A brief intro to using R and the tidyverse for data manipulation

Harly Durbin

9/16/2020

Some background on how we’re doing what we’re doing

- Currently, we’re working inside of **RStudio**, which allows you to interface with the R software.
- We are using an **R Markdown document**. R Markdown documents have the file extension “.Rmd” and can only be opened in RStudio. R Markdowns work just like normal R scripts, except that they allow you mix commentary as plain text and code housed in “chunks”.
  - To create a new chunk, click the “Insert” tab in the upper right hand of this window then select “R”
  - R Markdown documents can be “rendered” to lots of file formats, including PDFs, Word Docs, and interactive HTML. This allows you create a report of your rendered code and figures alongside commentary. We’re not going to cover this today, but I’d recommend checking out “R Markdown: the Definitive Guide” by Yihui Xie which can be found free online here:
- All of our data, analysis, and results will be in a single file with multiple subfiles called a **Project**. More about that here:

Import “Day 1” data using the `{readxl}` package

Our data is stored in the Excel file “sample-data.xlsx”, located in the “data” folder of our project. It’s a shuffled and modified version of a friend’s data from a project studying the effects of supplementation with a certain hormone on conception rate in beef cattle. Hormone concentration was collected on the same day as supplementation (in the “Day 1” sheet) and then again two days later (in the “Day 3” sheet). This project was conducted in 3 replicates over 3 years.

We’re going to load the `{readxl}` package to read our spreadsheet data into R. We’re also going to load the `here` package. When you use the the `here()` command from the `{here}` package to designate a file path, R knows to look for the file relative to root of your project directory. I’ve pre-installed all of the packages we’re using today, but normally you’d have to use `install.packages()` to download them to your computer first before using `library()` to load them.

```r
library(readxl)
library(here)
```

```
## here() starts at /cloud/project
```

Now that we’ve loaded our packages, let’s try taking a look at the data stored in the “Day 1” sheet. It should have 5 columns and 547 rows.

```r
read_excel(here("data/sample-data.xlsx"),
           sheet = "Day 1")
```

```
## New names:
## * `` -> ...2
## * `` -> ...3
## * `` -> ...4
## * `` -> ...5
```

```
## # A tibble: 548 x 5
##    `*Any concentration that came back as ~ ...2    ...3      ...4      ...5
##    <chr>                                   <chr>   <chr>     <chr>     <chr>
##  1  <NA>                                   <NA>    <NA>      <NA>      <NA>
##  2 "*Any score of \"0\" has been changed ~ <NA>    <NA>      <NA>      <NA>
##  3 "Cow ID"                                Hormon~ Rep 1     Rep 2     Rep 3
##  4 "08693"                                 Low     .         .         2.336207~
##  5 "13102"                                 Low     2.396259~ .         2.745059~
##  6 "07745"                                 Low     .         0.893256~ .
##  7 "13144"                                 Low     .         2.321018~ .
##  8 "13066"                                 Low     .         0.852166~ 2.207001~
##  9 "12831"                                 Low     .         0.991173~ .
## 10 "14263"                                 Low     .         0.588676~ .
## # ... with 538 more rows
```

It looks like the data wasn't imported correctly because the spreadsheet contains 3 rows of notes at the top. Try again skipping those rows by adding the `skip` argument in `read_excel()`. It also looks like missing values were coded with a dot. In order to avoid implied `NA`s or confusion about whether or not the dots represent a value, we're also going to add the `na` argument to tell `read_excel` to interpret dots as `NA`s.

```r
read_excel(here("data/sample-data.xlsx"),
           sheet = "Day 1",
           skip = 3,
           na = ".")
```

```
## # A tibble: 545 x 5
##    `Cow ID` HormoneTrt `Rep 1`          `Rep 2`           `Rep 3`
##    <chr>    <chr>      <chr>            <chr>             <chr>
##  1 08693    Low        <NA>             <NA>              2.33620793032117
##  2 13102    Low        2.39625917019355 <NA>              2.74505984432731
##  3 07745    Low        <NA>             0.893256859419714 <NA>
##  4 13144    Low        <NA>             2.32101837926091  <NA>
##  5 13066    Low        <NA>             0.852166834158408 2.20070171568918
##  6 12831    Low        <NA>             0.991173203186964 <NA>
##  7 14263    Low        <NA>             0.588676304573575 <NA>
##  8 12923    Low        <NA>             3.14628738079207  <NA>
##  9 08796    Low        <NA>             2.68025359704464  <NA>
## 10 14298    Low        <NA>             1.14464827771139  <NA>
## # ... with 535 more rows
```

We've imported the data, but it still violates a rule of tidy data storage in spreadsheets. Head back to the main room to discuss.

"Rectangulating" the Day 1 data using `pivot_longer()` from the `{tidyr}` package

Now, we'll use the `pivot_longer()` command from the `{tidyr}` package to reshape the Day 1 data. We'll use the pipe, loaded from the `{dplyr}` package, to chain this to our previous `read_excel()` command. R doesn't play well with column names, so after we pivot the data we're going to use the `clean_names()`

function from the {janitor} package in order to automatically generate R-friendly column names. We could have also chosen manually rename these columns using rename() or select() from the {dplyr} package.

Load the packages:

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

The resulting data frame should have 4 columns and 1,635 rows.

```
read_excel(here("data/sample-data.xlsx"),
           sheet = "Day 1",
           skip = 3,
           na = ".") %>%
  pivot_longer(cols = c("Rep 1", "Rep 2", "Rep 3"),
               names_to = "replicate",
               values_to = "conc_day1") %>%
  clean_names()
```

```
## # A tibble: 1,635 x 4
##    cow_id hormone_trt replicate conc_day1
##    <chr>  <chr>       <chr>     <chr>
##  1 08693  Low         Rep 1     <NA>
##  2 08693  Low         Rep 2     <NA>
##  3 08693  Low         Rep 3     2.33620793032117
##  4 13102  Low         Rep 1     2.39625917019355
##  5 13102  Low         Rep 2     <NA>
##  6 13102  Low         Rep 3     2.74505984432731
##  7 07745  Low         Rep 1     <NA>
##  8 07745  Low         Rep 2     0.893256859419714
##  9 07745  Low         Rep 3     <NA>
## 10 13144  Low         Rep 1     <NA>
## # ... with 1,625 more rows
```

Filtering

filter() from the {dplyr} package allows us to extract rows based on matching. In R, "==" means equal to and "!=" means NOT equal to.

For example, if we wanted to subset the data to replicate 1 only:

```
read_excel(here("data/sample-data.xlsx"),
           sheet = "Day 1",
           skip = 3,
           na = ".") %>%
  pivot_longer(cols = c("Rep 1", "Rep 2", "Rep 3"),
               names_to = "replicate",
               values_to = "conc_day1") %>%
  clean_names() %>%
  filter(replicate == "Rep 1")
```

```
## # A tibble: 545 x 4
##    cow_id hormone_trt replicate conc_day1
##    <chr>  <chr>       <chr>     <chr>
##  1 08693  Low         Rep 1     <NA>
##  2 13102  Low         Rep 1     2.39625917019355
##  3 07745  Low         Rep 1     <NA>
##  4 13144  Low         Rep 1     <NA>
##  5 13066  Low         Rep 1     <NA>
##  6 12831  Low         Rep 1     <NA>
##  7 14263  Low         Rep 1     <NA>
##  8 12923  Low         Rep 1     <NA>
##  9 08796  Low         Rep 1     <NA>
## 10 14298  Low         Rep 1     <NA>
## # ... with 535 more rows
```

Or to all replicates except replicate 1:

```
read_excel(here("data/sample-data.xlsx"),
           sheet = "Day 1",
           skip = 3,
           na = ".") %>%
  pivot_longer(cols = c("Rep 1", "Rep 2", "Rep 3"),
               names_to = "replicate",
               values_to = "conc_day1") %>%
  clean_names() %>%
  filter(replicate != "Rep 1")
```

```
## # A tibble: 1,090 x 4
##    cow_id hormone_trt replicate conc_day1
##    <chr>  <chr>       <chr>     <chr>
##  1 08693  Low         Rep 2     <NA>
##  2 08693  Low         Rep 3     2.33620793032117
##  3 13102  Low         Rep 2     <NA>
##  4 13102  Low         Rep 3     2.74505984432731
##  5 07745  Low         Rep 2     0.893256859419714
##  6 07745  Low         Rep 3     <NA>
##  7 13144  Low         Rep 2     2.32101837926091
##  8 13144  Low         Rep 3     <NA>
##  9 13066  Low         Rep 2     0.852166834158408
## 10 13066  Low         Rep 3     2.20700171568918
## # ... with 1,080 more rows
```

Pivoting our data from wide to long format created many rows with no value in the `conc_day1` column, which aren't helpful to us. To remove these using `filter()`, we need to give a slightly different command. The `NA` value is special in R, so to do this we need to use the `is.na()` function along with the exclamation point (which again, means "not").

```
read_excel(here("data/sample-data.xlsx"),
           sheet = "Day 1",
           skip = 3,
           na = ".") %>%
  pivot_longer(cols = c("Rep 1", "Rep 2", "Rep 3"),
               names_to = "replicate",
               values_to = "conc_day1") %>%
  clean_names() %>%
  filter(!is.na(conc_day1))
```

```
## # A tibble: 631 x 4
##    cow_id hormone_trt replicate conc_day1
##    <chr>  <chr>       <chr>     <chr>
##   1 08693  Low         Rep 3     2.33620793032117
##   2 13102  Low         Rep 1     2.39625917019355
##   3 13102  Low         Rep 3     2.74505984432731
##   4 07745  Low         Rep 2     0.893256859419714
##   5 13144  Low         Rep 2     2.32101837926091
##   6 13066  Low         Rep 2     0.852166834158408
##   7 13066  Low         Rep 3     2.20700171568918
##   8 12831  Low         Rep 2     0.991173203186964
##   9 14263  Low         Rep 2     0.588676304573575
##  10 12923  Low         Rep 2     3.14628738079207
## # ... with 621 more rows
```

To assign the imported, pivoted, and filtered day 1 data with R-friendly column names to an object called `d1`, use the the pointy arrow `<-`

```
d1 <-
  read_excel(here("data/sample-data.xlsx"),
             sheet = "Day 1",
             skip = 3,
             na = ".") %>%
  pivot_longer(cols = c("Rep 1", "Rep 2", "Rep 3"),
               names_to = "replicate",
               values_to = "conc_day1") %>%
  clean_names() %>%
  filter(!is.na(conc_day1))
```

Joining the day 1 data and the day 3 data using

First, import, pivot, and filter the data in the day 3 sheet in the same way as the day 1 sheet. Assign it to an object called `d3`.

```
d3 <-
  read_excel(here("data/sample-data.xlsx"),
             sheet = "Day 3",
             skip = 3,
             na = ".") %>%
  pivot_longer(cols = c("Rep 1", "Rep 2", "Rep 3"),
               names_to = "replicate",
               values_to = "conc_day3") %>%
  clean_names() %>%
  filter(!is.na(conc_day3))
```

Next, we want to combine the Day 1 and Day 3 data.

Here, `x` is the first data frame (`d1`), `y` is the second data frame (`d3`), and `by` is the common columns between `x` and `y` that we want to join on (`replicate`, `hormone_trt` and `cow_id`).

```
full_join(x = d1,
          y = d3,
          by = c("replicate", "hormone_trt", "cow_id"))
```

```
## # A tibble: 637 x 5
##    cow_id hormone_trt replicate conc_day1         conc_day3
##    <chr>  <chr>       <chr>     <chr>             <chr>
##  1 08693  Low         Rep 3     2.33620793032117  3.344
##  2 13102  Low         Rep 1     2.39625917019355  1.10754772273572
##  3 13102  Low         Rep 3     2.74505984432731  2.35030411058928
##  4 07745  Low         Rep 2     0.893256859419714 2.24013988872224
##  5 13144  Low         Rep 2     2.32101837926091  4.39514509744692
##  6 13066  Low         Rep 2     0.852166834158408 2.45448784231195
##  7 13066  Low         Rep 3     2.20700171568918  3.23679481915272
##  8 12831  Low         Rep 2     0.991173203186964 7.94542418771408
##  9 14263  Low         Rep 2     0.588676304573575 6.37368824755747
## 10 12923  Low         Rep 2     3.14628738079207  5.88472204739773
## # ... with 627 more rows
```

You might notice that some cows in some replicates had an E2 observation on one day but not the other. If we wanted to exclude cows that had an observation on day 3 but not day 1, we could provide `left_join()` the same arguments we gave to `full_join()` above. Below, the data frame `d1` is the `x` (or "left hand side") argument.

```
left_join(x = d1,
          y = d3,
          by = c("replicate", "hormone_trt", "cow_id"))
```

```
## # A tibble: 631 x 5
##    cow_id hormone_trt replicate conc_day1         conc_day3
##    <chr>  <chr>       <chr>     <chr>             <chr>
##  1 08693  Low         Rep 3     2.33620793032117  3.344
##  2 13102  Low         Rep 1     2.39625917019355  1.10754772273572
##  3 13102  Low         Rep 3     2.74505984432731  2.35030411058928
##  4 07745  Low         Rep 2     0.893256859419714 2.24013988872224
##  5 13144  Low         Rep 2     2.32101837926091  4.39514509744692
##  6 13066  Low         Rep 2     0.852166834158408 2.45448784231195
##  7 13066  Low         Rep 3     2.20700171568918  3.23679481915272
##  8 12831  Low         Rep 2     0.991173203186964 7.94542418771408
##  9 14263  Low         Rep 2     0.588676304573575 6.37368824755747
## 10 12923  Low         Rep 2     3.14628738079207  5.88472204739773
## # ... with 621 more rows
```

Notice that the resulting data frame above has fewer rows than when we used `full_join()`. This is because six cows had an observation on day 3, but not day 1. If we wanted instead exclude cows that had an observation on day 1 but not day 3, we could make `d3` the left hand side argument in `left_join()`.

```
left_join(x = d3,
          y = d1,
          by = c("replicate", "hormone_trt", "cow_id"))
```

```
## # A tibble: 637 x 5
##    cow_id hormone_trt replicate conc_day3        conc_day1
##    <chr>  <chr>       <chr>     <chr>            <chr>
##  1 08693  Low         Rep 3     3.344            2.33620793032117
##  2 13102  Low         Rep 1     1.10754772273572 2.39625917019355
##  3 13102  Low         Rep 3     2.35030411058928 2.74505984432731
##  4 07745  Low         Rep 2     2.24013988872224 0.893256859419714
##  5 13144  Low         Rep 2     4.39514509744692 2.32101837926091
##  6 13066  Low         Rep 2     2.45448784231195 0.852166834158408
##  7 13066  Low         Rep 3     3.23679481915272 2.20700171568918
##  8 12831  Low         Rep 2     7.94542418771408 0.991173203186964
##  9 14263  Low         Rep 2     6.37368824755747 0.588676304573575
## 10 12923  Low         Rep 2     5.88472204739773 3.14628738079207
## # ... with 627 more rows
```

Mutating

As mentioned previously, our data is from a study of the effects of hormone supplementation on pregnancy rate in cattle. Use `mutate()` from the `{dplyr}` package to calculate how much hormone concentration changed after supplementation for each row then store the results in a new column called `change`.

```
full_join(x = d1,
          y = d3,
          by = c("replicate", "hormone_trt", "cow_id")) %>%
  mutate(change = conc_day3 - conc_day1)
```

```
## Error: Problem with `mutate()` input `change`.
## x non-numeric argument to binary operator
## i Input `change` is `conc_day3 - conc_day1`.
```

Right now, R thinks that the columns `conc_day1` and `conc_day2` contain character values (like words) rather than numbers, so we get an error when we try to use those columns to do math. We can also use `mutate()` to change `conc_day1` and `conc_day2` to numeric columns. Assign the joined and mutated data frame to an object called `d1_d3`.

```
d1_d3 <-
  full_join(x = d1,
            y = d3,
            by = c("replicate", "hormone_trt", "cow_id")) %>%
  mutate(conc_day1 = as.numeric(conc_day1),
         conc_day3 = as.numeric(conc_day3),
         change = conc_day3 - conc_day1)
```

Grouping and summarizing

Similar to Microsoft Excel's pivot table functions, we can use `summarise()` from the `{dplyr}` package to quickly generate summaries of data. For example, if we wanted to calculate the mean of all values in the `change` column, we'd do:

```
d1_d3 %>%
  summarise(mean(change, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   `mean(change, na.rm = TRUE)`
##                          <dbl>
## 1                         1.86
```

The `na.rm` argument tells `mean()` to ignore missing values. Without it, `summarise()` would return an `NA`. Just like all of the other `{dplyr}` functions we've used, `summarise()` returns a data frame. This means we can change the column name in the summary data frame.

```
d1_d3 %>%
  summarise(mean_change = mean(change, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   mean_change
##         <dbl>
## 1         1.86
```

In addition to the mean change, summarize the minimum and maximum `change` values.

```
d1_d3 %>%
  group_by(hormone_trt) %>%
  summarise(mean_change = mean(change, na.rm = TRUE),
            min_change = min(change, na.rm = TRUE),
            max_change = max(change, na.rm = TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 4
##   hormone_trt mean_change min_change max_change
##   <chr>             <dbl>      <dbl>      <dbl>
## 1 High               1.77      -2.36       6.23
## 2 Low                1.80      -3.02      10.5
## 3 Middle             2.05      -3.16       7.08
```

We can also use `group_by()` to summarize data within different levels of a variable…

```
d1_d3 %>%
  group_by(hormone_trt) %>%
  summarise(mean_change = mean(change, na.rm = TRUE),
            min_change = min(change, na.rm = TRUE),
            max_change = max(change, na.rm = TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 4
##   hormone_trt mean_change min_change max_change
##   <chr>             <dbl>      <dbl>      <dbl>
## 1 High               1.77      -2.36       6.23
## 2 Low                1.80      -3.02      10.5
## 3 Middle             2.05      -3.16       7.08
```

…or within different levels of multiple variables.

```
d1_d3 %>%
  group_by(hormone_trt, replicate) %>%
  summarise(mean_change = mean(change, na.rm = TRUE),
            min_change = min(change, na.rm = TRUE),
            max_change = max(change, na.rm = TRUE))
```

```
## `summarise()` regrouping output by 'hormone_trt' (override with `.groups` argument)
```

```
## # A tibble: 9 x 5
## # Groups:   hormone_trt [3]
##   hormone_trt replicate mean_change min_change max_change
##   <chr>       <chr>           <dbl>      <dbl>      <dbl>
## 1 High        Rep 1            1.84      -2.27       5.97
## 2 High        Rep 2            1.57      -2.36       5.56
## 3 High        Rep 3            1.99      -1.79       6.23
## 4 Low         Rep 1            1.76      -1.50      10.5
## 5 Low         Rep 2            2.03      -3.02       6.95
## 6 Low         Rep 3            1.55      -2.05       7.92
## 7 Middle      Rep 1            1.82      -2.60       6.88
## 8 Middle      Rep 2            2.04      -1.77       6.06
## 9 Middle      Rep 3            2.31      -3.16       7.08
```

Finally, we can use `tally()` to generate simple counts of data. Using `tally()` is equivalent to using `summarise(n = n())`.

```
d1_d3 %>%
  group_by(hormone_trt) %>%
  tally()
```

```
## # A tibble: 3 x 2
##   hormone_trt     n
##   <chr>       <int>
## 1 High          210
## 2 Low           265
## 3 Middle        162
```

`group_by()` is not just for summarizing. It can be used in combination with almost all of the tools mentioned here to perform actions within groups. Often, I use `group_by()` to sample observations (i.e., to subset data down to n randomly selected rows per group). Remember to `ungroup()` data once you're done performing group-wise actions!

More resources:

There's an almost overwhelming amount of resources for R users ranging from casual to incredibly specific. Here are a few resources (other than the ones mentioned in the slides) pertaining to the topics we covered today that I often refer back to.

- `{tidyverse}` cheatsheets: I have these printed and bound and take them literally EVERYWHERE
- Sharla Gelfand's "Strategies for working with new data"
- `{readxl}` workflows
- Suzan Baert did a series of tutorials in early 2018 that I refer back to more often than the official manuals for the packages and functions she covers. I'd HIGHLY recommend checking them out:
  - Suzan Baert's "Data Wrangling Part 1: Basic to Advanced Ways to Select Columns"
  - Suzan Baert's "Data Wrangling Part 2: Transforming your columns into the right shape"
  - Suzan Baert's "Data Wrangling Part 3: Basic and more advanced ways to filter rows"
  - Suzan Baert's "Data Wrangling Part 4: Summarizing and slicing your data"
- A `{dplyr}` tool for more complicated column mutating not mentioned today, but that I use almost daily called `case_when()`:
  - Official manual