

Working with health data in R and RStudio

Corinne Riddell

August 28, 2020

Learning objectives for today:

1. What is a data frame
2. How to read a comma separated values (CSV) file using `read_csv()`
3. Get to know the data using `str()`, `head()`, `dim()`, and `names()`
4. Manipulate the data frame using the R package `dplyr`'s main functions:
 - `rename()`
 - `select()`
 - `arrange()`
 - `filter()`
 - `mutate()`
 - `group_by()`
 - `summarize()`

Readings

- There are no chapters from the textbook for this lecture.
- Here are some additional online resources (optional, but helpful!):
 - Data Frames
 - 15 min intro to dplyr
 - Data wrangling cheat sheet

What is a data frame?

- A data frame is a data set.
- We read data into R from common sources like Excel spreadsheets (.xls or .xlsx), text files (.txt), comma separate value files (.csv), and other formats.
- The simplest format of data contains one row for each individual in the study.
- The first column of the data identifies the **individual** (perhaps by a name or an ID **variable**).
- Subsequent columns are **variables** that have been recorded or measured.

Lake data from Baldi and Moore (B&M)

- Exercise 1.25 from Edition 4 of B&M
- Six rows of data from a study of mercury concentration across 53 lakes
- I've added three fabricated rows
- I've placed these data in Day-2 folder
- Let's find it there

readr is a library to import data into R

- To access `readr`'s functions we load the library like this:

```
library(readr)
```

- Click the green arrow to run the code or place your cursor on the line of code and type `cmd + enter` (Mac) or `control + enter` (PC)
- A green rectangle that temporarily appears next to the code shows you that it has run.

`read_csv()` to load the lake data in R

- `read_csv()` is a function from the `readr` library used to import csv files.
- code template: `your_data <- read_csv("pathway_to_data.csv")`
- The `<-` is called the **assignment operator**. It says to save the imported data into an object called `your_data`.

```
lake_data <- read_csv("Data_mercury_lake.csv")
```

```
## Parsed with column specification:
## cols(
##   lakes = col_character(),
##   ph = col_double(),
##   chlorophyll = col_double(),
##   mercury_in_fish = col_double(),
##   number_fish_sampled = col_double(),
##   age_data = col_character()
## )
```

- Anytime you see “##” on the html slides or in the PDF lecture files, the text in those lines are the **output** of running the code in the previous line. So the lines above are the output displayed when you run the `read_csv()` function.

Exercise 1

1. Execute the above code using either the green arrow or by clicking on it and hitting the keyboard shortcut (`cmd + enter` on mac or `Ctrl + enter` on PC).
2. Note that the data appears in the Environment pane in the top right.
 - Notice the number of **observations** and the number of **variables**.
3. Click the tiny table icon to the right of the `lake_data` in the Environment pane to open the **Viewer** tab and inspect the data.

Check your understanding!

Four functions to get to know a dataset

- `head(your_data)`: Shows the first six rows of the supplied dataset
- `dim(your_data)`: Provides the number of rows by the number of columns
- `names(your_data)`: Lists the variable names of the columns in the dataset
- `str(your_data)`: Summarizes the above information and more

I use these functions all the time! Multiple times per session when working with data to remind me what the variable names are, and what the data looks like.

`head()`

First six rows:

```
head(lake_data)
```

```
## # A tibble: 6 x 6
##   lakes          ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>        <dbl>      <dbl>      <dbl>          <dbl> <chr>
## 1 Alligator    6.1        0.7        1.23            5 year old
## 2 Annie        5.1        3.2        1.33            7 recent
## 3 Apopka       9.1       128.        0.04            6 recent
## 4 Blue Cypress 6.9        3.5        0.44           12 recent
## 5 Brick        4.6        1.8        1.2            12 year old
## 6 Bryant       7.3       44.1        0.27           14 year old
```

`dim()`

```
dim(lake_data)
```

```
## [1] 9 6
```

Are there 9 rows or columns of data?

`names()`

```
names(lake_data)
```

```
## [1] "lakes"          "ph"              "chlorophyll"
## [4] "mercury_in_fish" "number_fish_sampled" "age_data"
```

`str()`

```
str(lake_data)
```

```
## tibble [9 x 6] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ lakes      : chr [1:9] "Alligator" "Annie" "Apopka" "Blue Cypress" ...
## $ ph         : num [1:9] 6.1 5.1 9.1 6.9 4.6 7.3 5.5 7.3 8.2
## $ chlorophyll : num [1:9] 0.7 3.2 128.3 3.5 1.8 ...
## $ mercury_in_fish : num [1:9] 1.23 1.33 0.04 0.44 1.2 0.27 0.33 0.17 1.87
## $ number_fish_sampled: num [1:9] 5 7 6 12 12 14 5 8 3
## $ age_data     : chr [1:9] "year old" "recent" "recent" "recent" ...
## - attr(*, "spec")=
## .. cols(
## ..   lakes = col_character(),
## ..   ph = col_double(),
## ..   chlorophyll = col_double(),
## ..   mercury_in_fish = col_double(),
## ..   number_fish_sampled = col_double(),
## ..   age_data = col_character()
## .. )
```

Using `dplyr` functions for data manipulation

- `rename()`
- `select()`
- `arrange()`
- `filter()`
- `mutate()`
- `group_by()`
- `summarize()`

Load the dplyr library to access the functions

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
##
## The following objects are masked from 'package:stats':
##
##   filter, lag
##
## The following objects are masked from 'package:base':
##
##   intersect, setdiff, setequal, union
```

- These messages mean that some functions (e.g., filter()) share names with functions from other libraries. So, when we use filter() we will now use the dplyr version because the stats library version has been masked.
- You don't need to worry about masking for now.

Function 1: rename()

What do you think rename does?

First print the names of the variables:

```
names(lake_data)
```

```
## [1] "lakes"          "ph"              "chlorophyll"
## [4] "mercury_in_fish" "number_fish_sampled" "age_data"
```

Run the rename() function and assign it to lake_data_tidy:

```
lake_data_tidy <- rename(lake_data, name_of_lake = lakes)
```

Then reprint the variable names:

```
names(lake_data_tidy)
```

```
## [1] "name_of_lake"    "ph"              "chlorophyll"
## [4] "mercury_in_fish" "number_fish_sampled" "age_data"
```

Function 1: rename() multiple variables at once

You can rename() multiple variables at once:

```
lake_data_tidy <- rename(lake_data,
                        name_of_lake = lakes,
                        ph_level = ph)
```

Code template for rename() function

```
new_dataset <- rename(old_dataset, new_name = old_name)
```

Another way to write the above code is to use the pipe operator (%>%):

```
new_dataset <- old_dataset %>% rename(new_name = old_name)
```

The pipe will become very useful in a few slides

Function 2: select()

Based on the output below, what do you think `select()` does?

```
smaller_data <- select(lake_data, lakes, ph, chlorophyll)
names(smaller_data)
```

```
## [1] "lakes"      "ph"         "chlorophyll"
```

Function 2: select()

- We use `select()` to select a subset of **variables**.
- This is very handy if we inherit a large dataset with several variables that we do not need.

Function 2: “negative select()”

We can also use “negative `select()`” to deselect variables. Suppose we wanted to keep all variables except for `age_data`:

```
smaller_data_2 <- select(lake_data, - age_data)
names(smaller_data_2)
```

```
## [1] "lakes"      "ph"         "chlorophyll"
## [4] "mercury_in_fish" "number_fish_sampled"
```

We place a negative sign in front of `age_data` to remove it from the dataset.

Rewrite using the pipe operator

```
smaller_data <- lake_data %>% select(lakes, ph, chlorophyll)
smaller_data_2 <- lake_data %>% select(- age_data)
```

- Going forward, we will use the pipe operator to write code using any `dplyr` functions
- This is because we can use the pipe to stack many `dplyr` functions in a row

Function 3: arrange()

What does `arrange` do? First type `View(lake_data)` to look at the original data. Then run the code and examine its output below. What is different?:

```
lake_data %>% arrange(ph)
```

```
## # A tibble: 9 x 6
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>    <dbl>      <dbl>         <dbl>          <dbl> <chr>
## 1 Brick      4.6        1.8           1.2            12 year old
## 2 Annie      5.1        3.2           1.33           7 recent
## 3 Catalina    5.5       13.2           0.33           5 recent
## 4 Alligator   6.1        0.7           1.23           5 year old
## 5 Blue Cypress 6.9        3.5           0.44          12 recent
## 6 Bryant      7.3       44.1           0.27          14 year old
## 7 Four Mile   7.3        0.4           0.17           8 recent
## 8 Henry       8.2       12.2           1.87           3 year old
## 9 Apopka      9.1       128.           0.04           6 recent
```

Function 3: arrange() in descending order

```
lake_data %>% arrange(- ph)
```

```
## # A tibble: 9 x 6
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>    <dbl>    <dbl>        <dbl>          <dbl> <chr>
## 1 Apopka    9.1      128.          0.04             6 recent
## 2 Henry     8.2      12.2          1.87             3 year old
## 3 Bryant    7.3      44.1          0.27            14 year old
## 4 Four Mile  7.3        0.4          0.17             8 recent
## 5 Blue Cypress 6.9        3.5          0.44            12 recent
## 6 Alligator  6.1        0.7          1.23             5 year old
## 7 Catalina   5.5      13.2          0.33             5 recent
## 8 Annie      5.1        3.2          1.33             7 recent
## 9 Brick      4.6        1.8          1.2            12 year old
```

Function 3: arrange() by two variables

```
lake_data %>% arrange(age_data, ph)
```

```
## # A tibble: 9 x 6
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>    <dbl>    <dbl>        <dbl>          <dbl> <chr>
## 1 Annie      5.1        3.2          1.33             7 recent
## 2 Catalina   5.5      13.2          0.33             5 recent
## 3 Blue Cypress 6.9        3.5          0.44            12 recent
## 4 Four Mile  7.3        0.4          0.17             8 recent
## 5 Apopka    9.1      128.          0.04             6 recent
## 6 Brick      4.6        1.8          1.2            12 year old
## 7 Alligator  6.1        0.7          1.23             5 year old
## 8 Bryant    7.3      44.1          0.27            14 year old
## 9 Henry     8.2      12.2          1.87             3 year old
```

Function 4: mutate()

- **mutate() is one of the most useful functions!**
- It is used to add new variables to the dataset. Suppose that someone told you that the number of fish sampled was actually in hundreds, such that 5 is actually 500. You can use mutate to add a new variable to your dataset that is in the hundreds:

```
lake_data_new_fish <- lake_data %>%
  mutate(actual_fish_sampled = number_fish_sampled * 100)
```

```
lake_data_new_fish
```

```
## # A tibble: 9 x 7
##   lakes      ph chlorophyll mercury_in_fish number_fish_sam~ age_data
##   <chr> <dbl>    <dbl>        <dbl>          <dbl> <chr>
## 1 Alli~  6.1        0.7          1.23             5 year old
## 2 Annie  5.1        3.2          1.33             7 recent
## 3 Apop~  9.1      128.          0.04             6 recent
## 4 Blue~  6.9        3.5          0.44            12 recent
## 5 Brick  4.6        1.8          1.2            12 year old
## 6 Brya~  7.3      44.1          0.27            14 year old
```

```
## 7 Cata~    5.5      13.2      0.33      5 recent
## 8 Four~    7.3       0.4      0.17      8 recent
## 9 Henry    8.2      12.2      1.87      3 year old
## # ... with 1 more variable: actual_fish_sampled <dbl>
```

Use %>% to append several lines of code together

- We have saved many of new datasets in our environment!
- If these datasets were larger, they would take up a lot of space.
- Rather than saving a new dataset each time, we can make successive changes to one dataset like this:

```
tidy_lake_data <- lake_data %>%
  rename(name_of_lake = lakes) %>%
  mutate(actual_fish_sampled = number_fish_sampled * 100) %>%
  select(- age_data, - number_fish_sampled)
```

- When you see %>%, say the words “**and then...**”. For example, “Take lake_data **and then** rename lakes to name_of_lake, **and then** mutate...”

Use %>% to “pipe” several lines of code together

```
tidy_lake_data <- lake_data %>%
  rename(lake_name = lakes) %>%
  mutate(actual_fish_sampled = number_fish_sampled * 100) %>%
  select(- age_data, - number_fish_sampled)
```

```
tidy_lake_data
```

```
## # A tibble: 9 x 5
##   lake_name      ph chlorophyll mercury_in_fish actual_fish_sampled
##   <chr>      <dbl>      <dbl>      <dbl>      <dbl>
## 1 Alligator    6.1        0.7        1.23        500
## 2 Annie        5.1        3.2        1.33        700
## 3 Apopka       9.1       128.        0.04        600
## 4 Blue Cypress 6.9         3.5        0.44       1200
## 5 Brick        4.6         1.8        1.2        1200
## 6 Bryant       7.3       44.1        0.27       1400
## 7 Catalina     5.5       13.2        0.33        500
## 8 Four Mile    7.3         0.4        0.17        800
## 9 Henry        8.2       12.2        1.87        300
```

Function 5: filter()

Filter is another very useful function! What might filter() do?

Function 5: filter()ing on numeric variables

We use filter to select which rows we want to keep in the dataset. Suppose you were only interested in lakes with ph levels of 7 or higher.

```
lake_data_filtered <- lake_data %>% filter(ph > 7)
lake_data_filtered
```

```
## # A tibble: 4 x 6
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>    <dbl>      <dbl>      <dbl>      <dbl> <chr>
```

## 1 Apopka	9.1	128.	0.04	6 recent
## 2 Bryant	7.3	44.1	0.27	14 year old
## 3 Four Mile	7.3	0.4	0.17	8 recent
## 4 Henry	8.2	12.2	1.87	3 year old

Check your understanding!

Function 5: filter()ing on character/string variables

Let's try a few more ways to filter() the data set since subsetting data is so important:

```
lake_data %>% filter(age_data == "recent")
```

```
## # A tibble: 5 x 6
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>    <dbl>    <dbl>    <dbl>    <dbl> <chr>
## 1 Annie      5.1      3.2      1.33      7 recent
## 2 Apopka      9.1     128.      0.04      6 recent
## 3 Blue Cypress 6.9      3.5      0.44     12 recent
## 4 Catalina    5.5     13.2      0.33      5 recent
## 5 Four Mile   7.3      0.4      0.17      8 recent
```

- == is read as “is equal to”

```
lake_data %>% filter(age_data != "recent")
```

```
## # A tibble: 4 x 6
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>    <dbl>    <dbl>    <dbl>    <dbl> <chr>
## 1 Alligator 6.1      0.7      1.23      5 year old
## 2 Brick     4.6      1.8      1.2     12 year old
## 3 Bryant    7.3     44.1      0.27     14 year old
## 4 Henry     8.2     12.2      1.87      3 year old
```

- != is read as “is not equal to”

Function 5: filter()ing on character/string variables

```
lake_data %>% filter(lakes %in% c("Alligator", "Blue Cypress"))
```

```
## # A tibble: 2 x 6
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>    <dbl>    <dbl>    <dbl>    <dbl> <chr>
## 1 Alligator 6.1      0.7      1.23      5 year old
## 2 Blue Cypress 6.9      3.5      0.44     12 recent
```

- %in% is the “in” operator. We are selecting rows where the variable lakes belongs to the specified list.
- The c() combines “Alligator” and “Blue Cypress” into a list

Function 5: multiple filter()s at once

```
lake_data %>% filter(ph > 6, chlorophyll > 30)
```

```
## # A tibble: 2 x 6
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>    <dbl>    <dbl>    <dbl>    <dbl> <chr>
## 1 Apopka 9.1     128.      0.04      6 recent
```



```
## 2 Bryant    7.3      44.1      0.27      14 year old
```

#this is the same as:

```
lake_data %>% filter(ph > 6 & chlorophyll > 30)
```

```
## # A tibble: 2 x 6
```

```
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>   <dbl>      <dbl>      <dbl>      <dbl> <chr>
## 1 Apopka   9.1      128.        0.04         6 recent
## 2 Bryant   7.3      44.1        0.27        14 year old
```

- A comma or the “and” operator (&) are equivalent. Here they say, filter the dataset and keep only rows with `ph > 6` AND `chlorophyll > 30`

Function 5: filter() using “or”

```
lake_data %>% filter(ph > 6 | chlorophyll > 30)
```

```
## # A tibble: 6 x 6
```

```
##   lakes      ph chlorophyll mercury_in_fish number_fish_sampled age_data
##   <chr>   <dbl>      <dbl>      <dbl>      <dbl> <chr>
## 1 Alligator 6.1        0.7        1.23         5 year old
## 2 Apopka    9.1      128.        0.04         6 recent
## 3 Blue Cypress 6.9        3.5        0.44        12 recent
## 4 Bryant    7.3      44.1        0.27        14 year old
## 5 Four Mile 7.3        0.4        0.17         8 recent
## 6 Henry     8.2      12.2        1.87         3 year old
```

- | is the OR operator. At least one of `ph > 6` or `chlorophyll > 30` needs to be true.

Functions 6 and 7: group_by() and summarize()

Let’s execute the following code and see what it does.

```
lake_data %>%
  group_by(age_data) %>%
  summarize(mean_ph = mean(ph))
```

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

```
## # A tibble: 2 x 2
```

```
##   age_data mean_ph
##   <chr>      <dbl>
## 1 recent    6.78
## 2 year old  6.55
```

What happened?

Functions 6 and 7: group_by() and summarize()

Another one:

```
lake_data %>%
  group_by(age_data) %>%
  summarize(mean_ph = mean(ph),
            standard_deviation_ph = sd(ph))
```

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

```
## # A tibble: 2 x 3
##   age_data mean_ph standard_deviation_ph
##   <chr>      <dbl>                <dbl>
## 1 recent      6.78                1.59
## 2 year old    6.55                1.56
```

Recap: What functions did we use?

1. `library()` to load `readr` and `dplyr`.
2. `read_csv()` to read csv files from a directory.
3. `head()`, `str()`, `dim()`, and `names()` to look at our imported data.
4. `rename()` to rename variables in a data frame.
5. `select()` to select a subset of variables.
6. `arrange()` to sort a dataset according to one or more variables.
7. `mutate()` to create new variables.
8. `filter()` to select a subset of rows.
9. `group_by()` and `summarize()` to group the data by a categorical variable and calculate a statistic.
10. `mean()` and `sd()` to calculate the mean and standard deviation of variables.

Recap: What operators did we use?

1. Assignment arrow (`<-`): This is our most important operator!
2. Greater than (`>`) There are also:
 - Less than (`<`)
 - Greater than or equal to (`>=`), and,
 - Less than or equal to: (`<=`)
3. Is equal to (`==`), and is not equal to (`!=`)
4. `%in%` to select from a list, where the list is created using `c()`, i.e., `lakes %in% c("Alligator", "Annie")`

How to export from datahub and save onto your own computer

Some of you may want to edit this file in R markdown by adding notes, etc. In that case, you can make your edits and save your updated file on datahub. You can additionally save your updated file locally on your computer. Here's how to do that:

1. In the File view window, click the checkbox beside the file you'd like to export and then click More > Export. This will download the file to your computer's downloads folder.
2. You may want to Export slides as a PDF or MS Word document. To do that, you first need to change "slidy_presentation" to "pdf_document" or "word_document" in the file header (line 5 of the file, after the word "output:"). Make sure to keep the single space between "output:" and your option or it won't compile!
3. Word documents automatically download when you Knit them. PDF documents can be exported from the File viewer by following step 1.