Data Wrangling - Manipulating Data with dplyr & tidyr

Allison Theobold

Learning Objectives

- Understand the purpose of the `dplyr`, `stringr`, and `tidyr` packages.
- Use `dplyr` to:
  - select columns in a data frame
  - select rows in a data frame according to filtering conditions
  - add new columns to a data frame which are functions of existing columns
- Link `dplyr` statements together using the 'pipe' operator `%>%`.
- Use `summarise` and `group_by` to split a data frame into groups of observations, apply a summary statistic for each group, and then combine the results.
- Understand how datasets can be joined together and how to use `dplyr` to join data frames together
- Understand what wide and long data formats are for which purpose each format is useful.
- Use `tidyr` to reshape a data frame from long to wide format and back, using the `spread` and `gather` functions.

Data for Workshop

The data used in this workshop is a time-series for a small mammal community in southern Arizona. These data are part of a project studying the effects of rodents and ants on the plant community that has been running for almost 40 years. 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 a real dataset that has been used in over 100 publications.

These data provide a real world example of life-history, population, and ecological data, with sufficient complexity to focus on many aspects of data analysis and management. The data have been simplified just a bit for the workshop, but you can download the full dataset and work with it using exactly the same tools we'll learn about today. Data are available through Portal Project Teaching Database.

Load the data into your work space and inspect it to answer the following questions:

- How many observations (rows) do the data have?
- How many variables (columns) do the data have?
- What class are the variables in the data frame (e.g. numeric, double, character, factor)?

```
# Load in the surveys data (surveys.csv)
surveys <- read.csv("data/surveys.csv")

# Inspect the data here!
```

Data Manipulation using `dplyr`

It great to have the knowledge to subset your data using bracketing, but too many manipulations can get difficult to read. Enter **`dplyr`**. **`dplyr`** is a package which makes data manipulation simpler and more concise.

`dplyr` is one of the many packages available in `R`, developed and maintained by Hadley Wickham. Some of the functions that you have seen in previous workshops, or are potentially familiar with (e.g. `str`, `summary`) come built into `R`. In order to use the elements of the **`dplyr`** package, you have to first install the package (only once) and then load the package (every time). The **`dplyr`**, **`tidyr`**, and **`stringr`** packages are included in the **`tidyverse`** package. This is an "umbrella-package" that installs several packages useful for data analysis which work together well, such as **`purrr`**, **`readr`**, **`tibble`**, and **`ggplot2`**.

First download the **`dplyr`** package to your work space, using the Packages tab at the bottom-right corner. Once the package is downloaded, load the **`tidyverse`** package by typing:

```
library(tidyverse)    ## load the tidyverse packages, incl. dplyr
```

Notice that when loading in the **`tidyverse`** package it tells you which packages were added to your work space (**`ggplot`**, **`purrr`**, **`tibble`**, **`dplyr`**, **`tidyr`**, **`stringr`**, **`readr`**, **`forcats`**). The loading also tells you what functions have been masked (overwritten) from the loading of **`tidyverse`** (`filter` and `lag`). Lastly, it will tell you what version of `R` the package was built under. Running `R.version` will give you the version of `R` that you are currently working in, and will give you a good idea if you should update `R`!

What is `dplyr`?

The package **`dplyr`** houses simple to use tools for the most common data manipulation tasks. **`dplyr`** is built to work directly with data frames, and is written in C++ to enhance the speed of the operations.

Six of the key **`dplyr`** functions are:

- `filter`: picks observations (rows) based on value conditions
- `select`: picks variables (columns) based on names
- `mutate`: creates new variables based on existing variables
- `arrange`: reorders the rows of a data frame
- `group_by`: changes the scope of a function from the entire data frame to operating group-by-group
- `summarise`: collapses many values down to a single summary statistic (e.g. count, mean, median)

These six functions provide the verbs that you need for the language of data manipulation. Each of these verbs works similarly:

- The first argument is a data frame.
- The subsequent arguments (second, third,...) describe what to do with the data frame. These arguments use variable names **not** in quotations.
- The result of these operations is a new data frame.

`select` and `filter`

Two of the most common **`dplyr`** functions are `select()` and `filter()`. The `select()` function selects columns of a data frame. The first argument to this function is the data frame, and the subsequent arguments are the columns to keep (or omit).

```
# include named columns
select(surveys, month, day, year, plot_id)

# exclude named columns (selects everything but these columns)
select(surveys, -month, -day, -year)

# select a range of columns
select(surveys, month:sex)
```

Notice that these statements output a data frame in the console, but do not store this resulting data frame into a new `R` object.

Exercise 1: Subset the `surveys` data to include all columns but the dates AND store the resulting data frame as a new `R` object named `sml_survey`.

```
# Your select statement here!
```

The `filter()` function chooses rows based on a selection criteria (relational statement). Again, the first argument to this function is the data frame, and the subsequent arguments are the filters that you wish to apply to the rows of the data frame.

```
# subsetting data frame to only have rodents captured in 1989
filter(surveys, year == 1989)

# subsetting data frame to only have plot IDs of 10 or less
filter(surveys, plot_id <= 10)
```

Exercise 2: Subset the `surveys` data to include only female rodents who were captured in plots 12 or above AND store the resulting data frame as a new `R` object named `female`.

```
# Your select statement here!
```

Pipes

What if you want to select and filter at the same time? Well, there are three ways to do this, (1) intermediate steps, (2) nested functions, and (3) pipes.

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

```
surveys2 <- filter(surveys, species_id == "OL")
surveys_sml <- select(surveys2, month, day, year)
```

You can see that this method will quickly clutter your work space with temporary data frames that you do not need.

Nesting functions (i.e. a function inside of another function) is another method, and looks like this:

```
surveys_sml <- select(filter(surveys, species_id == "OL"), month, day, year)
```

Using this method, the output of the inside function (data frame) is then used as the input in the outside function. This is a handy trick, but becomes very difficult to read for more than two nestings.

The final (and best) option is using the piping operator (`%>%`), a fairly recent addition to `R`. Pipes perform the same function as nested functions, taking the output of one function and using it as the input in the next function. However, pipes help to make this process much more concise and help to keep your code read-able. Pipes are made available through the `magrittr` package, which is automatically installed 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.

```
ol <- surveys %>%
        filter(species_id == "OL") %>%
        select(month, day, year)
```

In the code above, we are using the pipe to send the `surveys` data frame into the `filter()` function first. The result of this is another data frame, which is then piped into the `select()` function to keep only the month, day, and year columns. Because of the pipe, we do not need to include the data frame argument into the `filter()`, `select()`, or `rename()` functions any more.

You may find it useful to think of the pipe as the word "then". For example, in the above code we take the `surveys` data frame, *then* filter it so that it contains only the OL species ID and *then* we keep only the date variables. These functions are relatively simple on their own, but joined together into a linear workflow with the pipe allows for us to accomplish more complex data manipulation tasks.

Exercise 3: Using pipes, subset the `surveys` data to include only the rodents recorded in the month of July and retain all of the columns except the date variables.

```
# piping code here!
```

mutate

Quite often we want to create new columns in our data frame based off of values of existing columns, such as unit conversions, log transformations, or ratios of two columns. For this task we will use the `mutate()` function.

To create a new column of rodent weights in kilograms:

```
# creating a new column called weight_kg
surveys %>%
  mutate(weight_kg = weight/1000))
```

You can also create a second column based on the first column you created:

```
# creating a new column called weight_kg, hindfoot_cm, and condition
surveys %>%
  mutate(weight_kg = weight/1000) %>%
  mutate(hindfoot_cm = hindfoot_length/1000) %>%
  mutate(condition = weight_kg/hindfoot_cm)

# could put all variables you are creating into a single mutate() command,
# separated by commas
```

Again, this output is printing the resulting data frame to the console. If you wish to only see the first few rows of the resulting dataset, you can use the pipe to view the `head()` of the data frame. The pipe works with non-`dplyr` functions too, as long as the `magrittr` package is loaded!

```
# creating a new column called weight_kg, hindfoot_cm, and condition
surveys %>%
  mutate(weight_kg = weight/1000,
         hindfoot_cm = hindfoot_length/1000,
         condition = weight_kg/hindfoot_length) %>%
  head()
```

Exercise 4: Create a new data frame from the surveys data that meets the following criteria: contains only the `species_id` column and a new column called `hindfoot_half`, containing half of the value of the `hindfoot_cm` values. In this `hindfoot_half` column, there should be no NAs and all values are less than 20.

```
# Exercise 4 code here!
```

Split-Apply Combine

We can often decompose data analysis tasks into the *split-apply-combine* paradigm: split the data into groups, apply an analysis to each group, and combine the results. This process is made very simple with the use of the `group_by()` and `summarise()` functions.

`group_by()` and `summarise()`

The `group_by()` function is frequently used in conjunction with the `summarise()` function. To split the data into groups we use the `group_by()` function. This function takes the column names of the **categorical** variable(s) for which you want to calculate summary statistics on. To apply an analysis to these groups

we then use the `sumarise()` function. This function collapses each group of a particular variable into a single-row summary of that group.

Note: Technically **dplyr** contains both the `sumarise()` and `sumarize()` function, so English or British spelling will work.

To view the mean weight by rodent sex:

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

You may have noticed that the output from your code no longer runs off the screen. This is because **dplyr** changed our `data.frame` object to an object of class `tbl_df`, also known as a "tibble". A tibble is a data structure similar to a data frame with only three main differences, (1) it prints the data type under each column, (2) it only prints the first few rows and only the columns that will fit on the screen, and (3) `character` columns are never converted into factors.

You could group by multiple columns:

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

You may notice that there are a great deal of `NaN` values in the `mean_weight` column for different species. This is because some rodents escaped before their sex and body weight could be determined. These are `NaN` values, which are different from `NA` values, as `NaN` stands for "Not a Number", and come from taking the mean of a variable with no numerical values (the only values are `NA`s). To avoid this we can remove the `NA` values for `weight` first.

```
# to remove the NA's first
surveys %>%
  filter(is.na(weight) != TRUE) %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight = mean(weight))
```

Once you've grouped the data, you can summarize across multiple variables at the same time (and not necessarily on the same variable). For example, we could add columns indicating the minimum and maximum hindfoot length for each species for each sex.

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight = mean(weight),
            min_hindfoot = min(hindfoot_length),
            max_hindfoot = max(hindfoot_length))
```

Sometimes we would like to rearrange the output of a query to inspect the values. If we wanted to sort the species by mean weight, we could arrange our previous query to sort `mean_weight` in ascending order.

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight = mean(weight),
            min_hindfoot = min(hindfoot_length),
            max_hindfoot = max(hindfoot_length)) %>%
  arrange(mean_weight)
```

To sort the mean weight in descending order instead, we need to use the `desc()` function inside the `arrange()` function.

```
surveys %>%
  filter(!is.na(weight)) %>%
  group_by(sex, species_id) %>%
  summarise(mean_weight = mean(weight),
            min_hindfoot = min(hindfoot_length),
            max_hindfoot = max(hindfoot_length)) %>%
  arrange(desc(mean_weight))
```

Exercise 5: Use `group_by()` and `summarise()` to find the mean and standard deviation of the `hindfoot_length` for every species.

```
# Exercise 5 code here!
```

Exercise 6: What was the lightest animal caught each year? Return a data frame with columns `year`, `species`, `sex`, `species_id`, and `weight`.

```
# Exercise 6 code here!
```

`count`

When working with categorical data, it is often useful to know how many observations were found for each factor or combination of factors. To accomplish this task, we will use **`dplyr`**'s `count()` function. For example, we may want to know the number of rodents captured in each plot.

```
surveys %>%
  count(plot_id)

# Equivalent to:
surveys %>%
  group_by(plot_id) %>%
  summarise(count = n())
```

Both of these statements accomplish the same task; however, the `count()` function also conveniently supplies a `sort` argument.

```
surveys %>%
  count(plot_id, sort = TRUE)
```

As we saw previously, we can group by multiple variables and obtain summaries for the intersection of these variables. We can also accomplish this with the `count()` function, obtaining counts for multiple categorical variables.

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

If we wanted to arrange this table by alphabetical order for species and descending order of count, we would add:

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

Exercise 7: How many rodents were caught each year?

```
# Exercise 7 code here!
```

Exercise 8: Adding to **Exercise 5**, also include the number of observations of each species.

```
# Exercise 8 code here!
```

Relational Data with `dplyr`

It is rare that ecological data analyses 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("data/plots.csv")

species <- read.csv("data/species.csv")
```

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

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?)

- plots key:
- species key:
- surveys key:

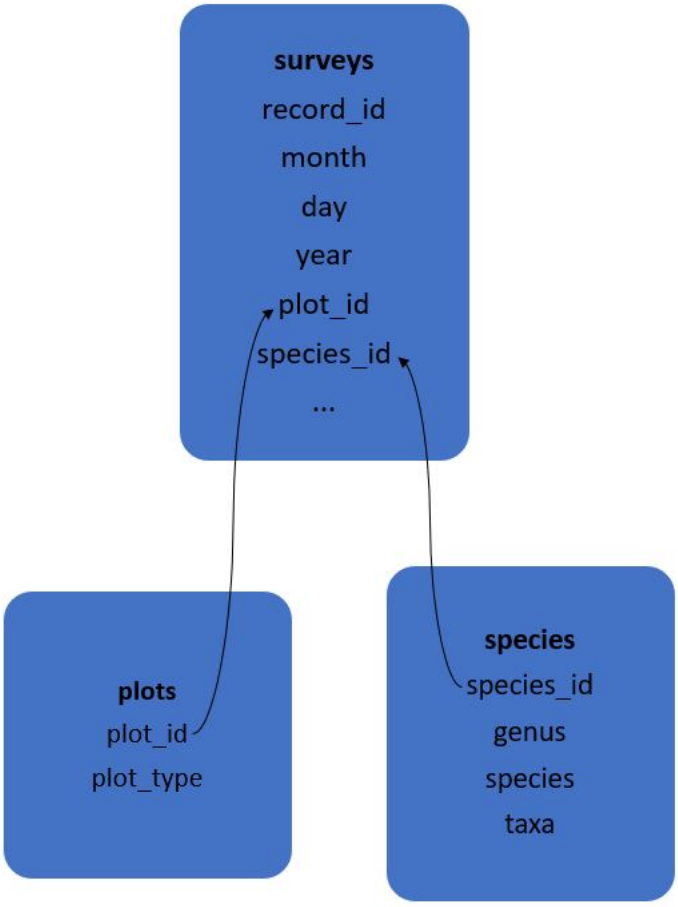


Figure 1: Relations of survey data tables

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.

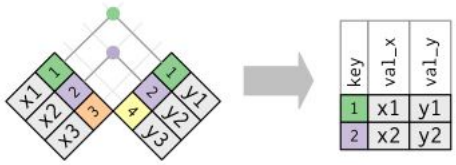
Joining Relational Data

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.

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

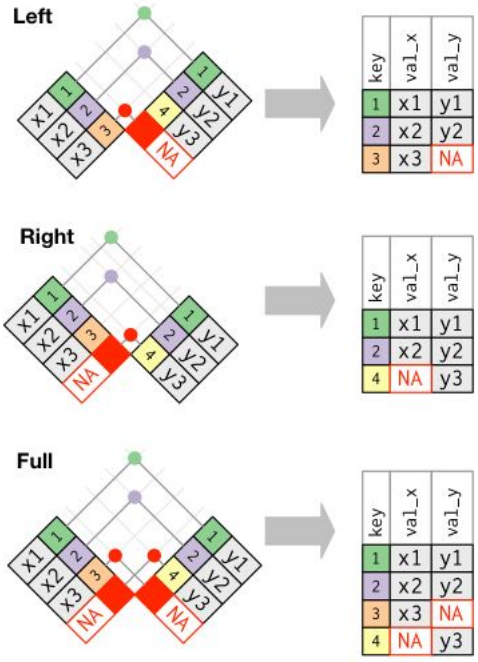


Outer Join

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:

- 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`.

Figure 3: Wickham, H. & 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 `survey` Data

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

```
combined <- surveys %>%
              left_join(plots, by = "plot_id") %>%  # adding the type of plot
              left_join(species, by = "species_id") # adding the genus, species, and taxa
```

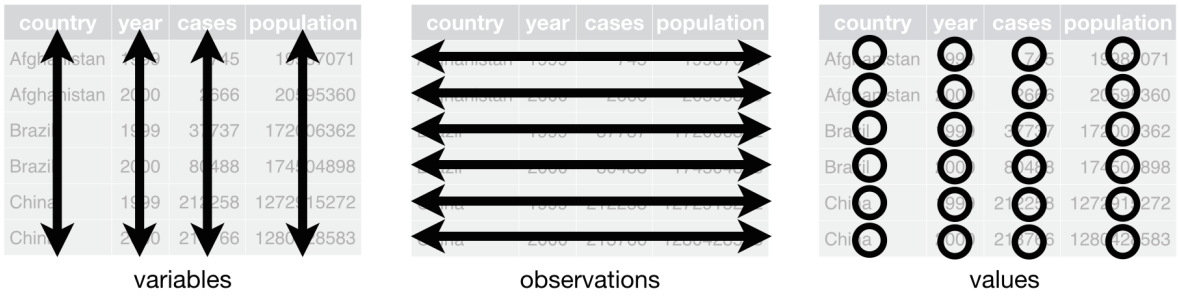
What is `tidyr`?

The **`tidyr`** package helps to address common problems you may encounter when tidying or reshaping your data. This package makes transitions between the various data formats necessary for plotting and analyzing take much less time. For example, the survey data that we have been working with is (almost) presented in what we would call a *long* format, since every observation of every rodent is its own row. This is a great format for many things in `R`, but it makes some relationships difficult to see. For example, how would we determine what is the relationship between mean weights of different species across every plot id?

To answer that question, we would need to have every plot id in its own row, and all of the measurements on that plot having their own column. This format is what we call a *wide* data format. For some purposes we prefer a long data format and for others we prefer a wide format. This sounds cumbersome, but moving from one of these formats to the other is much simpler with the help of the **`tidyr`** package!

Generally, we will consider data to be "tidy" if they follow three basic rules:
1. Each variable has its own column.
2. Each observation has its own row.
3. Each value has its own cell.

Figure 4: Wickham, H. & Grolemund, G. (2017) *R for Data Science*. Sebastopol, California: O'Reilly.



Why should your ensure that your data are tidy?

- Picking a consistent method for storing your data allows for you to better learn the tools for working with that specific structure of data.

- Placing each variable in its own column takes advantage of the structure of `R`, around vectors, to shine. Most of the built-in `R` functions you will encounter work with vectors of values.

Reshaping Long to Wide with `spread`

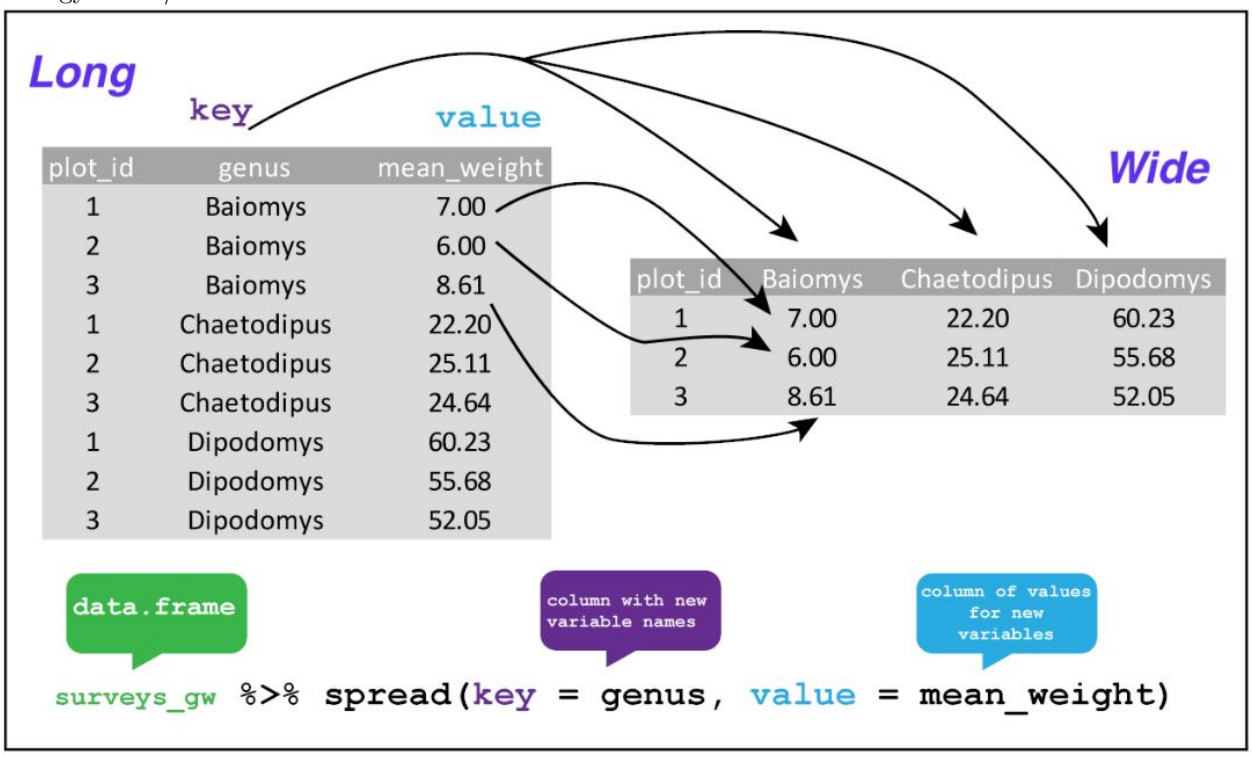
When an observation is spread across multiple rows, the `spread()` function comes in handy. For example, each `plot_id` has a mean weight for every species caught there. Take a look at these data:

```
combined %>%
  filter(!is.na(weight)) %>%
  group_by(genus, plot_id) %>%
  summarize(mean_weight = mean(weight)) %>%
  head(n = 10)
```

Each `plot_id` has multiple rows in our dataset, one for each genus. We can use the `spread()` function to create new columns for each `genus` name and fill them with the mean weight. The `spread()` function takes three main arguments:
1. the data to be reshaped
2. the *key* column variable whose values become the new columns
3. the *value* column whose values will fill the new columns

To make these long data into a wide format, we will use the `spread()` function from `tidyr` to spread the different genus into multiple columns.

Figure 5: Woo, K. & Hollister, J. (2019, March). Retrieved March 14, 2019, from http://datacarpentry.org/R-ecology-lesson/.




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

We notice that some genera have `NA` values. This is because those genera were never caught on that plot. If we wish to use these data to plot the mean weight of each genus across different plots, it may be helpful to turn the `NA`s into 0's. In this case we could add the `fill = 0` argument to our `spread()` function.

```
wide_survey <- combined %>%
                filter(!is.na(weight)) %>%
                group_by(genus, plot_id) %>%
                summarize(mean_weight = mean(weight)) %>%
                spread(genus, mean_weight, fill = 0)
```

Reshaping Wide to Long with `gather`

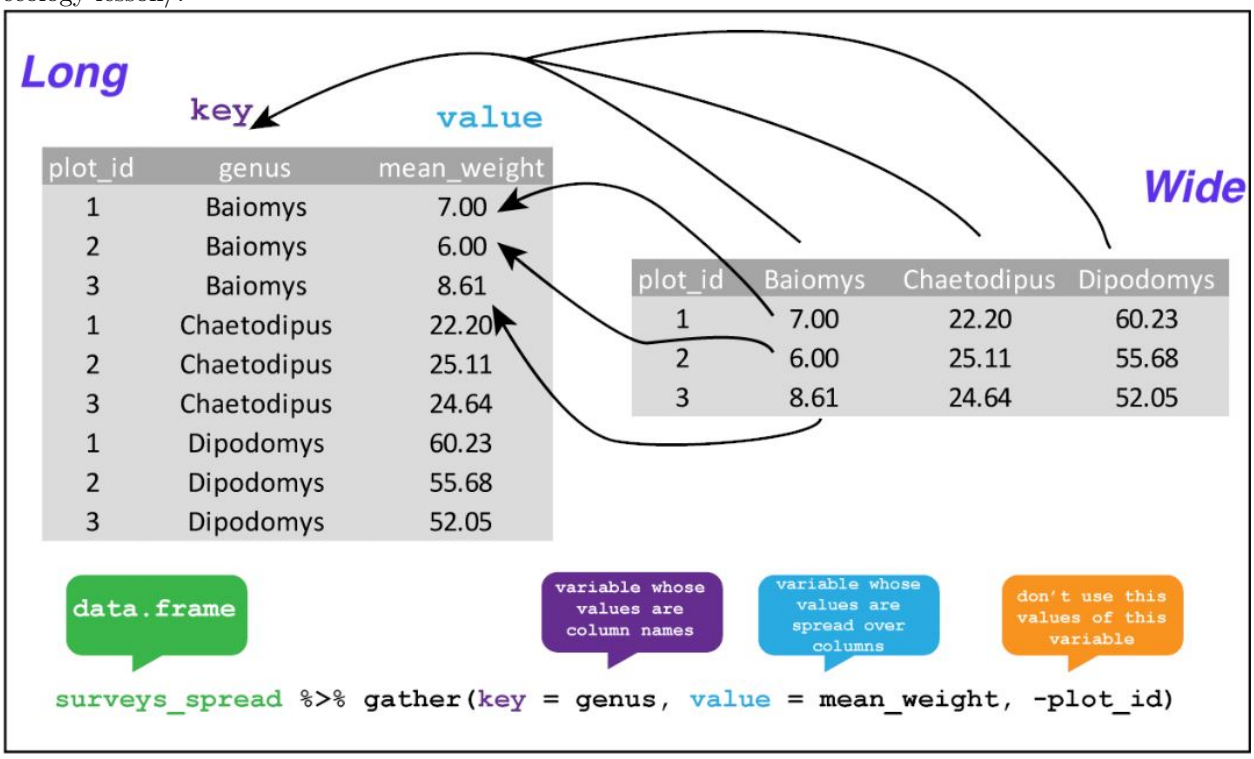
When we are in the opposite situation, where a variable is stretched across multiple columns, the `gather()` function comes in handy. For example, in the `long_survey` data each genus has its own column, but we would like to treat genus as a variable instead.

In this situation, we wish to gather all of these columns that represent values of the variable `genus` and turn them into a new variable (column). We then wish to add another column adjacent to this new variable that contains the values of the previous columns (Baiomys, Chaetodipus, etc.).

The `gather()` function takes four main arguments:
1. the data to be reshaped
2. the *key* column variable whose values form the existing column names (`genus`)
3. the *value* column whose values are spread over the cells
4. the names of the columns that we wish to fill the key variable

To make these wide data into a long format, we will use the `gather()` function from `tidyr` to gather the different genera into a single column.

Figure 6: Woo, K. & Hollister, J. (2019, March). Retrieved March 14, 2019, from http://datacarpentry.org/R-ecology-lesson/.




```
wide_survey %>%
  gather(key = genus, value = mean_weight, -plot_id) %>%
  head()
```

Notice that the `NA` values (stored as 0's) are not included in the re-gathered format. Spreading variables and then gathering is a clever way to balance a dataset, so that every replicate occurs the same number of times (e.g. every `genus` has a `mean_weight` for every `plot_id`).

Exercise 9: Spread the `combined` data frame with `year` as columns, `plot_id` as rows, and the number of genera per plot as the values. (hint: you will need to summarize before reshaping, and the function `n_distinct()` will be helpful in getting the number of unique genera)

```
# Exercise 9 code here!
```

Exercise 10: Now take your wide data frame and `gather()` it, so each row is a unique `plot_id` by `year` combination.

```
# Exercise 10 code here!
```

Workshop build on: 2019-03-29 11:06:00