyr** `select_helpers` when choosing which columns we want to gather into one. This can be pretty convenient.

Here are three more examples of how we could ask for the three time columns to be gathered into one. All give the exact same output as above, as each gathers the data in `time1`, `time2`, and `time3` into a single column.

```
gather(toy1, key = time,
       value = measurement,
       time1:time3)

gather(toy1, key = time,
       value = measurement,
       -indiv, -trt)

gather(toy1, key = time,
       value = measurement,
       contains("time") )
```

We'd better name this newly long-format object so we can use it in further examples. We'll be using this long dataset to practice putting it back into wide format.

```
toy1long = gather(toy1, key = time,
                  value = measurement,
                  contains("time") )
```

Reshaping from long to wide with `spread`

Now we can use the function `spread` to reshape the long dataset `toy1long` back into the original wide shape. You might want to do this if, for example, you were going to take a dataset from an analysis done in R to graph in a program like SigmaPlot, which apparently often works best on wide datasets.

In the `spread` function, the first argument after defining the dataset is to define the `key` column. This is the column that contains the groups that we will want to be in unique columns. The values in the `key` column will be used as the new variable names in the wide format dataset. The second argument is the `value` argument, which is where we define the column that contains the values we want to fill the new columns with.

```
spread(toy1long, key = time, value = measurement)
```

```
  indiv trt      time1      time2       time3
1     1   a -1.9353909 -0.6730491  0.47536551
2     1   b -0.6843409 -0.1661094  0.68641699
```

```
3     2   a  1.3593814  1.2380950 -0.66051331
4     2   b -2.2434534 -1.5773098 -0.14257312
5     3   a -1.2629170 -1.2364464  0.75585872
6     3   b -1.2527772 -1.6543452 -0.07109041
```

Duplicate identifiers in `spread`

If the rows of the long dataset aren't uniquely identified you will not be able to spread the dataset wide.

For example, if we were trying to spread `toy1long` but we only had the `trt` variable and not the `indiv` variable our rows wouldn't be uniquely identified. It is only the combination of `trt` and `indiv` that uniquely identifies a row.

Let's remove `indiv` from the dataset using `select`.

```
toy1long %>%
       select(-indiv)
```

```
   trt  time measurement
1    a time1 -1.93539092
2    a time1  1.35938138
3    a time1 -1.26291696
4    b time1 -0.68434091
5    b time1 -2.24345344
6    b time1 -1.25277716
7    a time2 -0.67304909
8    a time2  1.23809500
9    a time2 -1.23644637
10   b time2 -0.16610939
11   b time2 -1.57730976
12   b time2 -1.65434522
13   a time3  0.47536551
14   a time3 -0.66051331
15   a time3  0.75585872
16   b time3  0.68641699
17   b time3 -0.14257312
18   b time3 -0.07109041
```

There are now multiple observations of each time for each `trt` category; we don't have unique identifiers. Let's see what happens when we try to `spread` this dataset.

```
toy1long %>%
       select(-indiv) %>%
       spread(key = time, value = measurement)
```

```
Error: Duplicate identifiers for rows (1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12), (13, 14, 15), (16,
```

We get a common `spread` error. When you get this error message it's usually time to take a step back and think about what you are trying to do and why. It is likely that there is an additional step that needs to be done prior to reshaping the dataset.

In this case, if we really want the dataset without `indiv` to be spread out wide we'll need to summarize it in some way first so that each `trt` category has a single value for each `time`. For example, we could calculate the mean for each time period. This is done using `group_by` and `summarise`.

```
toy1long %>%
    select(-indiv) %>%
```

```
    group_by(trt, time) %>%
    summarise(measurement = mean(measurement) )
```

```
# A tibble: 6 x 3
# Groups: trt [?]
  trt    time  measurement
  <fctr> <chr>       <dbl>
1 a      time1      -0.613
2 a      time2      -0.224
3 a      time3       0.190
4 b      time1      -1.39
5 b      time2      -1.13
6 b      time3       0.158
```

Then it's easy to spread because now the rows in the wide dataset can be uniquely identified by `trt`. We can `spread` within the same chain of pipes we used to summarize the dataset.

```
toy1long %>%
    select(-indiv) %>%
    group_by(trt, time) %>%
    summarise(measurement = mean(measurement) ) %>%
    spread(key = time, value = measurement)
```

```
# A tibble: 2 x 4
# Groups: trt [2]
  trt     time1  time2 time3
  <fctr>  <dbl>  <dbl> <dbl>
1 a      -0.613 -0.224 0.190
2 b      -1.39  -1.13  0.158
```

Using `unite` prior to spreading

There are other useful functions in the **tidyr** package, although we don't have time to cover them today. We will end our **tidyr** adventure by learning how to use `unite`.

We use `unite` when we want to spread a dataset and need a combination of two variables as our new columns. It's pretty easy to use; we just need to name the new column we are making and then list the columns that contain values we want to unite together.

Using the defaults, the values in the new column will be separated by an underscore (`_`) and the columns that we united are removed from the dataset. See the help page for more options.

Here we'll unite the values of `trt` and `time` into a new variable we'll name `trt_time`.

```
toy1long %>%
    unite(col = trt_time, trt, time)
```

```
   indiv trt_time measurement
1      1  a_time1 -1.93539092
2      2  a_time1  1.35938138
3      3  a_time1 -1.26291696
4      1  b_time1 -0.68434091
5      2  b_time1 -2.24345344
6      3  b_time1 -1.25277716
7      1  a_time2 -0.67304909
8      2  a_time2  1.23809500
9      3  a_time2 -1.23644637
```

```
10     1  b_time2 -0.16610939
11     2  b_time2 -1.57730976
12     3  b_time2 -1.65434522
13     1  a_time3  0.47536551
14     2  a_time3 -0.66051331
15     3  a_time3  0.75585872
16     1  b_time3  0.68641699
17     2  b_time3 -0.14257312
18     3  b_time3 -0.07109041
```

Once the dataset has the new combined variable, we can use that in `spread` to make a very wide format dataset. Again notice that we still have the same amount of information in this dataset, it's just in a different format.

```
toy1long %>%
    unite(col = trt_time, trt, time) %>%
    spread(key = trt_time, value = measurement)
```

```
  indiv   a_time1    a_time2    a_time3    b_time1    b_time2     b_time3
1     1 -1.935391 -0.6730491  0.4753655 -0.6843409 -0.1661094  0.68641699
2     2  1.359381  1.2380950 -0.6605133 -2.2434534 -1.5773098 -0.14257312
3     3 -1.262917 -1.2364464  0.7558587 -1.2527772 -1.6543452 -0.07109041
```

The complement to `unite` is `separate`, which we won't see today but is often used when we want to gather multiple sets of columns.

Part 3: Joining two datasets together

The last topic we need to cover before you can practice these functions on a some data manipulation problems is merging or *joining*. For a variety of reasons, we might have data for a single analysis stored in separate datasets. Joining is the process of combining two datasets based on matching values in the columns you are using as the *unique identifiers*. The unique identifier variables are the variables in the dataset that tells the computer which rows in one dataset should be matched to rows in another dataset.

There is a `merge` function in base R, but we will be using some of the join functions from **dplyr** today, including `inner_join`, `left_join`, and `full_join`.

Let's create two toy datasets to join together. One dataset (`tojoin1`) will contain counts of some species in three different treatment plots (`treat`) within different sites (`site`). The other dataset (`tojoin2`) will contain an environmental variable, measured on the same plots and sites (`elev`). Both datasets are missing measurements from a treatment plot in site 3; the first dataset is missing treatment "c" and the second dataset is missing treatment "a".

This time I will make each dataset in a single step rather than creating each variable separately first. The key to making a `data.frame` like this is to make sure each variable is the same length as each other variable.

If we `set.seed` to the same number, we'll all get the same random numbers from `rpois` and `rgamma`.

```
set.seed(16) # If I set the seed, we will all get the same random numbers

# This dataset is slightly unbalanced, as site 3
    # doesn't have the "c" treatment count
( tojoin1 = data.frame(site = rep(1:3, each = 3, length.out = 8),
                treat = rep(c("a", "b", "c"), length.out = 8),
                count = rpois(8, 6) ) )
```

```
  site treat count
```

```
1    1     a     7
2    1     b     4
3    1     c     6
4    2     a     4
5    2     b     9
6    2     c     5
7    3     a     3
8    3     b     8
```

```
# This dataset is also slightly unbalanced,
    # missing the elevation measurement from
    # site 3 treatment "a"
( tojoin2 = data.frame(site = rep(1:3, length.out = 8),
                treat = rep(c("b", "c", "a"), each = 3, length.out = 8),
                elev = rgamma(8, 1000, 1) ) )
```

```
  site treat      elev
1    1     b 1036.1179
2    2     b  984.7461
3    3     b 1012.7026
4    1     c  977.3154
5    2     c 1017.7019
6    3     c 1058.7514
7    1     a 1003.0419
8    2     a  976.0533
```

The unique identifier of each measurement in each dataset is a combination of `site` and `treat`; those are the variables that we will use to tell the computer which rows within the two datasets to combine into one.

The inner join

Let's start our joining practice by joining these two datasets together using `inner_join`.

See the help page, `?join`, to see a description of each type of join available in **dplyr**. In the documentation, you will see that every join involves two datasets, called `x` and `y`, to be joined. The `x` dataset is the first dataset you give to the `join` function and the `y` dataset is the second.

An *inner join* matches on the unique identifiers and returns only rows that are shared in both datasets.

From the documentation, an `inner_join` will

> return all rows from x where there are matching values in y, and all columns from x and y

By default, `inner_join` joins on all columns shared by the two datasets. When we use this default, we will get a message telling us which variables were used for joining when we run the code.

We'll name our new combined dataset `joined`, and print the result to the R Console.

```
( joined = inner_join(tojoin1, tojoin2) )
```

```
Joining, by = c("site", "treat")
```

```
  site treat count      elev
1    1     a     7 1003.0419
2    1     b     4 1036.1179
3    1     c     6  977.3154
4    2     a     4  976.0533
5    2     b     9  984.7461
6    2     c     5 1017.7019
```

```
7    3     b     8 1012.7026
```

To make our code more explicit and easily understandable, we can also use the `by` argument to define which variables we want to join on.

```
inner_join( tojoin1, tojoin2, by = c("site", "treat") )
```

```
  site treat count      elev
1    1     a     7 1003.0419
2    1     b     4 1036.1179
3    1     c     6  977.3154
4    2     a     4  976.0533
5    2     b     9  984.7461
6    2     c     5 1017.7019
7    3     b     8 1012.7026
```

We see above that the joined dataset only has 7 rows. This is because there are only 7 site-treatment combinations that are present in both datasets. If we want to retain more rows, we'll need a different kind of join.

The left join

A left join is used when we want to keep all rows in the first dataset regardless of if they have a match in the second dataset.

From the documentation, `left_join` will

> return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns.

So in our scenario, we should get 8 rows back because we have 8 rows in the first dataset (`tojoin1`). We will still be missing a row for site 3 treatment "a", as this is not present in the first dataset.

```
left_join( tojoin1, tojoin2, by = c("site", "treat") )
```

```
  site treat count      elev
1    1     a     7 1003.0419
2    1     b     4 1036.1179
3    1     c     6  977.3154
4    2     a     4  976.0533
5    2     b     9  984.7461
6    2     c     5 1017.7019
7    3     a     3        NA
8    3     b     8 1012.7026
```

The full join

To keep all rows in both datasets regardless of a match, we can make a full join via `full_join`.

The `full_join` function will

> return all rows and all columns from both x and y. Where there are not matching values, returns NA for the one missing.

This is how we can get rows for all nine site-treatment combinations.

```
full_join( tojoin1, tojoin2, by = c("site", "treat") )
```

```
  site treat count      elev
1    1     a     7 1003.0419
2    1     b     4 1036.1179
3    1     c     6  977.3154
4    2     a     4  976.0533
5    2     b     9  984.7461
6    2     c     5 1017.7019
7    3     a     3        NA
8    3     b     8 1012.7026
9    3     c    NA 1058.7514
```

There is also a `right_join`, which we won't practice today but works a lot like the `left_join`.

Matching multiple rows when joining

There is an additional sentence in the documentation when describing the joins that we haven't discussed yet.

> If there are multiple matches between x and y, all combination of the matches are returned.

This is an important topic to cover, as sometimes we want this behavior but other times this will help us uncover a mistake we are making.

When this is useful

For example, if we wanted to join a variable that was only measured at the "site" level, this behavior is desirable. Let's make a dataset that has a variable measured at the site level.

```
# A site level variable, the amount of rainfall
( tojoin3 = data.frame(site = 1:3,
                       rainfall = rgamma(3, 10, 1) ) )
```

```
  site rainfall
1    1 16.43203
2    2  9.30625
3    3 11.35799
```

This new dataset only has 3 rows. Every treatment plot in the count dataset needs to be assigned to the same value of the `rainfall` site-level variable. So each row in the site-level dataset will be matched to multiple rows in the count dataset when we join, which is what we want. We end up with 8 rows, the same as what we had to start with.

```
left_join(tojoin1, tojoin3, by = "site")
```

```
  site treat count rainfall
1    1     a     7 16.43203
2    1     b     4 16.43203
3    1     c     6 16.43203
4    2     a     4  9.30625
5    2     b     9  9.30625
6    2     c     5  9.30625
7    3     a     3 11.35799
8    3     b     8 11.35799
```

When this indicates a mistake

This sort of behavior can cause unexpected results, though. If we join our original two joining datasets using only `site` instead of both variables that make up the unique identifier of each row, we will end up with multiple matches per row. This leads us with a dataset that is unexpectedly long. When this happens unexpectedly, we likely need to step back and evaluate if we have unique identifiers. We may need to rethink what we are doing versus what we want the final dataset to look like.

```
left_join(tojoin1, tojoin2, by = "site")
```

```
# A tibble: 22 x 5
    site treat.x count treat.y  elev
   <int> <fctr>  <int> <fctr>  <dbl>
 1     1 a           7 b        1036
 2     1 a           7 c         977
 3     1 a           7 a        1003
 4     1 b           4 b        1036
 5     1 b           4 c         977
 6     1 b           4 a        1003
 7     1 c           6 b        1036
 8     1 c           6 c         977
 9     1 c           6 a        1003
10     2 a           4 b         985
# ... with 12 more rows
```

Using `anti_join` to find missing data

The very last function we'll learn today is yet another kind of join, called the `anti_join`.

An `anti_join` will

> return all rows from x where there are not matching values in y, keeping just columns from x.

This is perfect for figuring out which rows are missing matches between two datasets. In an anti-join, we want to only return the values in the `x` dataset that are *not* in the `y` dataset.

Both `anti_join` and the related `semi_join` act more like filters than joins.

Here's how that looks, pulling out the row in `tojoin1` that is missing from `tojoin2`.

```
anti_join( tojoin1, tojoin2, by = c("site", "treat") )
```

```
  site treat count
1    3     a     3
```

If we wanted to find the row in `tojoin2` that is missing in `tojoin1`, we switch the order we put the datasets in `anti_join`.

```
anti_join( tojoin2, tojoin1, by = c("site", "treat") )
```

```
  site treat     elev
1    3     c 1058.751
```

Using the join functions with the pipe operator

The join functions can easily be used with the pipe operator. We can only pipe in one dataset at a time, so we have to decide if we want to pipe the dataset in as the `x` dataset or the `y` dataset.

Piping in a `join` function isn't super useful for these simple examples I'm showing you, but we can easily fit a join into a longer pipe chain.

If piping a dataset in as the `x` dataset, the piped-in dataset is just the first argument of whatever `join` function you are using. This example uses the `anti_join`.

```
tojoin1 %>%
    anti_join( tojoin2, by = c("site", "treat") )
```

```
  site treat count
1    3     a     3
```

We can pipe the dataset as the `y` dataset, as well, using the `.` placeholder we saw earlier.

```
tojoin1 %>%
    anti_join( tojoin2, ., by = c("site", "treat") )
```

```
  site treat     elev
1    3     c 1058.751
```

Part 4: Using what you've learned

In the last part of the workshop you have a chance to use some of the tools you've seen today. I've set up three example problems below. Each example will take a different set of functions to solve.

The `babynames` dataset

We'll be practicing using the `babynames` dataset. This can be found in package **babynames**, which we will need to install in this room so we can use it. We'll install this via the RStudio Packages pane, using the "Install" button, but we could also run the code `install.packages("babynames")`.

Once the package is installed, we can load the package.

```
library(babynames)
```

The help page for `babynames` gives us some basic information on the dataset.

```
?babynames
```

The `babynames` dataset contains data from the United States Social Security Administration on the number and proportion of babies given a name each year from 1880-2015. Rare names (recorded less than 5 times) are excluded from the dataset in R. The annual proportion of babies given a name was calculated separately for male and female babies (`sex`).

The dataset has five variables, shown below.

```
glimpse(babynames)
```

```
Observations: 1,858,689
Variables: 5
$ year <dbl> 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880, 1880...
$ sex  <chr> "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F", "F...
$ name <chr> "Mary", "Anna", "Emma", "Elizabeth", "Minnie", "Margaret", "Ida", "Alice", "Bertha...
$ n    <int> 7065, 2604, 2003, 1939, 1746, 1578, 1472, 1414, 1320, 1288, 1258, 1226, 1156, 1063...
$ prop <dbl> 0.072384329, 0.026679234, 0.020521700, 0.019865989, 0.017888611, 0.016167370, 0.01...
```

```
head(babynames)
```

```
# A tibble: 6 x 5
   year sex   name          n   prop
  <dbl> <chr> <chr>     <int>  <dbl>
1  1880 F     Mary       7065 0.0724
2  1880 F     Anna       2604 0.0267
3  1880 F     Emma       2003 0.0205
4  1880 F     Elizabeth  1939 0.0199
5  1880 F     Minnie     1746 0.0179
6  1880 F     Margaret   1578 0.0162
```

Practice problem 1

The first practice problem involves filtering and sorting.

> In the year you were born, which name was most popular for each sex?
>
> In that year, which were the top ten names overall based on the proportion of babies given a name?

Hint: For this second question you might want to use the `top_n` function. We did not review this function and it's not needed to answer this question, but you may find it useful.

Practice problem 2

The second practice problem involves filtering, adding a new variable, grouping a dataset, and summarizing.

Choose a name to work with. This might be your own name. However, I found that most of the names in this dataset are relatively common English or Spanish names. If you don't have a name that fits, you might want to choose one that is on the list. Ask me if you can't think of one.

> What is the rank of the name you chose, based on proportion, in 2015? Do this for one or both sexes (this may depend on the name you chose).
>
> What is the rank of the name you chose, based on proportion, for 2010-2015?

Hint: To filter to all variables in a vector instead of a single value we need `%in%` instead of `==`. So to filter to years from 1976 through 1978 we'd use `year %in% 1976:1978`.

Practice problem 3

The third practice problem involves filtering, selecting, and spreading.

> Find the proportion of babies given the name you chose for each year from 2010 to 2015.
>
> Convert the resulting dataset from a long format into a wide format, where years are now in columns.

Hint: Extra continuous variables will need to be removed prior to converting into a wide format to get the output as one row per sex.