Lesson 4: Data manipulation in the tidyverse

Patrick Mathias

A brief diversion to discuss the tidyverse

According to the official tidyverse website, "the tidyverse is an *opinionated* collection of R packages designed for data science." We've gotten a flavor of tidyverse functionality by using the readr packages and will wade deeper into the tidyverse in the next lessons. Because the tidyverse was not a component of the introductory MSACL data science course in previous years, we are going to cover basic functionality of many tidyverse packages throughout the rest of the course. Many of the data manipulation concepts will probably be familiar but the tidyverse offers a consistent interface for functions. Data is consistently the first argument for functions, and that enables compatibility with pipes. The tidyverse includes its own version of a data frame, the tibble, with the primary advantages being nicer printing of output and more predictable behavior with subsetting.

One of the key concepts of the tidyverse philosophy is maintaing "tidy" data. Tidy data is a data structure and a way of thinking about data that not only facilitates using tidyverse packages but more importantly it also provides a convention for organizing data that is amenable to data manipulation. The three criteria for tidy data are: 1. Each variable must have its own column. 2. Each observation must have its own row. 3. Each value must have its own cell.

As an example straight out of the R for Data Science text, consider a data set displaying 4 variables: country, year, population, and cases. One representation might split cases and population on different rows, even though each observation is a country and year:

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

Or case and population may be jammed together in one column:

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

The tidy representation is:

```
table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Each observation is on one row, and each column represents a variable, with no values being shoved together into a single column.

An advantage of using the tidyverse packages is the relatively robust support documentation around these packages. Stack Overflow is often a go to for troubleshooting but many tidyverse packages have nice vignettes and other online resources to help orient you to how the package functions work. There is a freely available online book, R for Data Science that covers the tidyverse (and more). Cheat Sheets provided by RStudio also provide great quick references for tidyverse and other packages.

You can load the core tidyverse packages by loading tidyverse: `library(tidyverse)`. ggplot2 is probably the most popular tidyverse package and arguably the go to for sophisticated visualizations in R, but inevitably data will need to be manipulated prior to plotting. So the two workhorse packages for many applications are dplyr and tidyr, which we will cover in this lesson.

Manipulating data with dplyr

The dplyr package provides functions to carve, expand, and collapse a data frame (or tibble).

Carving your data set

Reducing a data set to a subset of columns and/or rows are common operations, particularly on the path to answering a specific set of questions about a data set. If you need to go from a large number of columns (variables) to a smaller set, `select()` allows you to select specific columns by name. If you need only a subset of rows from your data set, `filter()` allows you to pick rows (cases) based on values, ie. you can subset your data based on logic.

Let's take these for a spin using the data we started examining in the last lesson.

Review the type of data we were working with:

```
samples_jan <- read_csv(
  here::here("class_data", "2017-01-06.csv"),
  # note that we added the package name to the front of the function
  # why? unfortunately lubridate has a deprecated here function, so we have to tell R which here to use
  col_types = cols(
    compoundName = col_factor(NULL),
    sampleType = col_factor(NULL)
    )
  ) %>%
  clean_names()
str(samples_jan)
```

Let's say we don't need the last two logical columns and want to get rid of them. We can use `select()` and provide a range of adjacent variables:

```
samples_jan %>%
  select(batch_name:expected_concentration)
```

Or we only care about the first 3 variables plus the concentration:

```
samples_jan %>%
  select(batch_name:compound_name, concentration)
```

Now let's carve the data set in the other direction. If we only care about the morphine data, we can use `filter()` to pick those rows based on a logical condition:

```
samples_jan %>%
  filter(compound_name == "morphine")
```

Or maybe we want to examine only the unknown samples with a concentration greater than 0:

```
samples_jan %>%
  filter(sample_type == "unknown", concentration > 0)
```

Note that a comma in the filter state implies a logical AND - condition A and condition B. You could include an OR condition as well using the pipe character | - condition A | condition B.

```
samples_jan %>%
  filter(sample_type == "unknown" | concentration > 0)
```

Exercise 1

Carve the January data set in both directions. We want sample information (batch, sample, compound) and ion ratio data for only oxycodone measurements in unknown sample types with a concentration > 0. Provide a summary of the data.

```
samples_jan_oxy_ir <- samples_jan %>%
  filter() %>%
  select()
summary()
```

Expanding your data set

Another common data manipulation task is adding or replacing columns that are derived from data in other columns. The `mutate()` function provides a quick and clean way to add additional variables that can include calculations, evaluating some logic, string manipulation, etc. You provide the function with the following argument(s): name of the new column = value. For example, if we continue with our January sample data set that includes concentrations and expected concentrations for standards, we can calculate the ratio of concentration to expected:

```
samples_jan %>%
  filter(sample_type == "standard", expected_concentration > 0) %>%
  mutate(conc_ratio = concentration/expected_concentration) %>%
  select(batch_name:compound_name, concentration, expected_concentration, conc_ratio)
```

Notice that we got around the issue of dividing by 0 by filtering for expected concentrations above 0. However, you may want to include these yet don't want R to throw an error. How can you deal with edge cases like this? `mutate()` borrows from SQL (Structured Query Language) and offers a `case_when` syntax for dealing with different cases. The syntax takes some getting used to but this can be helpful when you want to classify or reclassify values based on some criteria. Let's do the same calculation but spell out the case when expected_concentration is 0 and add a small number to numerator and denominator in that case:

```r
samples_jan %>%
  filter(sample_type == "standard") %>%
  mutate(
    conc_ratio = case_when(
      expected_concentration == 0 ~ (concentration + 0.001)/(expected_concentration + 0.001),
      TRUE ~ concentration/expected_concentration
    )
  ) %>%
  select(batch_name:compound_name, concentration, expected_concentration, conc_ratio)
```

Another common operation manipulation is wrangling dates. The lubridate package offers a helpful toolset to quickly parse dates and times. The bread and butter parsing functions are named intuitively based on the order of year, month, date, and time elements. For example, `mdy("1/20/2018")` will convert the string into a date that R can use. There are other useful functions like `month()` and `wday()` that pull out a single element of the date to use for grouping operations, for example. Let's work with a different January data set that has batch data and parse the collection dates in a variety of ways:

```r
batch_jan <- read_excel(here::here("class_data", "2017-01-06.xlsx")) %>%
  clean_names()
# note: here is also a function in lubridate so you must specify the package explicitly before calling
# syntax for specifying package is "package"::"function"
batch_jan_timestamps <- batch_jan %>%
  mutate(
    collect_datetime = ymd_hms(batch_collected_timestamp),
    collect_month = month(batch_collected_timestamp),
    collect_day_of_week = wday(batch_collected_timestamp),
    collect_week = week(batch_collected_timestamp),
    collect_week_alt = floor_date(collect_datetime, unit = "week")
    # floor_date to use datetime format but group to first day of week
  )
summary(batch_jan_timestamps)
```

You can see from the above example that these functions provide a great deal of flexibility in associating a row with arbitrary time scales. This allows the ability to group items by time and calculate summary data, which we will discuss in the next section.

> Exercise 2
>
> How long an average does it take to review each batch? Using the January batch data in "2017-01-06.xlsx", convert the review start timestamp and review complete timestamp fields into variables with a datetime type, then generate a new field the calculates the duration of the review in minutes. There are multiple approaches to this, but the `difftime()` function may be the most transparent - read the help on this function. The data will need to be collapsed by batch (which I do for you using the `distinct()` function) and display the min, max, median, and mean review times.

```r
batch_jan_reviews <- batch_jan %>%
  mutate(review_start_timestamp = ,
         review_complete_timestamp = ,
         review_duration = as.numeric(difftime( , , units = "minutes")))
# note: the output of difftime is a time interval, convert to numeric to avoid confusion downstream
reviews_jan_grouped <- batch_jan_reviews %>%
  distinct(batch_name, review_duration)
min()
```

```
median()
mean()
max()
```

Collapse (summarize) your data set

Carving and expanding your data are helpful but they are relatively simple. Often you will need to do more sophisticated analyses such as calculating statistical measures for multiple subsets of data. Grouping data by a variable using the `group_by()` function is critical tool provided by dplyr and naturally couples with its summary function `summarize()`. By grouping data you can apply a function within individual groups and calculate things like mean or standard deviation. As an example, we may want to look at our January sample data set and look at some statistics for the ion ratios by compound for the unknown sample type with non-zero concentation.

```
samples_jan %>%
  filter(sample_type == "unknown", concentration > 0) %>%
  group_by(compound_name) %>%
  summarize(median_ir = median(ion_ratio),
            mean_ir = mean(ion_ratio),
            std_dev_ir = sd(ion_ratio))
```

We may want to look at this on the batch level, which only requires adding another variable to the `group_by()` function.

```
samples_jan %>%
  filter(sample_type == "unknown", concentration > 0) %>%
  group_by(batch_name, compound_name) %>%
  summarize(median_ir = median(ion_ratio),
            mean_ir = mean(ion_ratio),
            std_dev_ir = sd(ion_ratio))
```

Let's revisit our batch dataset with timestamps that we have parsed by time period (eg. month or week) and look at correlation coefficient statistics by instrument, compound, and week:

```
batch_jan_timestamps %>%
  group_by(instrument_name, compound_name, collect_week) %>%
  summarize(median_cor = median(calibration_r_2),
            mean_cor = mean(calibration_r_2),
            min_cor = min(calibration_r_2),
            max_cor = max(calibration_r_2))
```

A relatively new package that provides nice grouping functionality based on times is called tibbletime. This package provides similar functionality to the mutating and summarizing we did with times above but has a cleaner syntax for some operations and more functionality.

> Exercise 3
>
> From the January sample dataset, for samples with unknown sample type, what is the minimum, median, mean, and maximum concentration for each compound by batch? What is the mean of the within-batch means by compound?

```
sample_stats_jan <- samples_jan %>%
  filter() %>%
  group_by() %>%
  summarize(min_conc = ,
            median_conc = ,
            mean_conc = ,
            max_conc = )
head()
sample_means_jan <- sample_stats_jan %>%
  group_by() %>%
  summarize(overall_mean = )
sample_means_jan
```

Shaping and tidying data with tidyr

Data in the real world are not always tidy. Consider a variant of the January sample data we've reviewed previously in the "2017-01-06-messy.csv" file.

```
samples_jan_messy <- read_csv(here::here("class_data", "messy", "2017-01-06-sample-messy.csv"))
head(samples_jan_messy)
```

This certainly isn't impossible to work with, but there are some challenges with not having separate observations on each row. Arguably the biggest challenges revolve around built-in tidyverse functionality, with grouping and plotting as the most prominent issues you might encounter. Luckily the tidyr package can help reshape your data. The `gather()` function can take a list of columns as arguments, the key argument to name the variable you are gathering, and the value argument to name the new column with the values you extract from the old column.

```
samples_jan_tidy <- samples_jan_messy %>%
  gather("morphine", "hydromorphone", "oxymorphone", "codeine", "hydrocodone", "oxycodone",
         key = "compound_name", value = "concentration")
head(samples_jan_tidy)
```

The syntax takes some getting used to, so it's important to remember that you are taking column names and shoving those into rows, so you have name that variable (the key), and you are also putting values across multiple columns into one column, whose variable also needs to be named (the value).

Sometimes other people want your data and they prefer non-tidy data. Sometimes you need messy data for quick visualization purposes. Or sometimes you have data that is actually non-tidy not because multiple observations are on one row, but because a single observation is split up between rows when it could be on one row. It is not too difficult to perform the opposite operation of `gather()` using the `spread()` function. You specify the key, which is the variable than needs to be used to generate multiple new columns, as well as the value, which takes the specifies the variable that will need to populate those new columns. Let's do the opposite on the data set we just gathered:

```
samples_jan_remessy <- samples_jan_tidy %>%
  spread(key = "compound_name", value = "concentration")
head(samples_jan_remessy)
```

There are other useful tidyr functions such as `separate()` and `unite()` to split one column into multiple columns or combine multiple columns into one column, respectively. These are pretty straightforward to pick up so can be an independent exercise if you are intereted.

> Exercise 4
>
> The "2017-01-06-batch-messy.csv" file in the messy subdirectory of class_data is related to the "2017-01-06.xlsx" batch file you have worked with before. Unfortunately, it is not set up to have a single observation per row. There are two problems that need to be solved: 1. Each parameter in a batch is represented with a distinct column per compound, but all compounds appear on the same row. Each compound represents a distinct observation, so these should appear on their own rows. 2. There are 3 parameters per obsevation (compound) - calibration slope, intercept,

and R^2. However these appear on different lines. All 3 paramters need to appear on the same row. After solving these problems, each row should contain a single compound with all three parameters appearing on that single row. Use `gather()` and `spread()` to tidy this data.

```r
batch_jan_messy <- read_csv(here::here("class_data", "messy", ))
batch_jan_gathered <- batch_jan_messy %>%
  gather()
batch_jan_tidy <- batch_jan_gathered %>%
  spread()
```

Summary

- The dplyr package offers a number of useful functions for manipulating data sets
  - `select()` subsets columns by name and `filter()` subset rows by condition
  - `mutate()` adds additional columns, typically with calculations or logic based on other columns
  - `group_by()` and `summarize()` allow grouping by one or more variables and performing calculations within the group
- Manipulating dates and times with the lubridate package can make grouping by time periods easier
- The tidyr package provides functions to tidy and untidy data