Data and Donuts: Data Wrangling in R
Based on the data carpentry ecology lessons:
http://www.datacarpentry.org/R-ecology-lesson/03-dplyr.html

Slide 1: Hi, and welcome to Coding and Cookies. I'm Tobin Magle, the cyberinfrastructure facilitator at the Morgan Library. Today we're going to be covering data wrangling in R based on the Data Carpentry curriculum.

Slide 2: In brief, we'll

1. Learn a new way to import data using the read_csv() funcion
2. Demonstrate the 6 dplyr verbs for data manipulation
3. Combine these verbs with an operator called the pipes
4. Use 2 tidyr verbs to reshape your data
5. Create a clean dataset to export to a file

Slide 3: for these exercises, we're assuming that you have a basic working knowledge of R and R studio. You'll need to

- Install both R and R Studio. See the setup instructions from Data Carpentry Linked on this slide if you need help.
- Download and unzip the quickstart files from the bitly link on the slide. This file provides a premade working directory and file structure for this lesson.
- If you want to know how to set up a directory for yourself or are unfamiliar with R and R studio, see the Basic data analysis in R lesson linked on this slide.

Slide 4: In this lesson, we are going to move beyond the R base installation and install packages from the **tidyverse**, which is A set of packages that provides easy tools for data manipulation.

- All the tidyverse packages are built around tidy data tables
- You can do all the things that the tidyverse in base R but these new functions make data manipulation easier
- These functions are designed to work with the pipe operator from the magrittr package, which allows you to make the output of one verb the input of another verb. This feature makes your code easier to edit and read.

Slide 5:

- Before you can use any of these features, you need to install the package using the install.packages function. You only need to do this once on each R installation
- Then you need to load the package using the library function. You need to do this every time you start RStudio to use the functions it contains.

Let's load the data and the packages before we learn more about dplyr.

Demo 1: Setting up

- Open the R project
- Point out the file structure:
    - Rproj file – save your place while you're working

    - complete R script – follow along if you don't like typing. But I recommend opening a new script and typing
    - Data folder with data file
- Open a new script file
- Load the tidyverse package

- Read the data into a new variable called surveys

Slide 6: Now let's load the data into a variable called surveys:

- We're going to use read_csv instead of read.csv
- Just like read.csv, it takes a file path as input
- However, instead of outputting a standard data frame, it's output is a data structure called a tibble.
    - When you print a tibble, it includes data type under column name
    - Also, it does not convert characters into factors by default

Demo 2: Loading data

- surveys<-read_csv("data/portal_data_joined.csv")
- surveys – look at output
- str(surveys)

Slide 7: Now that we have the data loaded, let's manipulate it with our first verb: select

- Select picks columns from a data frame
- It takes a tibble and a list of column names as input
- And its output is a tibble with only the columns you selected
- Let's see an example of how select works

Demo 3: select

- Let's say we have a collaborator who only wants the plot, species id and weight data
- **select**(surveys, plot_id, species_id, weight)

Slide 8: We've seen how to choose particular columns using select, now let's look at how to pick rows using filter

- Filter chooses rows based on specified criteria
- It takes a tibble and relational expression as arguments
- And it outputs a tibble with only the rows that meet the relational expression
- For example, you can specify that you only want rows where the year is equal to 1995
- Let's see select in action

Demo 3: filter

- Let's say we only want to look at records taken in 1995
- **filter**(surveys, year **==** 1995)

Slide 9: At this point, you might be thinking that you can do these things in R without using dplyr

- However, dplyr provides a convenient way to string verbs together
- using the pipe operator (%>%), (percent sign)-(greater than)-(percent sign)
- The operator goes at the end of each line that you want to string together.
- Then the output of the previous line becomes the data frame input for the next line
- This means that you don't have to explicitly provide the data frame argument in each verb function
- For example we could specify that we only want records that have weights of less than 5 g and only the species_id, sex and weight columns in one statement.
- Let's see how pipes work in a demo

Demo 4: pipes

- We can use the assignment operator to save the output in a data frame called surveys_sml

```
surveys_sml<-surveys %>%
                filter(weight<5) %>%
                select(species_id, sex, weight)
```

Slide 10: Now that you know about select, filters and pipes, let's do an exercise involving these commands

Exercise 1:

Using pipes, subset the survey data to include individuals collected before 1995 and retain only the columns year, sex, and weight.

Solution 1:

- Start with the surveys data frame, followed by the pipe operator
- Filter on year ==1995
- Select year, sex, and weight
- Could we reverse select and filter?
    - Yes, but only in cases where the select statement contains the variable being evaluated in the filter statement

Slide 11: dplyr also allows you to create new columns using the mutate function

- Which creates a new column as defined by the input
- Mutate takes a tibble and an expression that names and defines the value of the new column
- This function outputs a tibble that includes the new column as defined in the input
- For example, the weight is currently in grams. We could create a new column called weight_kg that stores the weight in kilograms
- Let's see how this works in practice

Demo 5:

```
mutate(surveys, weight_kg = weight/1000)
same as surveys %>%
                mutate(weight_kg = weight/1000)
surveys %>%
          mutate(weight_kg = weight / 1000,
                    weight_kg2 = weight_kg *2)

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

Slide 12: Here's a slide with the syntax we went over above

Slide 13: Another useful function for data cleaning is is.na()

- Which takes a column as input
- And returns a T/F vector of the same length that has the value true where there is a missing value and false where the input vector has a value
- This T/F vector can be used as input to a filter statement and the not operator (!) to remove

Demo 6:

Show the T/F vector

```
!is.na(weight)
```

Use it as input to filter -> all weight values are NA

```
surveys %>%
      filter(is.na(weight))
```

Add the not operator -> no weight values are NA

```
surveys %>%
      filter(!is.na(weight))
```

Can be strung together with other functions

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

Slide 14: Let's do another exercise combining the 3 verbs with pipes

Exercise 2:

Create a new data frame from the survey data that meets the following criteria:

1. contains only the species_id column and a new column called hindfoot_half
2. hindfood_half contains values that are half the hindfoot_length values.

3. Only include records from 1990 and after

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

Solution 2:

- Start with the surveys data frame
- First, let's filter the rows: filter(year>=1990)
- Then create hindfoot half: mutate(hindfoot_half = hindfoot_length/2)
- Finally, select the columns:
- Why did I do this in this order?
    - Have to do select last because the other 2 depend on year and hindfoot length, which aren't included in the output of the select statement
    - Mutate and filter could be switched, but reducing the number of rows first makes the computation a bit more efficient.

Slide 13: Next, we're going to look at the group and summarize by functions.

- Group_by groups data in the table by an attribute
    - It takes a tibble and a column with a categorical variable to group by
    - and outputs a tibble that looks a lot like the original, but indicates which rows are part of each attribute
- summarize applies a summary statistic to grouped data
    - It takes a grouped tibble and a definition of a summary statistic as input
    - And outputs another tibble with the groups as rows, and the summary stats as columns

Let's look at how this works

Demo 6:

```
#Overall mean weight
surveys %>%
        summarize(mean_weight = mean(weight))

#remove NAs
surveys %>%
        summarize(mean_weight = mean(weight, na.rm = TRUE))

#Mean weight by sex
surveys %>%
        group_by(sex) %>%
        summarize(mean_weight = mean(weight, na.rm = TRUE))
```

Slide 16: You can also group by multiple attributes, for example, sex and species id

- To do this, add multiple columns as input to group_by
- The output will then include a column for each attribute and the summary statistic

Let's see how this works.

Demo 8:

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

Slide 17: Let's look at a couple of other things you can do. Instead of using the na.rm argument to summarize, you can filter out the NAs ahead of time.

Demo 9:

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

Slide 18: You can also choose how many lines to print with the n argument to the print function

Demo 10:

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarize(mean_weight = mean(weight)) %>%
  print(n = 15)
```

Slide 19: And finally, you can calculate multiple summary statistics

Demo 11:

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

Slide 20: Let's look at another function that works well with group by: tally

- Tally counts the number of observations in a group
- Grouped tibble as input
- And returns a tibble with a column for each grouped variable, and one for the count of each row in that category

Let's see how this works

Demo 12: Tally

```
surveys %>%
```

```
group_by(sex) %>%
tally
```

Slide 21: Let's pull this all together in an exercise

Exercise 3:

- How many individuals were caught in each plot_type surveyed?
    - Start with surveys
    - Group by plot type
    - tally

- Use group_by() and summarize() to find the mean, min, and max hindfoot length for each species (using species_id).
    - Start with surveys
    - Filter(!is.na(weight))
    - Group by species id
    - Summarize
        - Mean_hf = mean(hindfood_length)
        - Min_hf = mean(hindfood_length)
        - Min_hf = mean(hindfood_length)

- What was the heaviest animal measured in each year? Return the columns year, genus, species_id, and weight.
    - Start with surveys
    - Filter(!is.na(weight))
    - Group by year
    - Filter(weight == max(weight)) %>%
    - Select year, genus, species_id, weight
    - Arrange(year)

Slide 22: Now let's talk about how to reshape data with tidyr

- Reshaping data is important because what you can do with your data depends on how its formatted
- For example, if you want to make a table that shows mean weight by plot, you need to reshape the sata so that the rows are plot
- To do this, we're going to use two tidyr verbs:
    - Spread, which makes your table wider and
    - Gather, which makes your table longer

Slide 23: Let's look at the spread function

- Spread makes the table wide by turning values in cells into column headers
- Spread takes a tibble, a key column, and a value column as input
    - The key column is the column whose values you want to be column headers
    - And the value column is the column that holds the values to fill out the output table

- The output is a tibble with the key column converted to column headers and the value column filling out the table.

Slide 24: Let's see how this works.
Demo 13:

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

surveys_spread <- surveys_gw %>%
                       spread(key = genus, value = mean_weight)

surveys_gw %>%
             spread(genus, mean_weight, fill = 0) %>%
             head()
```

Slide 25: now let's talk about gather

- Gather makes a long table by converting column headers to values in a table
- It takes a tibble, a key column, and a value column, and a specification of the columns to gather as input
    - The key column is the column that you want to create from column names
    - The value column is the column you want to put table values into

Slide 26: Let's see how this works

Demo 14:

```
surveys_spread %>%
                  gather(key = genus,
                           value = mean_weight,
                           Baiomys:Spermophilus) %>%
                  head()

surveys_spread %>%
                   gather(key = genus,
                            value = mean_weight,
                            Baiomys:Spermophilus) %>%
                   head()
```

Slide 27: Let's look at an exercise doing gather and spread
Exercise 4
Goal: look at the relationship between mean values of weight and hindfoot length per year in different plot types.
Step 1: Use gather() to create a dataset where we have a key column called measurement and a value column that takes on the value of either hindfoot_length or weight.

Step 2: Calculate the average of each measurement in each year for each different plot_type.
Step 3: spread() them into a data set with a column for hindfoot_length and weight.

Solution:
Step 1:

```
long_data<-surveys%>%
      gather(key = measurement, #new col from col headers
            value = value,     #values
            hindfoot_length, weight) #columns to gather
```

Step 2:

```
mean_values<-long_data %>%
      filter(!is.na(value))%>%
      group_by(measurement, plot_type, year)%>%
      summarise(mean = mean(value))
```

Step 3:

```
mean_values%>%
      spread(key = measurement,
            value = mean)
```

Slide 28: Since the next data and donuts session is about graphing with ggplot, we want to get ready by cleaning up this dataset. First we're going to remove all of the rows with missing values.

Demo 10: remove missing values

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

Slide 29: We can also eliminate rare species from the dataset.

Demo 11: eliminate rare species

```
## Extract the most common species_id
species_counts <- surveys_complete %>%
  group_by(species_id) %>%
  tally %>%
  filter(n >= 50)
## Only keep the most common species
surveys_complete <- surveys_complete %>%
  filter(species_id %in% species_counts$species_id)
```

Slide 30: Now that we have a clean dataset, we're going to write surveys complete to a file using the write_csv function

- write.csv takes a data frame, the name of an output file at minimum as arguments
- you can also specify other parameters, like whether or not to include row names in the file
- The content of a data frame is then output to the specified file

Demo 12: write data

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

Slide 31: Thanks for listening. As always, email me at the address on the slide if you need help with these or any other data management topics. See out web site for a list of the topics I can help with. Additionally, see the data carpentry lessons for the full source material for this lesson. Finally, the data wrangling cheat sheet is a good resource for dplyr as you're coding.