Wrangling, Analyzing and Exporting Data with the Tidyverse Handout

Data Carpentry contributors & Montana State University R Workshops Team

11/5/2020

Orientation of/for the Workshop

- This workshop assumes some basic familiarity with working in `R` such as what you might obtain in the "Introduction to R" workshop or in a statistics course that uses `R` heavily, such as STAT 217 or STAT 411/511. If you have not interacted with `R` previously, some of the assumptions of your background for this workshop might be a barrier. We would recommend getting what you can from this workshop and you can always revisit the materials at a later date after filling in some of those basic `R` skills. We all often revisit materials and discover new and deeper aspects of the content that we were not in a position to appreciate in a first exposure.
- In order to focus this workshop on coding, we developed this interactive website for you to play in a set of "sandboxes" and try your hand at implementing the methods we are discussing. When each code chunk is ready to run (all can be edited, many have the code prepared for you), you can click on "Run Code". This will run `R` in the background on a server. For the "Challenges", you can get your answer graded although many have multiple "correct" answers, so don't be surprised if our "correct" answer differs from yours. The "Solution" is also provided in some cases so you can see a solution - but you will learn more by trying the challenge first before seeing the answer. Each sandbox functions independently, which means that you can pick up working at any place in the documents and re-set your work without impacting other work (this is VERY different from how `R` usually works!). Hopefully this allows you to focus on the code and what it does... The "Start over" button can be used on any individual sandbox or you can use the one on the left tile will re-set all the code chunks to the original status.
- These workshops are taught by Sara Mannheimer, Greta Linse, and Mark Greenwood and co-organized by the MSU Library, Statistical Consulting and Research Services (SCRS), and the Department of Mathematical Sciences. More details on us and other workshops are available at the end of the session.

Let's get started!

Learning Objectives

- Describe the purpose of the **`dplyr`** and **`tidyr`** packages.
- Select certain columns in a data frame with the **`dplyr`** function `select`.
- Select certain rows in a data frame according to filtering conditions with the **`dplyr`** function `filter` .
- Link the output of one **`dplyr`** function to the input of another function with the 'pipe' operator **`%>%`**.
- Add new columns to a data frame that are functions of existing columns with `mutate`.
- Use the split-apply-combine concept for producing data summaries.
- Use `summarize`, `group_by`, and `count` to split a data frame into groups of observations, apply summary statistics for each group, and then combine the results.

- Describe the concept of a wide and a long table format and for which purpose those formats are useful.
- Describe what key-value pairs are.
- Reshape a data frame from long to wide format and back with the `pivot_wider` and `pivot_longer` commands from the **`tidyr`** package.
- Export a data frame to a .csv file.

Data Wrangling using `dplyr` & `tidyr` Intro

Note that we're not using "data manipulation" for this workshop, but are calling it "data wrangling." To us, "data manipulation" is a term that captures the event where a researcher manipulates their data (e.g., moving columns, deleting rows, merging data files) in a **non-reproducible** manner. Whereas, with data wrangling, all of these process are done, but in a **reproducible** manner, such as using an `R` script!

Packages in `R` are basically sets of additional functions that let you do more stuff. The functions we've been using so far, like `str()` or `data.frame()`, come built into `R`; packages give you access to more of them. Before you use a package for the first time you need to install it on your machine, and then you should import it in every subsequent `R` session when you need it. You should already have installed the **`tidyverse`** package. This is an "umbrella-package" that installs several packages useful for data analysis which work together well such as **`tidyr`**, **`dplyr`**, **`ggplot2`**, **`tibble`**, etc.

The **`tidyverse`** package tries to address 3 common issues that arise when doing data analysis with some of the functions that come with `R`:

1. The results from a base `R` function sometimes depend on the type of data.
2. Using `R` expressions in a non-standard way, which can be confusing for new learners.
3. Hidden arguments, having default operations that new learners are not aware of.

We have seen in our previous lesson that when building or importing a data frame, the columns that contain characters (i.e., text) are coerced (=converted) into the `factor` data type. We had to set **`stringsAsFactors`** to **`FALSE`** to avoid this hidden argument to convert our data type.

This time we will use the **`tidyverse`** package to read the data and avoid having to set **`stringsAsFactors`** to **`FALSE`**

In order to install the `tidyverse` package, you can type `install.packages("tidyverse")` straight into the RStudio console. In fact, it's better to write this in the console than in a script for any package, as there's no need to re-install packages every time we run the script. If you work in a .Rmd (R-markdown) format, any missing packages will be identified and a prompt added to the top of the document about installing the packages.

Then, to load the package we would need to type:

```
## load the tidyverse packages -- including dplyr, tidyr, readr, stringr
library(tidyverse)
```

What are `dplyr` and `tidyr`?

The package **`dplyr`** is built to work directly with data frames, with many common tasks optimized by being written in a compiled language (C++). An additional feature is the ability to work directly with data stored in an external database. The benefits of doing this are that the data can be managed natively in a relational database, queries can be conducted on that database, and only the results of the query are returned.

This addresses a common problem with `R` in that all operations are conducted in-memory and thus the amount of data you can work with is limited by available memory. The database connections essentially remove that limitation in that you can connect to a database of many hundreds of GB, conduct queries on it directly, and pull back into `R` only what you need for analysis.

The package **tidyr** addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per measurement. Sometimes we want a data frame where each measurement type has its own column, and rows are instead more aggregated groups - like plots or aquaria. Moving back and forth between these formats is nontrivial, and **tidyr** gives you tools for this and more sophisticated data wrangling.

To learn more about **dplyr** and **tidyr** after the workshop, you may want to check out this handy data transformation with **dplyr** cheatsheet and this cheatsheet about **tidyr**.

Presentation of the Survey Data

The data used in this workshop are a time-series for a small mammal community in southern Arizona. This is part of a project studying the effects of rodents and ants on the plant community that has been running for almost 40 years, but we will focus on the years 1996 to 2002 (n=11332 observations). The rodents are sampled on a series of 24 plots, with different experimental manipulations controlling which rodents are allowed to access which plots. This is simplified version of the full data set that has been used in over 100 publications and was provided by the Data Carpentries (https://datacarpentry.org/ecology-workshop/data/). We are investigating the animal species diversity and weights found within plots in this workshop. The dataset is stored as a comma separated value (CSV) file. Each row holds information for a single animal, and the columns represent:

Column	Description
record_id	Unique id for the observation
month	month of observation
day	day of observation
year	year of observation
plot_id	ID of a particular plot
species_id	2-letter code
sex	sex of animal ("M", "F")
hindfoot_length	length of the hindfoot in mm
weight	weight of the animal in grams

We'll read in our data using the `read_csv()` function, from the tidyverse package **readr**, instead of `read.csv()`.

```
surveys <- read_csv("https://raw.githubusercontent.com/saramannheimer/data-science-r-workshops/master/Da
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   record_id = col_double(),
##   month = col_double(),
##   day = col_double(),
##   year = col_double(),
##   plot_id = col_double(),
##   species_id = col_character(),
##   sex = col_character(),
##   hindfoot_length = col_double(),
##   weight = col_double()
## )
```

You will see the message `Parsed with column specification`, followed by each column name and its data type. When you execute `read_csv` on a data file, it looks through the first 1000 rows of each column and guesses the data type for each column as it reads it into R. For example, in this dataset, `read_csv` reads

`weight` as `col_double` (a numeric data type), and `species` as `col_character`. You have the option to specify the data type for a column manually by using the `col_types` argument in `read_csv`.

```
## inspect the data
glimpse(surveys)
```

```
## Rows: 11,332
## Columns: 9
## $ record_id       <dbl> 23215, 23216, 23217, 23218, 23220, 23221, 23222, 23...
## $ month           <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day             <dbl> 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27,...
## $ year            <dbl> 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 199...
## $ plot_id         <dbl> 21, 1, 17, 17, 2, 18, 1, 2, 17, 2, 1, 12, 21, 18, 1...
## $ species_id      <chr> "PF", "DM", "DM", "DM", "DM", "PF", "DM", "DO", "DM...
## $ sex             <chr> "F", "M", "M", NA, "F", "F", "M", "M", "F", "M", "F...
## $ hindfoot_length <dbl> 16, NA, 36, 37, 36, NA, 34, 37, 39, 40, 27, 39, 21,...
## $ weight          <dbl> 7, 27, 25, NA, 47, 9, 27, 66, 49, 54, 38, NA, 16, 9...
```

```
## Preview the data (opens a spreadsheet-like interface in RStudio)
View(surveys)
```

```
## # A tibble: 11,332 x 9
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     23215     1    27  1996      21 PF         F                  16      7
##  2     23216     1    27  1996       1 DM         M                  NA     27
##  3     23217     1    27  1996      17 DM         M                  36     25
##  4     23218     1    27  1996      17 DM         <NA>               37     NA
##  5     23220     1    27  1996       2 DM         F                  36     47
##  6     23221     1    27  1996      18 PF         F                  NA      9
##  7     23222     1    27  1996       1 DM         M                  34     27
##  8     23223     1    27  1996       2 DO         M                  37     66
##  9     23224     1    27  1996      17 DM         F                  39     49
## 10     23225     1    27  1996       2 DM         M                  40     54
## # ... with 11,322 more rows
```

Notice that the class of the data is now `tbl_df`

This is referred to as a "tibble". Tibbles tweak some of the behaviors of the data frame objects we introduced previously. The data structure is very similar to a data frame. For our purposes the only differences are that:

1. In addition to displaying the data type of each column under its name, it only prints the first few rows of data and only as many columns as fit on one screen.
2. Columns of class `character` are never converted into factors.

We're going to learn some of the most common **`dplyr`** functions:

- `select()`: subset columns
- `filter()`: subset rows on conditions
- `mutate()`: create new columns by using information from other columns
- `group_by()` and `summarize()`: create summary statistics on grouped data
- `arrange()`: sort results
- `count()`: count discrete values

Select, Filter, and Mutate

Selecting Columns and Filtering Rows

To select columns of a data frame, use `select()`. The first argument to this function is the data frame (`surveys`), and the subsequent arguments are the columns to keep.

Modify the following code to select the `plot_id`, `species_id`, and `weight` columns from the `survey` dataset:

```
select(surveys)
```

```
## # A tibble: 11,332 x 0
```

To select all columns *except* certain ones, put a "-" in front of the variable to exclude it.

Modify the following code to select all columns *except* `record_id` and `species_id`:

```
select(surveys)
```

```
## # A tibble: 11,332 x 0
```

This will select all the variables in `surveys` except `record_id` and `species_id`.

To choose rows based on a specific criteria, use `filter()`:

```
filter(surveys, year == 1999)
```

```
## # A tibble: 1,064 x 9
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     29024     1    16  1999       1 DM         F                  33     41
##  2     29025     1    16  1999       1 DM         F                  35     52
##  3     29026     1    16  1999       1 DM         M                  35     52
##  4     29027     1    16  1999       1 DO         M                  36     55
##  5     29028     1    16  1999       1 DO         F                  33     53
##  6     29029     1    16  1999       2 DO         M                  36     50
##  7     29030     1    16  1999       2 OT         M                  20     22
##  8     29031     1    16  1999       2 OT         M                  20     26
##  9     29032     1    16  1999       2 DO         F                  34     46
## 10     29033     1    16  1999       2 DO         F                  35     51
## # ... with 1,054 more rows
```

In the code above `==` keeps all rows where the year is 1999.

Other filtering options include `!=`, which keeps all rows that are **not** a certain criteria, `,` which means **"and"**, and `|` which means **"or"**. Filter can also do `<` for "less than", `>` for "greater than", `<=` for "less than or equal to", and `>=` for "greater than or equal to". We type these last two options the same way we would typically say them.

1. `!=` example:

   ```
   filter(surveys, year != 1999)
   ```

   ```
   ## # A tibble: 10,268 x 9
   ##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
   ##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
   ##  1     23215     1    27  1996      21 PF         F                  16      7
   ##  2     23216     1    27  1996       1 DM         M                  NA     27
   ##  3     23217     1    27  1996      17 DM         M                  36     25
   ##  4     23218     1    27  1996      17 DM         <NA>               37     NA
   ##  5     23220     1    27  1996       2 DM         F                  36     47
   ##  6     23221     1    27  1996      18 PF         F                  NA      9
   ```

```
##  7     23222     1    27  1996       1 DM         M                   34     27
##  8     23223     1    27  1996       2 DO         M                   37     66
##  9     23224     1    27  1996      17 DM         F                   39     49
## 10     23225     1    27  1996       2 DM         M                   40     54
## # ... with 10,258 more rows
```

The code above keeps all rows where the year is not 1999.

2. , example:

```
filter(surveys, year == 1999 , plot_id == 2)
```

```
## # A tibble: 57 x 9
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     29029     1    16  1999       2 DO         M                  36     50
##  2     29030     1    16  1999       2 OT         M                  20     22
##  3     29031     1    16  1999       2 OT         M                  20     26
##  4     29032     1    16  1999       2 DO         F                  34     46
##  5     29033     1    16  1999       2 DO         F                  35     51
##  6     29034     1    16  1999       2 OT         F                  20     25
##  7     29035     1    16  1999       2 PE         M                  20     18
##  8     29036     1    16  1999       2 DM         M                  36     44
##  9     29037     1    16  1999       2 DM         M                  37     47
## 10     29039     1    16  1999       2 NL         F                  34    162
## # ... with 47 more rows
```

The code above keeps all rows where the year is 1999 for plot id 2, i.e., year 1999 and plot 2. The rows meet **both** of these criteria.

3. | example:

```
filter(surveys, year == 1999 | plot_id == 2)
```

```
## # A tibble: 1,743 x 9
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     23220     1    27  1996       2 DM         F                  36     47
##  2     23223     1    27  1996       2 DO         M                  37     66
##  3     23225     1    27  1996       2 DM         M                  40     54
##  4     23234     1    27  1996       2 DM         F                  35     45
##  5     23237     1    27  1996       2 DM         M                  35     46
##  6     23239     1    27  1996       2 PB         M                  29     46
##  7     23242     1    27  1996       2 DO         M                  36     54
##  8     23243     1    27  1996       2 DM         M                  36     49
##  9     23257     1    27  1996       2 DM         M                  36     50
## 10     23258     1    27  1996       2 PE         M                  20     25
## # ... with 1,733 more rows
```

The code above keeps all rows where the year is 1999 **or** is plot id 2, i.e., year 1999 or plot 2. The rows meet **either** of these criteria but not both.

4. < example:

```
filter(surveys, weight < 8)
```

```
## # A tibble: 163 x 9
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
```

```
##  1     23215     1    27  1996      21 PF         F                    16      7
##  2     23240     1    27  1996      20 PF         M                    15      6
##  3     23250     1    27  1996      21 PF         M                    15      6
##  4     23271     1    28  1996      13 PF         F                    16      7
##  5     23283     1    28  1996       3 PF         M                    15      5
##  6     23317     1    28  1996       6 PF         M                    15      7
##  7     23330     1    28  1996       6 PF         F                    15      7
##  8     23334     1    28  1996       9 PF         M                    17      7
##  9     23380     2    24  1996      12 PF         F                    14      7
## 10     23436     2    25  1996       5 PF         F                    16      7
## # ... with 153 more rows
```

The code above keeps all rows where weight is less than 8.

5. > example:

```
filter(surveys, hindfoot_length > 30)
```

```
## # A tibble: 3,828 x 9
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     23217     1    27  1996      17 DM         M                  36     25
##  2     23218     1    27  1996      17 DM         <NA>               37     NA
##  3     23220     1    27  1996       2 DM         F                  36     47
##  4     23222     1    27  1996       1 DM         M                  34     27
##  5     23223     1    27  1996       2 DO         M                  37     66
##  6     23224     1    27  1996      17 DM         F                  39     49
##  7     23225     1    27  1996       2 DM         M                  40     54
##  8     23227     1    27  1996      12 DM         <NA>               39     NA
##  9     23230     1    27  1996      17 DM         M                  36     51
## 10     23231     1    27  1996      22 DM         F                  36     43
## # ... with 3,818 more rows
```

The code above keeps all rows where hindfoot length is greater than 30.

Pipes

What if you want to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

With intermediate steps, you create a temporary data frame and use that as input to the next function, like this:

```
surveys2 <- filter(surveys, weight < 6)
surveys_sml <- select(surveys2, species_id, sex, weight)
```

This is readable, but can clutter up your workspace with lots of objects that you have to name individually. With multiple steps, that can be hard to keep track of.

You can also nest functions (i.e., one function inside of another), like this:

```
surveys_sml <- select(filter(surveys, weight < 6), species_id, sex, weight)
```

This is handy, but can be difficult to read if too many functions are nested, as R evaluates the expression from the inside out (in this case, filtering, then selecting).

The last option, *pipes*, are a more recent addition to `R`. Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipes in `R` look like `%>%` and are made available via the **`magrittr`** package, installed automatically with **`dplyr`**. If you

use RStudio, you can type the pipe with Ctrl + Shift + M if you have a PC or Cmd + Shift + M if you have a Mac.

```
surveys %>%
  filter(weight < 6) %>%
  select(species_id, sex, weight)
```

```
## # A tibble: 7 x 3
##   species_id sex   weight
##   <chr>      <chr>  <dbl>
## 1 PF         M          5
## 2 PF         M          5
## 3 PF         F          5
## 4 PF         F          5
## 5 PF         F          5
## 6 PP         M          4
## 7 PF         F          5
```

In the above code, we use the pipe to send the `surveys` dataset first through `filter()` to keep rows where `weight` is less than 6, then through `select()` to keep only the `species_id`, `sex`, and `weight` columns. Since `%>%` takes the object on its left and passes it as the first argument to the function on its right, we don't need to explicitly include the data frame as an argument to the `filter()` and `select()` functions any more.

Some may find it helpful to read the pipe like the word "then". For instance, in the above example, we took the data frame `surveys`, *then* we `filter`ed for rows with `weight < 6`, *then* we `select`ed columns `species_id`, `sex`, and `weight`. The `dplyr` functions by themselves are somewhat simple, but by combining them into linear workflows with the pipe, we can accomplish more complex wrangling of data frames.

If we want to create a new object with this smaller version of the data, we can assign it a new name:

```
surveys_sml <- surveys %>%
  filter(weight < 6) %>%
  select(species_id, sex, weight)

surveys_sml
```

```
## # A tibble: 7 x 3
##   species_id sex   weight
##   <chr>      <chr>  <dbl>
## 1 PF         M          5
## 2 PF         M          5
## 3 PF         F          5
## 4 PF         F          5
## 5 PF         F          5
## 6 PP         M          4
## 7 PF         F          5
```

Note that the final data frame is the leftmost part of this expression.

Challenge 1

Using pipes, subset the `surveys` data to include:

- animals collected on or after 2001 and
- retain only the columns `year`, `sex`, and `weight`.

```
## Pipes Challenge:
##  Using pipes, subset the data to include animals collected
##  on or after 2001, and retain the columns `year`, `sex`, and `weight.`
```

Mutate

Frequently you'll want to create new columns based on the values in existing columns, for example to do unit conversions, or to find the ratio of values in two columns. For this we'll use `mutate()`.

To create a new column of weight in kg from weight in grams:

```
surveys %>%
  mutate(weight_kg = weight / 1000)
```

```
## # A tibble: 11,332 x 10
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     23215     1    27  1996      21 PF         F                  16      7
##  2     23216     1    27  1996       1 DM         M                  NA     27
##  3     23217     1    27  1996      17 DM         M                  36     25
##  4     23218     1    27  1996      17 DM         <NA>               37     NA
##  5     23220     1    27  1996       2 DM         F                  36     47
##  6     23221     1    27  1996      18 PF         F                  NA      9
##  7     23222     1    27  1996       1 DM         M                  34     27
##  8     23223     1    27  1996       2 DO         M                  37     66
##  9     23224     1    27  1996      17 DM         F                  39     49
## 10     23225     1    27  1996       2 DM         M                  40     54
## # ... with 11,322 more rows, and 1 more variable: weight_kg <dbl>
```

You can also create a second new column based on the first new column within the same call of `mutate()`:

```
surveys %>%
  mutate(weight_kg = weight / 1000,
         weight_lb = weight_kg * 2.2)
```

```
## # A tibble: 11,332 x 11
##    record_id month   day  year plot_id species_id sex   hindfoot_length weight
##        <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
##  1     23215     1    27  1996      21 PF         F                  16      7
##  2     23216     1    27  1996       1 DM         M                  NA     27
##  3     23217     1    27  1996      17 DM         M                  36     25
##  4     23218     1    27  1996      17 DM         <NA>               37     NA
##  5     23220     1    27  1996       2 DM         F                  36     47
##  6     23221     1    27  1996      18 PF         F                  NA      9
##  7     23222     1    27  1996       1 DM         M                  34     27
##  8     23223     1    27  1996       2 DO         M                  37     66
##  9     23224     1    27  1996      17 DM         F                  39     49
## 10     23225     1    27  1996       2 DM         M                  40     54
## # ... with 11,322 more rows, and 2 more variables: weight_kg <dbl>,
## #   weight_lb <dbl>
```

If this runs off your screen and you just want to see the first few rows, you can use a pipe to view the `head()` of the data. (Pipes work with non-**dplyr** functions, too, as long as the **dplyr** or `magrittr` package is loaded).

```
surveys %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

```
## # A tibble: 6 x 10
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 1     23215     1    27  1996      21 PF         F                  16      7
## 2     23216     1    27  1996       1 DM         M                  NA     27
## 3     23217     1    27  1996      17 DM         M                  36     25
## 4     23218     1    27  1996      17 DM         <NA>               37     NA
## 5     23220     1    27  1996       2 DM         F                  36     47
## 6     23221     1    27  1996      18 PF         F                  NA      9
## # ... with 1 more variable: weight_kg <dbl>
```

The first few rows of the data set contain some missing observations (`NA`s). If we wanted to remove any observations where there were missing values on `weight`, we could insert a `filter()` in the chain:

```
surveys %>%
  filter(!is.na(weight)) %>%
  mutate(weight_kg = weight / 1000) %>%
  head()
```

```
## # A tibble: 6 x 10
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 1     23215     1    27  1996      21 PF         F                  16      7
## 2     23216     1    27  1996       1 DM         M                  NA     27
## 3     23217     1    27  1996      17 DM         M                  36     25
## 4     23220     1    27  1996       2 DM         F                  36     47
## 5     23221     1    27  1996      18 PF         F                  NA      9
## 6     23222     1    27  1996       1 DM         M                  34     27
## # ... with 1 more variable: weight_kg <dbl>
```

`is.na()` is a function that determines whether something is an `NA`. The `!` symbol negates the result, so in the code above we're asking for every row where weight *is not* an `NA`.

Challenge 2 

Create a new data frame from the `surveys` data named `surveys_hindfoot_cm` that meets the following criteria:

- contains only the `species_id` column and
- a new column called `hindfoot_cm` containing the `hindfoot_length` values converted to centimeters (they are in mm).
- Make sure that you only retain values in the hindfoot_cm column that are not missing (not NA) and are less than 3 cm.
- Then print out the `head()` of the new data frame.

Hint: think about how the commands should be ordered to produce this data frame!

```
## Mutate Challenge:
##  Create a new data frame from the `surveys` data named `surveys_hindfoot_cm`
##  that meets the following criteria:
##  * contains only the `species_id` column and
##  * a new column called `hindfoot_cm` containing the `hindfoot_length` values
##    converted to centimeters.
##  * Make sure that you only retain values in the hindfoot_cm column that are
##    not missing (not NA) and are less than 3 cm.
##  Then print out the head of the new data frame.
```

```
##  Hint: think about how the commands should be ordered to produce this data frame!
```

Using `lubridate` for Dates

Date-time data can be frustrating to work with in `R`, since `R` commands for date-times are generally un-intuitive and change depending on the type of date-time object being used. Moreover, the methods we use with date-times must be robust to time zones, leap days, daylight savings times, and other time related quirks, and `R` lacks these capabilities in some situations. The `lubridate` package makes it easier to do the things `R` does with date-times and possible to do things that base `R` does not.

`Lubridate` has functions that handle easy parsing of times, such as:

- `ymd()`
- `dmy()`
- `mdy()`

```
library(lubridate)
today() # Today's date
```

```
## [1] "2021-03-28"
```

```
now() # Today's date, with time and timezone!
```

```
## [1] "2021-03-28 12:25:23 MDT"
```

```
surveys_w_days <- surveys %>%
  mutate(date = ymd(paste(year,
                          month,
                          day,
                          sep = "-")
                    ),
         day_of_week = wday(date, label = TRUE)
         ## Creating a day of the week variable
         ## label = TRUE prints the name, not the level!
         )
```

```
## Warning: Problem with `mutate()` input `date`.
## i  125 failed to parse.
## i Input `date` is `ymd(paste(year, month, day, sep = "-"))`.
```

```
## Warning: 125 failed to parse.
```

```
surveys_w_days %>%
  head()
```

```
## # A tibble: 6 x 11
##   record_id month   day  year plot_id species_id sex   hindfoot_length weight
##       <dbl> <dbl> <dbl> <dbl>   <dbl> <chr>      <chr>           <dbl>  <dbl>
## 1     23215     1    27  1996      21 PF         F                  16      7
## 2     23216     1    27  1996       1 DM         M                  NA     27
## 3     23217     1    27  1996      17 DM         M                  36     25
## 4     23218     1    27  1996      17 DM         <NA>               37     NA
## 5     23220     1    27  1996       2 DM         F                  36     47
## 6     23221     1    27  1996      18 PF         F                  NA      9
## # ... with 2 more variables: date <date>, day_of_week <ord>
```

```
surveys_w_days %>%
  select(day_of_week) %>%
```

```
  summary()
```

```
##   day_of_week
##  Sun    :4212
##  Sat    :4202
##  Wed    : 924
##  Mon    : 572
##  Thu    : 525
##  (Other): 772
##  NA's   : 125
```

```
surveys_w_days %>%
  filter(is.na(date) == TRUE) %>%
  select(month, day) %>%
  table()
```

```
##       day
## month 31
##     4 70
##     9 55
```

Challenge 3 

- What dates were unable to be converted?
- Explore the results and objects in the previous sandbox to figure out why that happened.

We can pull off components of dates using a large array of `lubridate` functions, such as:

- `year()`
- `month()`
- `mday()`
- `hour()`
- `minute()`
- `second()`

For additional information about `lubridate` visit the `lubridate` reference website or look over the `lubridate` cheatsheet.

Character Wrangling

If we inspect the day of week variable we created in the last code chunk, we'll see that it is an ordered (<ord>) factor.

Challenge 4 

What are the names of the days of the week taken from the dates?

The `case_when()` Function

We notice that the labels for the days of the week are not necessarily what we would like to have for a graphical display of our data. To reword the names of the days of the week, we can use the `case_when()` function from `dplyr`.

The `case_when()` function can be thought of as a "generalized form for multiple `if_else()` statements." We talked about `ifelse()` statements in the *Intermediate R* workshop, but let's break them down here to review.

For `case_when()` the inputs are sequences of two-sided formulas. The left hand side finds the values that match the case and the right hand side says what should be done with these matches.

Let's look at this in action!

```
surveys_days_full <- surveys_w_days %>%
  mutate(day_of_week = case_when(day_of_week == "Mon" ~ "Monday",
                                 day_of_week == "Tue" ~ "Tuesday",
                                 day_of_week == "Wed" ~ "Wednesday",
                                 day_of_week == "Thu" ~ "Thursday",
                                 day_of_week == "Fri" ~ "Friday",
                                 day_of_week == "Sat" ~ "Saturday",
                                 day_of_week == "Sun" ~ "Sunday")
         )
glimpse(surveys_days_full$day_of_week)
```

```
##  chr [1:11332] "Saturday" "Saturday" "Saturday" "Saturday" "Saturday" ...
```

NOTE:

If you only want to recode a couple levels of a variable, you can still use `case_when()` without specifying the behavior for **ALL** levels. See the example below:

```
# Create a variable weekday that takes on a value of 0 for Saturday/Sunday
# and 1 otherwise and recodes Friday to missing
surveys_weekday <- surveys_w_days %>%
  mutate(weekday = case_when(day_of_week == "Sat" ~ 0,
                             day_of_week == "Sun" ~ 0,
                             day_of_week == "Fri" ~ as.numeric(NA),
                             TRUE ~ 1))

surveys_weekday %>%
  count(weekday)
```

```
## # A tibble: 3 x 2
##   weekday     n
##     <dbl> <int>
## 1       0  8414
## 2       1  2589
## 3      NA   329
```

But, perhaps these days are not in the order that we want them to be in.

Challenge 5 

What order did `R` put the days of the week in? What data type is `day_of_week` now?

There are small differences between character data types and factor data types. Typically, `R` uses factors to handle categorical variables, variables that have a fixed and known set of possible values. Factors are also helpful for reordering character vectors to improve display. However, factors are often difficult to work with. Enter the `forcats` package, whose goal is to provide a suite of tools that solve common problems with factors, including changing the order of levels or the values.

The order of the levels `R` chose may not be what we wanted, but we can reorder them using the `fct_relevel()` function from the `forcats` package. The function takes three arguments:

1. the data
2. the factor to be reordered
3. the order of the new levels separated by commas

This process looks like this:

```
surveys_edited <- surveys_days_full %>%
  mutate(day_of_week = fct_relevel(day_of_week,
                                   "Monday",
                                   "Tuesday",
                                   "Wednesday",
                                   "Thursday",
                                   "Friday",
                                   "Saturday",
                                   "Sunday")
  )
glimpse(surveys_edited$day_of_week)
```

```
##  Factor w/ 7 levels "Monday","Tuesday",..: 6 6 6 6 6 6 6 6 6 6 ...
```

Challenge 6

Verify that `R` put the days in the order that you specified!

Split-Apply-Combine Data Analysis

Many data analysis tasks can be approached using the *split-apply-combine* paradigm: split the data into groups, apply some analysis to each group, and then combine the results. **`dplyr`** makes this very easy through the use of the `group_by()` function.

The `summarize()` Function

`group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group. `group_by()` takes as arguments the column names that contain the **categorical** variables for which you want to calculate the summary statistics. So to compute the mean `weight` by sex:

```
surveys_edited %>%
  group_by(sex) %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE))
```

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

```
## # A tibble: 3 x 2
##   sex   mean_weight
##   <chr>       <dbl>
## 1 F            33.1
## 2 M            33.3
## 3 <NA>        NaN
```

One of the advantages of `tbl_df` over data frame is that is provides more compact output, although the current format of these materials makes that hard to see.

You can also group by multiple columns:

```
surveys_edited %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight, na.rm = TRUE))
```

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

```
## # A tibble: 25 x 3
## # Groups:   sex [3]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     DM                43.6
##  2 F     DO                49.4
##  3 F     NL               168.
##  4 F     OL                32.1
##  5 F     OT                25.3
##  6 F     PB                30.2
##  7 F     PE                22.5
##  8 F     PF                 8.44
##  9 F     PM                22.0
## 10 F     PP                17.5
## # ... with 15 more rows
```

When grouping both by `sex` and `species_id`, the last row is for animals that escaped before their sex and body weights could be determined. You may notice that the last column does not contain `NA` but `NaN` (which refers to "Not a Number"). To avoid this, we can remove the missing values for weight before we attempt to calculate the summary statistics on weight. Because the missing values are removed first, we can omit `na.rm = TRUE` when computing the mean:

```
surveys_edited %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight))
```

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

```
## # A tibble: 24 x 3
## # Groups:   sex [2]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     DM                43.6
##  2 F     DO                49.4
##  3 F     NL               168.
##  4 F     OL                32.1
##  5 F     OT                25.3
##  6 F     PB                30.2
##  7 F     PE                22.5
##  8 F     PF                 8.44
##  9 F     PM                22.0
## 10 F     PP                17.5
## # ... with 14 more rows
```

If you want to display more data, you can use the `print()` function at the end of your chain with the argument `n` specifying the number of rows to display:

```
surveys_edited %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
```

```
  summarize(mean_weight = mean(weight)) %>%
  print(n = 15)
```

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

```
## # A tibble: 24 x 3
## # Groups:   sex [2]
##    sex   species_id mean_weight
##    <chr> <chr>            <dbl>
##  1 F     DM               43.6
##  2 F     DO               49.4
##  3 F     NL              168.
##  4 F     OL               32.1
##  5 F     OT               25.3
##  6 F     PB               30.2
##  7 F     PE               22.5
##  8 F     PF                8.44
##  9 F     PM               22.0
## 10 F     PP               17.5
## 11 F     RM               11.9
## 12 F     SH               77.4
## 13 M     DM               45.1
## 14 M     DO               48.5
## 15 M     NL              167.
## # ... with 9 more rows
```

Once the data are grouped, you can also summarize multiple variables at the same time (and not necessarily on the same variable). For instance, we could add a column indicating the minimum weight for each species for each sex:

```
surveys_edited %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight))
```

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

```
## # A tibble: 24 x 4
## # Groups:   sex [2]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 F     DM               43.6          19
##  2 F     DO               49.4          22
##  3 F     NL              168.           63
##  4 F     OL               32.1          21
##  5 F     OT               25.3          11
##  6 F     PB               30.2          12
##  7 F     PE               22.5          11
##  8 F     PF                8.44          5
##  9 F     PM               22.0           9
## 10 F     PP               17.5           8
## # ... with 14 more rows
```

It is sometimes useful to rearrange the result of a query to inspect the values. For instance, we can sort on `min_weight` to put the lighter species first:

```
surveys_edited %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(min_weight)
```

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

```
## # A tibble: 24 x 4
## # Groups:   sex [2]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 M     PP               17.1           4
##  2 F     PF                8.44          5
##  3 M     PF                8.39          5
##  4 F     RM               11.9           7
##  5 M     PM               20.3           7
##  6 M     RM               10.8           7
##  7 F     PP               17.5           8
##  8 M     PE               20.3           8
##  9 F     PM               22.0           9
## 10 F     OT               25.3          11
## # ... with 14 more rows
```

To sort in descending order, we need to add the `desc()` function. If we want to sort the results by decreasing order of mean weight:

```
surveys_edited %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight),
            min_weight = min(weight)) %>%
  arrange(desc(mean_weight))
```

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

```
## # A tibble: 24 x 4
## # Groups:   sex [2]
##    sex   species_id mean_weight min_weight
##    <chr> <chr>            <dbl>      <dbl>
##  1 F     NL               168.          63
##  2 M     NL               167.          62
##  3 F     SH                77.4         38
##  4 M     SH                59.1         28
##  5 F     DO                49.4         22
##  6 M     DO                48.5         23
##  7 M     DM                45.1         18
##  8 F     DM                43.6         19
##  9 M     PB                33.8         13
## 10 F     OL                32.1         21
## # ... with 14 more rows
```

Challenge 7 Part 1: 

Use `group_by()` and `summarize()` to find the mean, min, and max hindfoot length for each species (using `species_id`). Also add the number of observations. (HINT: see `?n`.)

Challenge 7 Part 2: 

What was the heaviest animal measured in each year?

Return the columns `year` and `weight`.

Counting

When working with data, we often want to know the number of observations found for each factor or combination of factors. For this task, **`dplyr`** provides `count()`. For example, if we wanted to count the number of rows of data for each sex, we would do:

```
surveys_edited %>%
    count(sex)
```

```
## # A tibble: 3 x 2
##   sex       n
##   <chr> <int>
## 1 F      5451
## 2 M      5879
## 3 <NA>      2
```

The `count()` function is shorthand for something we've already seen: grouping by a variable, and summarizing it by counting the number of observations in that group. In other words, `surveys %>% count()` is equivalent to:

```
surveys_edited %>%
    group_by(sex) %>%
    summarize(count = n())
```

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

```
## # A tibble: 3 x 2
##   sex   count
##   <chr> <int>
## 1 F      5451
## 2 M      5879
## 3 <NA>      2
```

For convenience, `count()` provides the `sort` argument:

```
surveys_edited %>%
    count(sex, sort = TRUE)
```

```
## # A tibble: 3 x 2
##   sex       n
##   <chr> <int>
## 1 M      5879
## 2 F      5451
## 3 <NA>      2
```

The previous example shows the use of `count()` to count the number of rows/observations for *one* factor (i.e., `sex`). If we wanted to count the *combination of factors*, such as `sex` and `species`, we would specify the first and the second factor as the arguments of `count()`:

```
surveys_edited %>%
  count(sex, species_id)
```

```
## # A tibble: 25 x 3
##    sex   species_id     n
##    <chr> <chr>      <int>
##  1 F     DM          1111
##  2 F     DO           389
##  3 F     NL           134
##  4 F     OL            10
##  5 F     OT           507
##  6 F     PB          1610
##  7 F     PE           102
##  8 F     PF           272
##  9 F     PM           208
## 10 F     PP           973
## # ... with 15 more rows
```

With the above code, we can proceed with `arrange()` to sort the table according to a number of criteria so that we have a better way to compare groups. For instance, we might want to arrange the table above in (i) an alphabetical order of the levels of the species and (ii) in descending order of the count:

```
surveys_edited %>%
  count(sex, species_id) %>%
  arrange(species_id, desc(n))
```

```
## # A tibble: 25 x 3
##    sex   species_id     n
##    <chr> <chr>      <int>
##  1 M     DM          1558
##  2 F     DM          1111
##  3 <NA>  DM             2
##  4 M     DO           611
##  5 F     DO           389
##  6 F     NL           134
##  7 M     NL            72
##  8 M     OL            18
##  9 F     OL            10
## 10 M     OT           523
## # ... with 15 more rows
```

From the table above, we may learn that, for instance, there are 72 observations of the *albigula* species (`species_id` = "NL") for males.

Challenge 8: 

How many animals were caught in each plot (`plot_id`) surveyed?

```
## Count Challenge:
##  How many animals were caught in each `plot_type` surveyed?
```

Relational Data with `dplyr`

It is rare that data analyses, especially with longitudinal measurements, involve only a single table of data. More typically, you have multiple tables of data, describing different aspects of your study. When you embark on analyzing your data, these different data tables need to be combined. Collectively, multiple tables of data are called *relational data*, as the data tables are not independent, rather they relate to each other.

Relations are defined between a pair of data tables. There are three families of joining operations: mutating joins, filtering joins, and set operations. Today we will focus on mutating joins.

The `survey` data have two other data tables they are related to: `plots` and `species`. Load in these data and inspect them to get an idea of how they relate to the `survey` data we've been working with.

```
plots <- read_csv("https://raw.githubusercontent.com/saramannheimer/data-science-r-workshops/master/Data
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   plot_id = col_double(),
##   plot_type = col_character()
## )
```

```
head(plots)
```

```
## # A tibble: 6 x 2
##   plot_id plot_type
##     <dbl> <chr>
## 1       1 Spectab exclosure
## 2       2 Control
## 3       3 Long-term Krat Exclosure
## 4       4 Control
## 5       5 Rodent Exclosure
## 6       6 Short-term Krat Exclosure
```

Table 2: Columns in the `plots.csv` file:

Column	Description
plot_id	ID of a particular plot
plot_type	type of plot

```
species <- read_csv("https://raw.githubusercontent.com/saramannheimer/data-science-r-workshops/master/Da
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   species_id = col_character(),
##   genus = col_character(),
##   species = col_character(),
##   taxa = col_character()
## )
```

```
head(species)
```

```
## # A tibble: 6 x 4
##   species_id genus           species         taxa
##   <chr>      <chr>           <chr>           <chr>
## 1 AB         Amphispiza      bilineata       Bird
```

```
## 2 AH         Ammospermophilus harrisi         Rodent
## 3 AS         Ammodramus       savannarum      Bird
## 4 BA         Baiomys          taylori         Rodent
## 5 CB         Campylorhynchus  brunneicapillus Bird
## 6 CM         Calamospiza      melanocorys     Bird
```

Table 3: Columns in the `species.csv` file:

Column	Description
species_id	2-letter code
genus	genus of animal
species	species of animal
taxon	e.g. Rodent, Reptile, Bird, Rabbit

The variables used to connect a pair of tables are called *keys*. A key is a variable that uniquely identifies an observation in that table. What are the keys for each of the three data tables? (hint: What combination of variables uniquely identifies a row in that data frame?)

```
quiz(
  question("What is the key for the plots data table?",
    answer("plot_id",correct=TRUE),
    answer("plot_type")
  ),
  question("What is the key for the species data table?",
    answer("species_id", correct = TRUE),
    answer("genus"),
    answer("species"),
    answer("taxa")
  ),
  question("What is the key for the surveys data table?",
    answer("record_id", correct = TRUE),
    answer("month"),
    answer("day"),
    answer("year"),
    answer("plot_id"),
    answer("species_id"),
    answer("sex"),
    answer("hindfoot_length"),
    answer("weight")
  )
)
```

There are two types of keys:

- A *primary key* uniquely identifies an observation in its own table.
- A *foreign key* uniquely identifies an observation in another table.

A primary key and the corresponding foreign key form a *relation* between the two data tables. These relations are typically many-to-one, though they can be 1-to-1. For example, there are many rodents captured that are of one species_id, hence a many-to-one relationship.

For me, the easiest way to think about the relationships between the different data tables is to draw a picture:

```
knitr::include_graphics("images/relations.jpg")
```

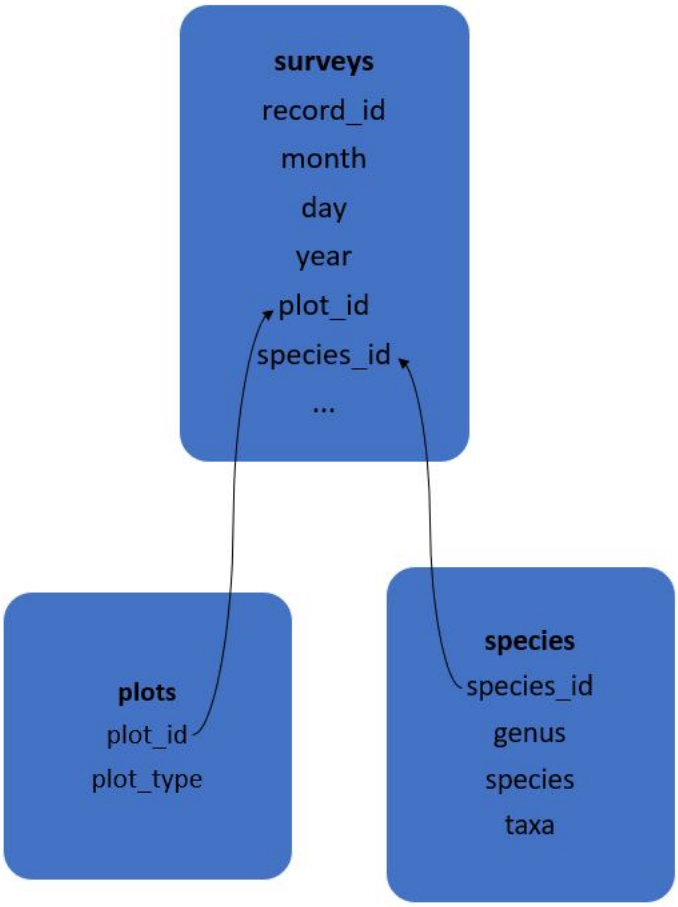


Figure 1: Relations of survey data tables

Joining Relational Data

The tool that we will be using is called a *mutating join*. A mutating join is how we can combine variables from two tables. The join matches observations by their keys, and then copies variables from one table to the other. Similar to `mutate()` these join functions add variables to the right of the existing data frame, hence their name. There are two types of mutating joins, the inner join and the outer join.

Inner Join

The simplest join is an *inner join*, which creates a pair of observations whenever their keys are equal. This join will output a new data frame that contains the key, the values of `x`, and the values of `y`. Importantly, this join deletes observations that do not have a match.

```
knitr::include_graphics("images/inner.jpg")
```

Outer Join

While an inner join only keeps observations with keys that appear in both tables, an *outer join* keeps observations that appear in *at least one* of the data tables. When joining `x` with `y`, there are three types of outer join:

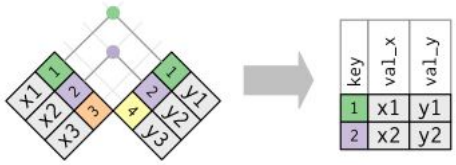


Figure 2: Wickham, H. and Grolemund, G. (2017) *R for Data Science*. Sebastopol, California: O'Reilly.

- A *left join* keeps all of the observations in `x`.
- A *right join* keeps all of the observations in `y`.
- A *full join* keeps all of the observations in both `x` and `y`.

```
knitr::include_graphics("images/joins.jpg")
```

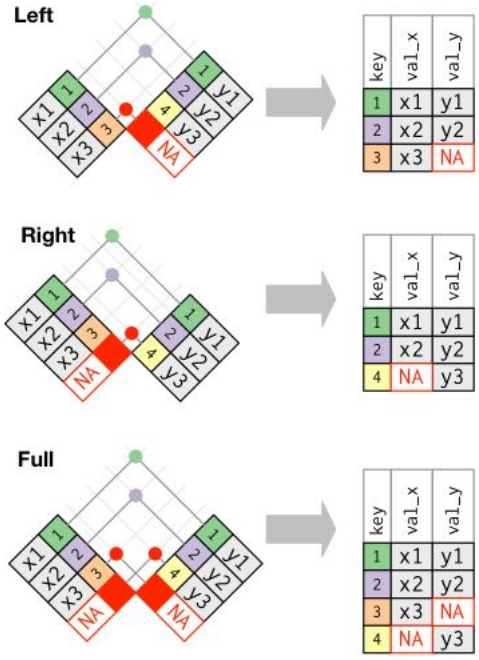


Figure 3: Wickham, H. and Grolemund, G. (2017) *R for Data Science*. Sebastopol, California: O'Reilly.

The left join is the most common, as you typically have a data frame (`x`) that you wish to add additional information to (the contents of `y`). This join will preserve the contents of `x`, even if there is not a match for them in `y`.

Joining `surveys_edited` Data

To join the `surveys_edited` data with the `plots` data and `species` data, we will need two join statements. As we are interested in adding this information to our already existing data frame, `surveys_edited`, a left join is the most appropriate.

```
combined <- surveys_edited %>%
              left_join(plots, by = "plot_id") %>%  # adding the type of plot
```

```
              left_join(species, by = "species_id") # adding the genus, species, and taxa
glimpse(combined)
```

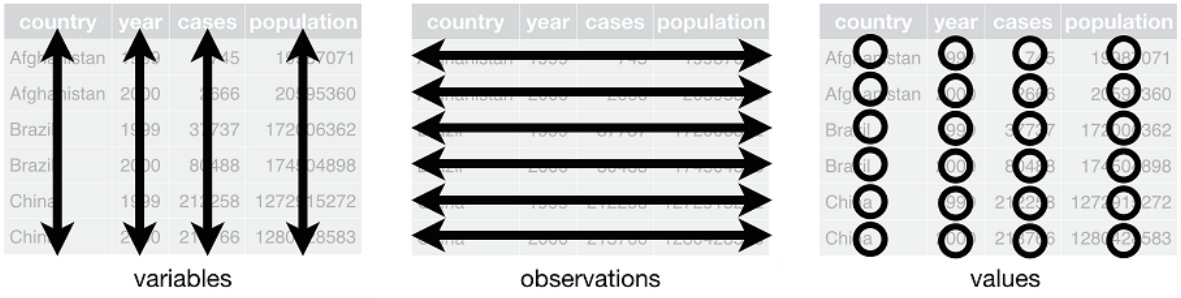
```
## Rows: 11,332
## Columns: 15
## $ record_id       <dbl> 23215, 23216, 23217, 23218, 23220, 23221, 23222, 23...
## $ month           <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day             <dbl> 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27,...
## $ year            <dbl> 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 199...
## $ plot_id         <dbl> 21, 1, 17, 17, 2, 18, 1, 2, 17, 2, 1, 12, 21, 18, 1...
## $ species_id      <chr> "PF", "DM", "DM", "DM", "DM", "PF", "DM", "DO", "DM...
## $ sex             <chr> "F", "M", "M", NA, "F", "F", "M", "M", "F", "M", "F...
## $ hindfoot_length <dbl> 16, NA, 36, 37, 36, NA, 34, 37, 39, 40, 27, 39, 21,...
## $ weight          <dbl> 7, 27, 25, NA, 47, 9, 27, 66, 49, 54, 38, NA, 16, 9...
## $ date            <date> 1996-01-27, 1996-01-27, 1996-01-27, 1996-01-27, 19...
## $ day_of_week     <fct> Saturday, Saturday, Saturday, Saturday, Saturday, S...
## $ plot_type       <chr> "Long-term Krat Exclosure", "Spectab exclosure", "C...
## $ genus           <chr> "Perognathus", "Dipodomys", "Dipodomys", "Dipodomys...
## $ species         <chr> "flavus", "merriami", "merriami", "merriami", "merr...
## $ taxa            <chr> "Rodent", "Rodent", "Rodent", "Rodent", "Rodent", "...
```

If the keys being used have different names in the data tables, you can use `by=c("a" = "b")` where `a` is the key name in the `x` data set and `b` is the name in the `y` data set. Or you could mutate the variable names so that they do match prior to using `left_join`.

Reshaping Data

Data Carpentry's spreadsheet lesson (link), discusses how to structure our data leading to the four rules defining a tidy dataset:

1. Each variable has its own column
2. Each observation has its own row
3. Each value must have its own cell
4. Each type of observational unit forms a table



Here we examine the fourth rule: Each type of observational unit forms a table.

In `surveys_edited` , the rows of `surveys_edited` contain the values of variables associated with each record (the unit), values such as the weight or sex of each animal associated with each record. What if instead of comparing records, we wanted to compare the different mean weights of each genus between plots? (Ignoring `plot_type` for simplicity).

We'd need to create a new table where each row (the unit) is comprised of values of variables associated with

each plot. In practical terms this means the values in `genus` would become the names of column variables and the cells would contain the values of the mean weight observed on each plot.

Having created a new table, it is therefore straightforward to explore the relationship between the weight of different genera within, and between, the plots. The key point here is that we are still following a tidy data structure, but we have **reshaped** the data according to the observations of interest: average genus weight per plot instead of recordings per date.

The opposite transformation would be to transform column names into values of a variable.

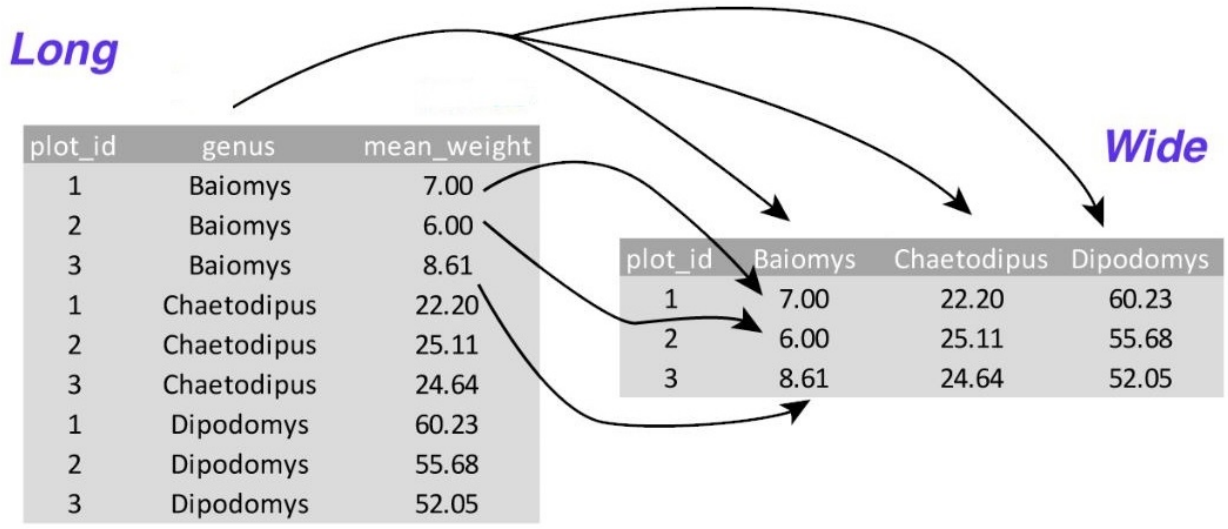
We can do both these of transformations with two `tidyr` functions, `pivot_longer()` and `pivot_wider()`.

Pivoting to a Wider Table

`pivot_wider()` takes three principal arguments:

1. the data
2. the column whose values will become new column names.
3. the column whose values will fill the new columns.

Further arguments include `fill` which, if set, fills in missing values with the value provided.



Let's use `pivot_wider()` to transform surveys to find the mean weight of each genus in each plot over the entire survey period. We use `filter()`, `group_by()` and `summarize()` to filter our observations and variables of interest, and create a new variable for the `mean_weight`. We use the pipe as before too.

```
surveys_gw <- combined %>%
  filter(!is.na(weight)) %>%
  group_by(plot_id, genus) %>%
  summarize(mean_weight = mean(weight))
```

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

```
glimpse(surveys_gw)
```

```
## Rows: 165
## Columns: 3
## Groups: plot_id [24]
## $ plot_id     <dbl> 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3...
```

```
## $ genus       <chr> "Chaetodipus", "Dipodomys", "Neotoma", "Onychomys", "Pe...
## $ mean_weight <dbl> 23.482625, 46.957377, 178.750000, 24.482143, 7.384615, ...
```

```
surveys_gw %>%
  head()
```

```
## # A tibble: 6 x 3
## # Groups:   plot_id [1]
##   plot_id genus       mean_weight
##     <dbl> <chr>             <dbl>
## 1       1 Chaetodipus       23.5
## 2       1 Dipodomys         47.0
## 3       1 Neotoma          179.
## 4       1 Onychomys         24.5
## 5       1 Perognathus        7.38
## 6       1 Peromyscus        21.4
```

This yields `surveys_gw` where the observations for each plot are spread across multiple rows, 164 observations of 3 variables.

Using `pivot_wider()` to pivot on `genus` with values from `mean_weight` this becomes 24 observations of 9 variables, one row for each plot. We again use pipes:

```
surveys_wide <- surveys_gw %>%
  pivot_wider(names_from = genus, values_from = mean_weight)

glimpse(surveys_wide)
```

```
## Rows: 24
## Columns: 9
## Groups: plot_id [24]
## $ plot_id         <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...
## $ Chaetodipus     <dbl> 23.48263, 26.63729, 28.27273, 24.04444, 18.71429, 2...
## $ Dipodomys       <dbl> 46.95738, 46.38756, 48.60748, 45.64417, 47.97059, 4...
## $ Neotoma         <dbl> 178.7500, 169.1475, 171.0000, NA, 147.0000, 191.500...
## $ Onychomys       <dbl> 24.48214, 25.45238, 24.81159, 24.43478, 25.42308, 2...
## $ Perognathus     <dbl> 7.384615, 8.000000, 7.875000, 8.457143, 8.809524, 7...
## $ Peromyscus      <dbl> 21.42857, 22.53571, 21.00000, 22.60000, 20.52174, 2...
## $ Reithrodontomys <dbl> 14.00000, 11.36364, 12.28571, 10.00000, 11.46154, 1...
## $ Sigmodon        <dbl> NA, 69.0, NA, 82.0, NA, 73.0, NA, NA, 77.0, NA, NA,...
```

```
surveys_wide %>%
  head()
```

```
## # A tibble: 6 x 9
## # Groups:   plot_id [6]
##   plot_id Chaetodipus Dipodomys Neotoma Onychomys Perognathus Peromyscus
##     <dbl>       <dbl>     <dbl>   <dbl>     <dbl>       <dbl>      <dbl>
## 1       1        23.5      47.0    179.      24.5        7.38       21.4
## 2       2        26.6      46.4    169.      25.5        8          22.5
## 3       3        28.3      48.6    171       24.8        7.88       21
## 4       4        24.0      45.6     NA       24.4        8.46       22.6
## 5       5        18.7      48.0    147       25.4        8.81       20.5
## 6       6        27.2      46      192.      24.8        7.74       21.9
## # ... with 2 more variables: Reithrodontomys <dbl>, Sigmodon <dbl>
```

Challenge 9: 

Pivot the `combined` data frame to a wide format, with `year` as columns, `plot_id` as rows, and the number of genera per plot as the values. You will need to summarize before reshaping, and use the function `n_distinct()` to get the number of unique genera within a particular chunk of data. It's a powerful function! See `?n_distinct` or go to https://dplyr.tidyverse.org/reference/n_distinct.html for more information.

Save the wide dataset as an object, with an intuitive name! Then use `glimpse` to take a look at the structure.

```
## Make a wide data frame by pivoting on year.
## Fill the values in these columns with the number of genera per plot.
## Make sure to save the new dataset with an intuitive name!
```

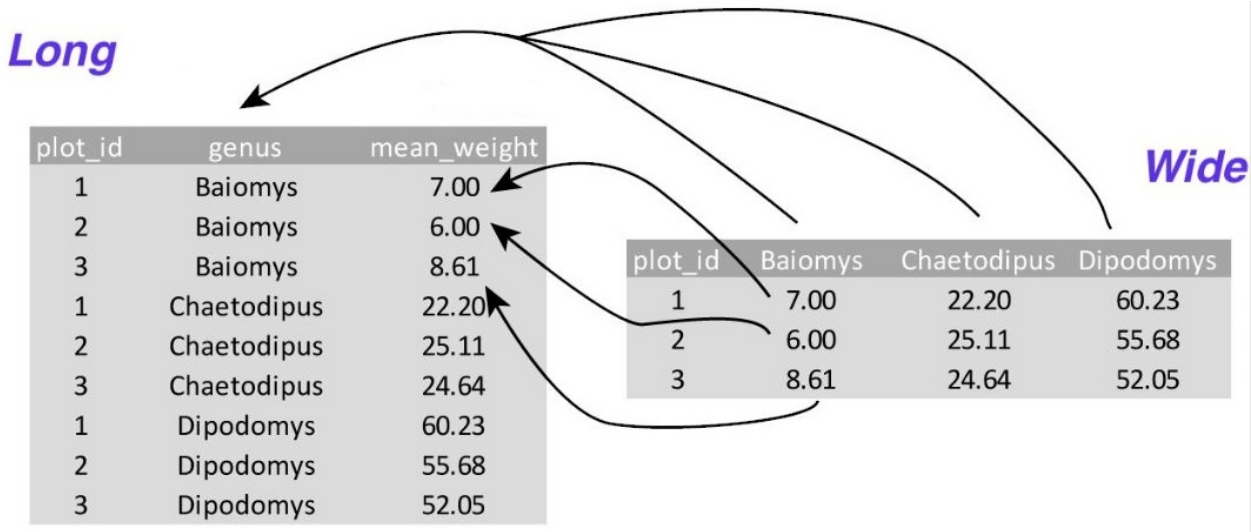
Pivoting to a Longer Table

The opposing situation could occur if we had been provided with data in the form of `surveys_wide`, where the genus names are column names, but we wish to treat them as values of a genus variable instead. This task is extremely common in longitudinal data where the columns are the measurement events over time on the same variable and the rows are for the locations or subjects and we want to align all the responses in one long vector for plotting (e.g., `ggplot`) or analyses.

In this situation we are gathering the column names and turning them into a pair of new variables. One variable represents the column names as values, and the other variable contains the values previously associated with the column names.

`pivot_longer()` takes four principal arguments:

1. the data
2. the columns we wish to pivot into a single column
3. the name of the new column to create to store the names of each selected column
4. the name of the new column to create to store the data filled in each cell



To recreate `surveys_gw` from `surveys_wide` we would create a key called `genus` and value called `mean_weight` and use all columns except `plot_id` for the key variable. Here we drop the `plot_id` column with a minus sign.

```
surveys_long <- surveys_wide %>%
  pivot_longer(cols = -plot_id, names_to = "genus", values_to = "mean_weight")

glimpse(surveys_long)
```

```
## Rows: 192
## Columns: 3
## Groups: plot_id [24]
## $ plot_id     <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3...
## $ genus       <chr> "Chaetodipus", "Dipodomys", "Neotoma", "Onychomys", "Pe...
## $ mean_weight <dbl> 23.482625, 46.957377, 178.750000, 24.482143, 7.384615, ...
```

Note that now the `NA` genera are included in the re-gathered format. Pivoting your data to a wide format and then pivoting to a long format can be a useful way to balance out a dataset so every replicate has the same composition and you can see where you could have obtained observations.

We could also have used a specification for what columns to include. This can be useful if you have a large number of identifying columns, and it's easier to specify what to gather than what to leave alone. And if the columns are in a row, we don't even need to list them all out - just use the `:` operator!

```
surveys_wide %>%
  pivot_longer(cols = Chaetodipus:Sigmodon, names_to = "genus",
               values_to = "mean_weight") %>%
  head()
```

```
## # A tibble: 6 x 3
## # Groups:   plot_id [1]
##   plot_id genus       mean_weight
##     <dbl> <chr>             <dbl>
## 1       1 Chaetodipus       23.5
## 2       1 Dipodomys         47.0
## 3       1 Neotoma          179.
## 4       1 Onychomys         24.5
## 5       1 Perognathus        7.38
## 6       1 Peromyscus        21.4
```

Challenge 10

Take the `surveys_wide_genera` dataset and use `pivot_longer()` to pivot it to the long format it was in before, so that each row is a unique `plot_id` by `year` combination.

HINT: The year column names look like numbers so you need to use back ticks (“‘”) to indicate they are variables instead of numbers.

```
## Now take the surveys_wide_genera dataset, and make it long again, by
## (re)pivoting on the year columns.

names(surveys_wide_genera)
```

```
## [1] "plot_id" "1996"    "1997"    "1998"    "1999"    "2000"    "2001"
## [8] "2002"
```

Challenge 11 Part 1: 

The `combined` data set has two measurement columns: `hindfoot_length` and `weight`. This makes it difficult to do things like look at the relationship between mean values of each measurement per year in different plot types.

Let's walk through a common solution for this type of problem.

First, use `pivot_longer()` to create a dataset called `combined_longer` where we have a names column called `measurement` and a values column that takes on the value of either `hindfoot_length` or `weight`.

HINT: You'll need to specify which columns to pivot into longer format!

```
## Use pivot_long() to create an even longer dataset.
## Create a column called measurement, containing the hindfoot and weight columns
## And a value column that takes on the value of either of these measurements
## Hint: You'll need to specify which columns are being used to pivot!
```

Challenge 11 Part 2: 

With this new data set, `combined_longer`, calculate the average of each `measurement` in each `year` for each different `plot_type`.

Then pivot these summaries into a data set with a column for `hindfoot_length` and `weight`.

HINT: This sounds like you want to pivot the data to be a wider format!

```
## With this new very long data set, calculate the average of each
## measurement in each year for each different plot_type.



## Now pivot these summaries into a wide data set.
## With a columns for hindfoot_length and weight.
## Filled with the summary values you calculated.
```

Challenge 11 Bonus: 

If you attended the Data Visualization workshop, make a plot of average hindfoot_lengths and weights with colors for the points based on the plot_type.

Exporting Data

Now that you have learned how to use `dplyr` to extract information from or summarize your raw data, you may want to export these new data sets to share them with your collaborators or for archival.

Similar to the `read_csv()` function used for reading CSV files into `R`, there is a `write_csv()` function that generates CSV files from data frames.

Before using `write_csv()`, it is good to create a new folder, `data`, in our working directory that will store the generated datasets. It is best to avoid writing generated datasets in the same directory as our raw data as that may create confusion later about which data set was the source and which was the "wrangled" version. So it is good practice to keep them separate. The `data_raw` folder should only contain the raw, unaltered

data, and should be left alone to make sure we don't delete or modify it. In contrast, our script will generate the contents of the `data` directory, so even if the files it contains are deleted, we can always re-generate them. If you were working in RStudio, the following code would create a `data` directory inside the folder where your .Rmd exists.

```
if(!dir.exists("data")){dir.create("data")}
```

For future use, we might want to prepare a cleaned up version of the dataset that doesn't include any missing data.

Let's start by removing observations of animals for which `weight` and `hindfoot_length` are missing, or the `sex` has not been determined:

```
surveys_complete <- surveys_edited %>%
  filter(!is.na(weight),           # remove missing weight
         !is.na(hindfoot_length),  # remove missing hindfoot_length
         !is.na(sex))                # remove missing sex

glimpse(surveys_complete)
```

```
## Rows: 11,328
## Columns: 11
## $ record_id       <dbl> 23215, 23217, 23220, 23222, 23223, 23224, 23225, 23...
## $ month           <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day             <dbl> 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27,...
## $ year            <dbl> 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 199...
## $ plot_id         <dbl> 21, 17, 2, 1, 2, 17, 2, 1, 21, 18, 17, 22, 12, 2, 1...
## $ species_id      <chr> "PF", "DM", "DM", "DM", "DO", "DM", "DM", "PB", "PP...
## $ sex             <chr> "F", "M", "F", "M", "M", "F", "M", "F", "F", "F", "...
## $ hindfoot_length <dbl> 16, 36, 36, 34, 37, 39, 40, 27, 21, 16, 36, 36, 38,...
## $ weight          <dbl> 7, 25, 47, 27, 66, 49, 54, 38, 16, 9, 51, 43, 44, 4...
## $ date            <date> 1996-01-27, 1996-01-27, 1996-01-27, 1996-01-27, 19...
## $ day_of_week     <fct> Saturday, Saturday, Saturday, Saturday, Saturday, S...
```

If we were interested in plotting how species abundances have changed through time, we might also want to remove observations for rare species (i.e., that have been observed less than 50 times). We will do this in two steps: first we are going to create a data set that counts how often each species has been observed, and filter out the rare species; then, we will extract only the observations for these more common species:

```
## Extract the most common species_id
species_counts <- surveys_complete %>%
    count(species_id) %>%
    filter(n >= 50)

## Only keep the most common species
surveys_complete_subset <- surveys_complete %>%
  filter(species_id %in% species_counts$species_id)
  ## using the relational operator %in%

glimpse(surveys_complete_subset)
```

```
## Rows: 11,266
## Columns: 11
## $ record_id       <dbl> 23215, 23217, 23220, 23222, 23223, 23224, 23225, 23...
## $ month           <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ day             <dbl> 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27,...
## $ year            <dbl> 1996, 1996, 1996, 1996, 1996, 1996, 1996, 1996, 199...
```

```
## $ plot_id         <dbl> 21, 17, 2, 1, 2, 17, 2, 1, 21, 18, 17, 22, 12, 2, 1...
## $ species_id      <chr> "PF", "DM", "DM", "DM", "DO", "DM", "DM", "PB", "PP...
## $ sex             <chr> "F", "M", "F", "M", "M", "F", "M", "F", "F", "F", "...
## $ hindfoot_length <dbl> 16, 36, 36, 34, 37, 39, 40, 27, 21, 16, 36, 36, 38,...
## $ weight          <dbl> 7, 25, 47, 27, 66, 49, 54, 38, 16, 9, 51, 43, 44, 4...
## $ date            <date> 1996-01-27, 1996-01-27, 1996-01-27, 1996-01-27, 19...
## $ day_of_week     <fct> Saturday, Saturday, Saturday, Saturday, Saturday, S...
```

We can check that `surveys_complete_subset` has 11266 rows and 11 columns by typing `dim(surveys_complete_subset)` in the previous sandbox.

Now that our data set is ready, we can save it as a CSV file in our `data` folder.

```
write_csv(surveys_complete_subset, path = "data/surveys_complete_subset.csv")
```

Happy wrangling!

Other Workshops in the Series

The goal of this workshop was to teach you to write code in `R` to perform data wrangling in a reproducible fashion. Recordings of the previous workshops are available at http://www.montana.edu/datascience/training/. We plan to offer this same series in the spring (dates to be determined) and so these web-interface "Shiny" apps will continue to evolve and links might change. If they are not available when you try to revisit them, please contact one of the authors and we can point you to the current versions of them.

The first workshop in our series contains more information on how to get started working in `R` using RStudio. The second workshop contains information on how to write code to visualize data using `ggplot2`. The third workshop, Intermediate R, explored more sophisticated R code involving logicals, loops, and functions. The codechunks in this interactive document mimic the codechunks you can use on your own projects in RMarkdown but you will need to download and install both `R` and RStudio on your own computer.

Montana State University `R` Workshops Team

These materials were adapted from materials generated by the Data Carpentries (https://datacarpentry.org/) and were originally developed at MSU by Dr. Allison Theobold. The workshop series is co-organized by the Montana State University Library, Department of Mathematical Sciences, and Statistical Consulting and Research Services (SCRS, https://www.montana.edu/statisticalconsulting/). SCRS is supported by Montana INBRE (National Institutes of Health, Institute of General Medical Sciences Grant Number P20GM103474). The workshops for 2020-2021 are supported by Faculty Excellence Grants from MSU's Center for Faculty Excellence.

Research related to the development of these workshops is to appear in:

- Theobold, A., Hancock, S., & Mannheimer, S.. Data Science Workshops for Data-Intensive Environmental Science Research, *Journal of Statistics Education*.

The workshops for 2020-2021 involve modifications of materials and are being taught by:

Sara Mannheimer

- Sara Mannheimer is an Associate Professor and Data Librarian at Montana State University Library (http://www.lib.montana.edu/people/about/139), where she helps shape practices and theories for curation, publication, and preservation of data (https://www.lib.montana.edu/services/data/). Her

research is rooted in the examination of the social, ethical, and technical issues that arise as we grapple with the implications of a data-driven world.

Greta Linse

- Greta Linse is the Assistant Director of Statistical Consulting and Research Services (https://www.montana.edu/statisticalconsulting/) and the Project Manager for the Human Ecology Learning and Problem Solving (HELPS) Lab (https://helpslab.montana.edu). Greta has been teaching, documenting and working with statistical software including R and RStudio for over 10 years.

Mark Greenwood

- Mark Greenwood is a Professor of Statistics in the Department of Mathematical Sciences at Montana State University and Director of Statistical Consulting and Research Services (https://www.montana.edu/statisticalconsulting/). His research interests have involved statistical methods and applications in environmental sciences, education, and biological sciences. Recent work has involved researching diagnostic methods for Multiple Sclerosis. His current research grants include funding from the Mountain West IDeA Clinical and Translational Research - Infrastructure Network (National Institute of General Medical Sciences Grant 5U54GM104944-07) and a grant from the National Multiple Sclerosis Society (RG-1907-34348); SCRS is supported by Montana INBRE (National Institutes of Health, Institute of General Medical Sciences, Grant P20GM103474).