Data Importing and Wrangling I

Graphical Analysis of Biological Data

This web page is available as a PDF file

These notes cover homeworks 6 and 7.

Reading and Resources

I've spread the reading assignments and related resources throughout these notes. *Read these notes carefully and thoroughly.* These notes provide answers for most of your questions. But, ask questions if you don't find the answer here.

Tibbles

R4ds Chapter 10: Tibbles, sections 1-3.

Tibbles are enhanced data frames that make coding in the `tidyverse` much easier.[1] We will work with tibbles throughout this course. You can convert any data frame to a tibble using the `as_tibble()` function. functions like `read_csv()` automatically create tibbles from your data so you will not usually have to think in tibble terms.

Importing data

Read R4ds Chapter 11: Data Import, sections 1, 2, and 5.

Cheatsheet: Data Import

File formats

The first step of any analysis is to import the raw data. Raw data are often stored in the rows and columns of software like Excel. Each row of the spreadsheet is one observation of related variables. Each column is one variable. The table below shows data about my cats. Each row is an observation for one cat. Each cat has variables name, breed, color, and personality.

```
## # A tibble: 3 x 4
##   Name  Breed              Color             Personality
##   <chr> <chr>              <chr>             <chr>
## 1 Pipit Ragamuffin         Gray              Spoiled rotten
## 2 Lynx  American Shorthair Lynx-point tabby Spoiled rotten
## 3 Jet   Egyptian Mau       Black             Spoiled rotten
```

Although R can read data directly from Excel (with the right packages), not all data are originally in Excel files. More often, data are stored in "comma-separated values" format, a type of text file that ends with `.csv`. Data in Excel and other formats can be exported as csv files.

The cat data in a csv text file looks like this. The value of each column is separated by a comma. Each observation begins on a new line. In this example, the first row has the names of each column.

[1] A tibble is an R class called "tbl" so "tibble"" is word-play on "table" and the pronunciation of "tbl."

```
Name,Breed,Color,Personality
Pipit,Ragamuffin,Gray,Spoiled rotten
Lynx,American Shorthair,Lynx-point tabby,Spoiled rotten
Jet,Egyptian Mau,Black,Spoiled rotten
```

The first row is not required to have column names. The data might begin in the first row of the file. The data might not begin until the 6th or 20th row. The first several lines might, for example, describe the data. In this example, the column header and data begin in row 6.

```
Data about my cats
Name: cat name
Breed: best guess based on traits but none are purebred.
Color:  the dominant color or pattern
Personality: purely subjective. The cats might debate otherwise.
Name,Breed,Color,Personality
Pipit,Ragamuffin,Gray,Spoiled rotten
Lynx,American Shorthair,Lynx-point tabby,Spoiled rotten
Jet,Egyptian Mau,Black,Spoiled rotten
```

Other file formats are possible. Tab-separated values use tabs instead of commas to separate the values in each row. Characters other than commas and tabs might separate data. Or, the values might not be separated by any character. Instead, the columns are fixed widths. You'll read more about fixed-width files below.

`readr` is the `tidyverse` package that imports data. `readr` has sensible defaults that makes it easy to import data. I will give you several different formats and variations to give you practice importing data.The `readr` fuctions you will use are `read_csv()` (most often), `read_tsv()` (like csv but tab-separated), and `read_fwf()` where you have to specify the column. Here are a few things to remember.

- If the file has only a row of column headers and data, then the default setting is usually sufficient.
- If the file has only data without column headers, then the `col_names` argument must be set to false.

```r
raw_data <- read_csv(file, col_names = FALSE)
```

- If the file has lines that have to be skipped, use the `skip` or `comment =` arguments.

```r
raw_data <- read_csv(file, skip = 5)
```

The data you import for this assignment will use only these arguments. In future exercises, we'll explore additional arguments that are often necessary for importing data.

Important!

`readr` from the `tidyverse` packages uses an underscore in the function names of most import functions, such as `read_csv()` and `read_tsv()`. Base R (not `tidyverse`) uses a dot, like `read.csv()` and `read.tsv()`. The tidyverse and base functions work in similar ways but have *important* differences. Therefore …

1. Use the `readr` version of a function. We are using the `tidyverse` packages. You must use functions from these packages to get the results I expect.
2. If you search the web for help, adding `tidyverse` as one of your search terms helps you get the results you need. *Check to be sure the examples you find are using underscores, not periods (e.g., `read_csv`, not `read.csv`).*
3. Reread points 1 and 2 above and commit them to memory. You cannot get maximum score if you're not meeting the expectations of this course.

File paths

You have to tell R where your data is located to import it. The location, or path, of a file is handled somewhat differently between Macs, Linux, and PCs. Macs and Linux use the unix system. PCs use the DOS system.

- Mac/Linux: `~/Documents/folder1/folder2/file.txt`
- PC: `C:\Documents\folder1\folder2\file.txt`

Take note of two important differences. Mac paths begin with `~`. PC paths begin with `C:\` (or some other letter). Mac paths use forward slashes (`/`). PC paths use backslashes (`\`).

You can use Mac paths directly in R. PC paths need to be modified because `\` is used by R as an "escape" character to do certain things. For example, `\n` creates a new line in R (and in unix systems in general), which you can see if you run this example in the console.

```
cat("This is one line.\nThis is a new line.")
```

```
## This is one line.
## This is a new line.
```

To fix the PC path, you can escape the backslash with a second backslash (which tells R you want to use an actual backslash) or replace the backslash with a forward slash, which parallels Mac paths.

- `C:\\Documents\\folder1\\folder2\\file.txt`
- `C:/Documents/folder1/folder2/file.txt`

Use either method but use it consistently. My examples will use the second method.

From here on, you will import data from files stored on your computer. Create a `data` folder *in the same folder as your `Rproj` file*. Put all of your data files for this course in your `data` folder so you can access them as needed for any assignment.

Let's look more closely at methods to access your data. Assume, for this example, that you have a folder called `bi485` in your Documents folder. Your `Rproj` and various homework folders are in the `bi485` folder. If you want to import data from `my_file.csv`, you could use this command.

```
dat <- read_csv("~/Documents/bi485/lastname_firstname/data/my_file.csv")  # Mac or
dat <- read_csv("C:/Documents/bi485/lastname_firstname/data/my_file.csv") # PC
```

This works fine, but what if have to open more than one data file? You would have to type the full path for each file. You can copy and paste the path, and change the name for each file. That is not efficient. Plus, if you change the location of the files, you have to to edit every path.

A better method is to store the path to the *folder* in an R variable, and then use the `file.path()`[2] function, which appends your file name to your path, and then opens the file. The path is a character vector so it must be quoted.

```
file_path <- "~/Documents/bi485/lastname_firstname/data/"  # Mac or
file_path <- "C:/Documents/bi485/lastname_firstname/data/" # PC

dat <- read_csv(file.path(file_path, "my_file.csv"))
```

The best method takes advantage of your `Rproj` file. When opened, the project file changes the working directory of R to the same folder as the project file. This step is equivalent to running `setwd("~/Documents/bi485/lastname_firstname/")` in the console. You can then use the "relative" file path, as shown here. R will automatically append your path to the working directory path set by your project file. Brilliant!

```
## Not recommended but OK for Mac, PC, and Linux
dat <- read_csv("data/my_file.csv"))

## Recommended for Mac, PC, and Linux
file_path <- "data/"

dat <- read_csv(file.path(file_path, "my_file.csv"))
```

[2]`file.path()` is a base R function.

Fixed Width files

Fixed-width files do not have defined delimiters to separate columns. Instead, each column has a fixed width. For example, characters 1-6 in a row might be Variable One, characters 7-22 might be Variable Two, and so on.

This format used to be very common, especially on older computer systems, but you can still find these types of data today. I ran across a few when I was looking for data to use in this class. Fixed-width files are space and time efficient for very large datasets, but there is no significant gain for smaller files.

The `read_fwf()` function reads fixed-width files. Like `read_csv()`, `read_fwf()` needs a file name to open, and you can use `skip =` or `comment =` arguments. You also must use the `col_positions` argument to define the position of each column.

Consider these fake data.

```
dat <- "aa111xx\nbb222yy\ncc333zz"
cat(dat)
```

```
## aa111xx
## bb222yy
## cc333zz
```

Characters 1-2 of each row are the first variable (Var1). Characters 3-5 are Var2 and characters 6-7 are Var3. You can read the data by specifying the width of each column or the position of each column in a vector.

To open the file using widths, you calculate the width of each column, then specify those values in a vector to the `fwf_width` argument. Var1 has two characters, Var2 has three characters, Var3 has two characters, so the vector is `c(2, 3, 2)`.

```
read_fwf(dat, fwf_widths(c(2, 3, 2)))
```

```
## # A tibble: 3 x 3
##   X1       X2 X3
##   <chr> <dbl> <chr>
## 1 aa      111 xx
## 2 bb      222 yy
## 3 cc      333 zz
```

To open the file by position, you calculate the starting position of each column, then specify those values in a vector passed to the `fwf_position` argument. Var1 starts at position 1, Var2 at position 3, and Var3 at position 6, so the vector is `c(1, 3, 6)`. You also have to provide another vector with the end position of each variable. In this case, the values are `c(2, 5, 7)`.

```
start_pos <- c(1, 3, 6)
stop_pos <- c(2, 5, 7)
read_fwf(dat, fwf_positions(start_pos, stop_pos))
```

```
## # A tibble: 3 x 3
##   X1       X2 X3
##   <chr> <dbl> <chr>
## 1 aa      111 xx
## 2 bb      222 yy
## 3 cc      333 zz
```

Each column is given a default name, which do are not the original variable names (Var1, Var2, Var3). The example below modifies the first example above to show how you can use the original variable names. The `widths` vector has the column widths. The `names` vector has the desired column names. The vector of names is passed to the `fwf_widths` argument with the `col_names` argument.

```
widths <- c(2, 3, 2)
names <- c("Var1", "Var2", "Var3")
```

```
read_fwf(dat, fwf_widths(widths,
                         col_names = names))
```

```
## # A tibble: 3 x 3
##   Var1   Var2 Var3
##   <chr> <dbl> <chr>
## 1 aa      111 xx
## 2 bb      222 yy
## 3 cc      333 zz
```

Reading fixed-width files takes more work than reading csv files. I have given you one fixed-width dataset to read in the assignment, so that you are familiar with some of the options. After that, we will work only (or at least mostly) with csv files.

Column types

The `read_csv()` and related functions are very good at guessing the type of data stored in each column but they are not perfect. They read in the first 1000 records to decide if the data are integers, numeric, characters, etc. Once it decides, then data of any different type will raise cause an error.

You can fix this in a couple ways. One is to change `guess_max`, which is the number of observations read before a decision is made (see the help file for `read_csv()` for more information). The better method is to provide the data type of each column as a vector with the `col_types` argument. For example,

```
dat <- read_csv("my_file.csv", col_types = "cncidl")
```

In this example, the column types are character, numeric, character, integer, double, and logical. `read_csv()` will apply your types (if possible) to the imported data. You can see all of the options for `col_types` by typing `?read_csv()` in the console.

Pipes

Read

R4ds Chapter 18: Pipes, sections 1-3.

I condensed much of what follows from the assigned reading. You can read this instead of the chapter but I suggest you go back and read the sections above once you have more familiarity with the tidyverse.

Pipes allow you to pass the output from one function directly into another function. The pipe function looks like %>% and works very well with `tidyverse` packages. Pipes make code writing *and* reading much easier. Here's an example by analogy from the reading, based on a nursery rhyme about little bunny Foo Foo..

> Little bunny Foo Foo Went hopping through the forest Scooping up the field mice And bopping them on the head

From this, we learn that little bunny Foo Foo was not particularly nice. Foo Foo also did three actions: hop, scoop, and bop. Let's start by defining an object to represent little bunny Foo Foo. (This is fake code called pseudocode that represents the coding logic but it does not run.)

Each action Foo Foo did is reresented by a function: `hop()`, `scoop()`, and `bop()`. Using this object and these actions, there are (at least) four ways the story could be told in code:

- Save each intermediate step as a new object.
- Overwrite the original object many times.
- Compose functions.
- Use the pipe.

Let's work through each approach with the code to learn the advantages and disadvantages of each.

Approach 1: create new objects for each step

This is the simplest approach.

```
foo_foo_1 <- hop(foo_foo, through = forest)
foo_foo_2 <- scoop(foo_foo_1, up = field_mice)
foo_foo_3 <- bop(foo_foo_2, on = head)
```

The main downside of this form is that it forces you to name each intermediate element. If there are natural names, this is a good idea, and you should do it. But many times, like in this example, there aren't natural names, and you add numeric suffixes to make the names unique. That leads to several problems:

- The code is cluttered with unimportant names
- You have to carefully increment the suffix on each line.
- If you have to insert a new function between two steps, say between `hop()` and `scoop()`, then your naming system has to be revised.
- This method also requires a bit more memory but rarely is that an issue, except for very large data sets. And Foo Foo is just a small (but not nice) bunny.

Overwrite the original

Instead of creating intermediate objects at each step, you could overwrite the original object:

```
foo_foo <- hop(foo_foo, through = forest)
foo_foo <- scoop(foo_foo, up = field_mice)
foo_foo <- bop(foo_foo, on = head)
```

This is less typing (and less thinking), so you are less likely to make mistakes. However, there are two problems:

- Debugging is painful. If you make a mistake you'll need to re-run the complete pipeline from the beginning to find out which function has the error.
- The repetition of the object being transformed (we've written foo_foo six times!) obscures what's changing on each line.

Function composition

Another approach is to abandon assignment and just string the function calls together:

```
bop(
  scoop(
    hop(foo_foo, through = forest),
    up = field_mice
  ),
  on = head
)
```

Here the disadvantage is that you have to read from inside-out, from right-to-left, and that the arguments end up spread far apart (evocatively called the dagwood sandwhich problem). In short, this code is hard for a human to consume. This type of code is common. You even saw it earlier:

```
dat <- read_csv(file.path(file_path, your_file))
```

The first function to run is the inner function, `file.path()` because it has to assemble the path and file name. `file.path()` then passes the result to `read_csv()`. You could so the same thing like this:

```
the_file <- file.path(file_path, my_file)
dat <- read_csv(the_file)
```

These are all valid methods but the longer your series of functions, the harder it gets to read. So …

Use the pipe

Finally, we can use the pipe (%>%):

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mice) %>%
  bop(on = head)
```

This form is the easiest to read because it emphasizes verbs, not nouns. You can read this series of function compositions like it's a set of imperative actions. Foo Foo hops, then scoops, then bops. The downside, of course, is that you need to be familiar with the pipe, especially if you have lots of programming experience.

If you are learning R programming from scratch, then the '`%>%` pipe is easy to learn.

Note: Instead of typing a space, %, then >, then %, and then another space, type `cmd/alt + shift + m` in RStudio to insert the whole pipe symbol at once!

I will use pipes throughout my examples. I encourage you to use the pipe in your own code. In realilty, you may use the pipe in some parts of your code and not use the pipe in other parts. It depends on your needs, which become clearer with experience.

Here is some pseudocode that outlines a typical set of steps you will take in the coming exercises. First, you import the raw data. Then, you pipe the raw data to a series of functions to wrangle it into the format(s) you want for analysis. You then store the wrangled data in another object (here called final_data). You then use final_data for analysis and graphing.

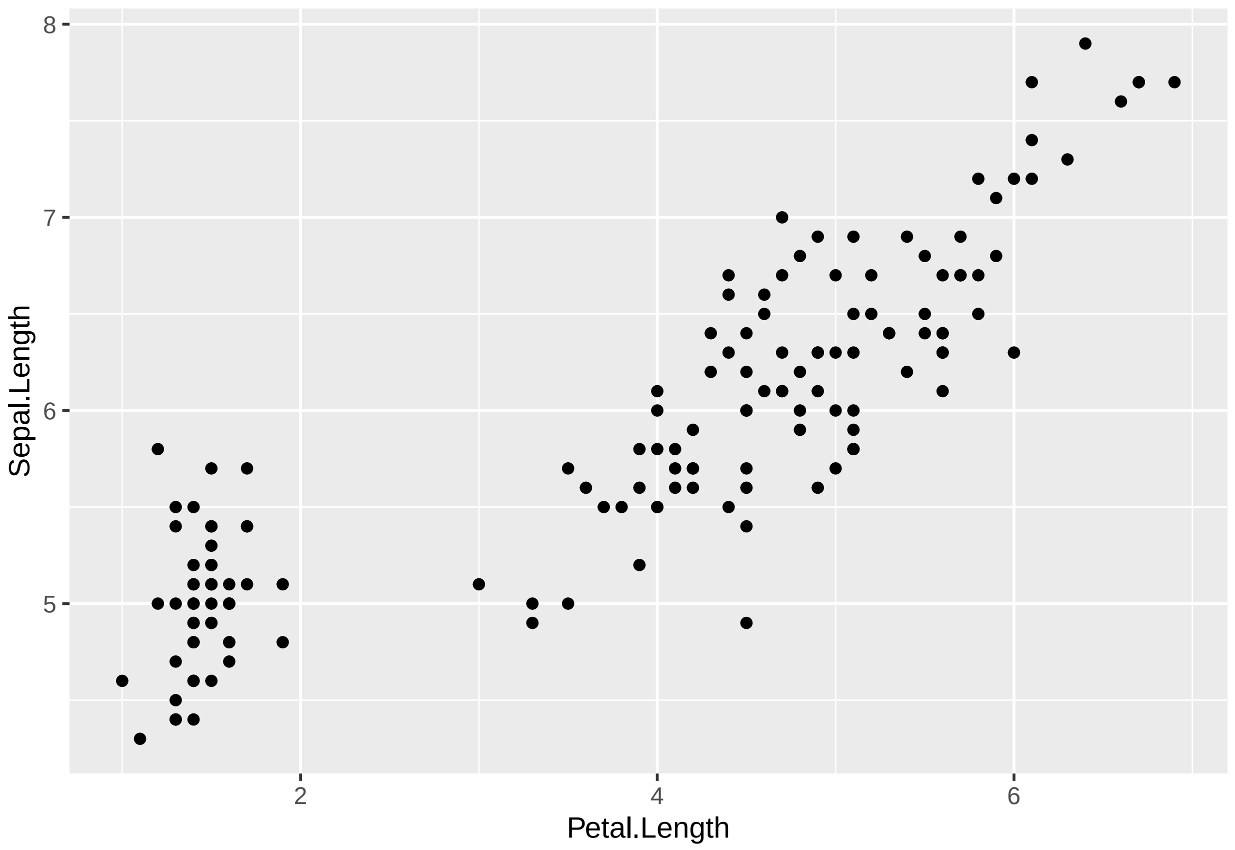
```
raw_data <- import the data

final_data <- raw_data %>%
  select some columns %>%
  filter the data %>%
  mutate the data %>%
  summarize the data
```

Recall that you tell `ggplot()` which data to plot with the `data =` argument, such as

```
ggplot(data = iris) +
  geom_point(aes(x = Petal.Length,
                 y = Sepal.Length))
```

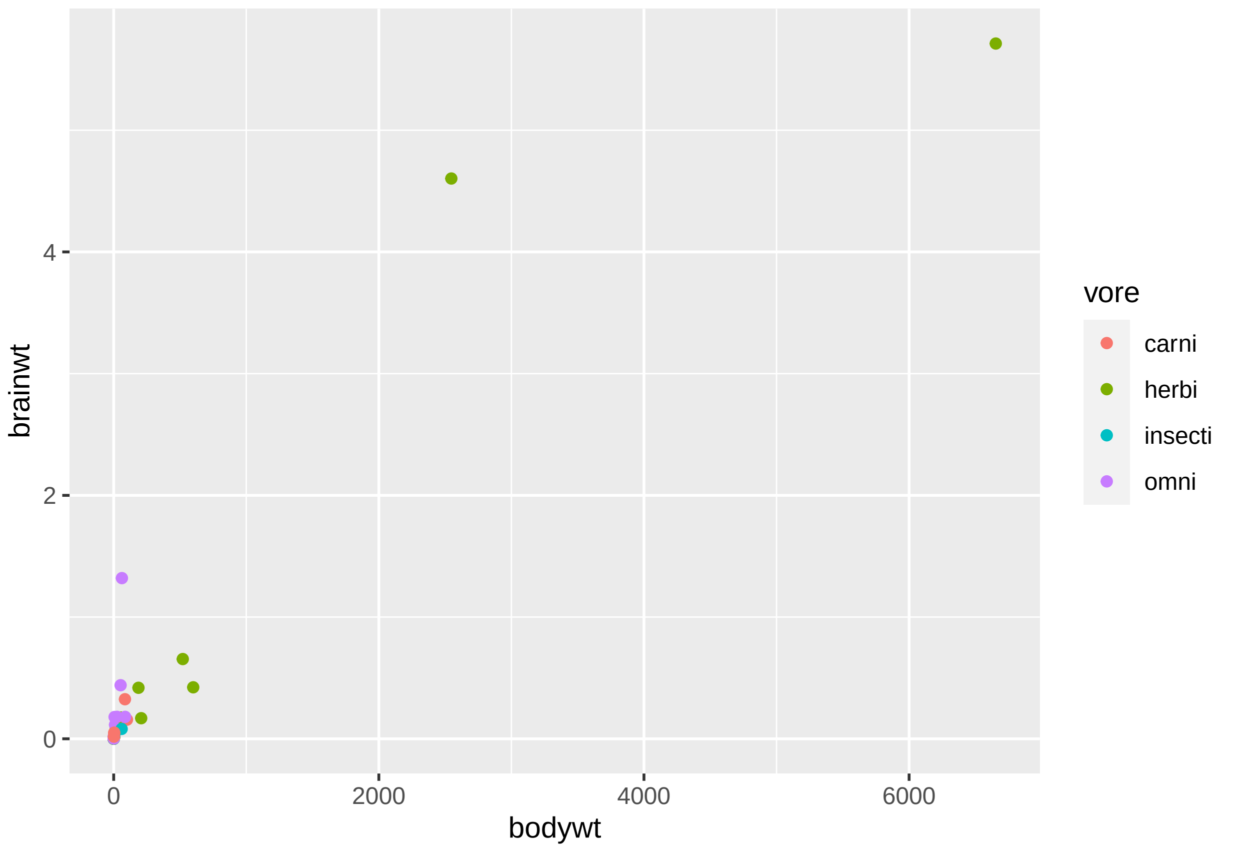
You can instead pipe your data to `ggplot()`:

```
iris %>%
  ggplot() +
  geom_point(aes(x = Petal.Length,
                 y = Sepal.Length))
```



While there is little difference between the those two brief examples above, the pipe lets you easily do some data wrangling before plotting your data. For example, the `drop_na()` function removes all rows with missing data (`NA`). (You may recall seeing warnings about dropping missing data when you made some graphs with the `msleep` data.) A simple method to remove missing data for columns you want to plot before plotting is

```
msleep %>%
  select(bodywt, brainwt, vore) %>%
  drop_na() %>%
  ggplot() +
  geom_point(aes(x = bodywt,
                 y = brainwt,
                 color = vore))
```



Tidy Data

Read R4ds Chapter 12: Tidy Data, sections 1-3, 7.

"Tidy data" refers to a specific layout of data, as explained in the required reading. Each row is an observation and each column is a *unique* variable. However, consider this data set with four species of beetles that were counted from three habitats across three years.

```
beetles
```

```
## # A tibble: 12 x 5
##    species habitat `2012` `2013` `2014`
##    <chr>   <chr>    <dbl>  <dbl>  <dbl>
##  1 Pasi    sand         0      0      0
##  2 Lebi    sand        10     12      0
##  3 Csexg   sand       130     21      8
##  4 Cunip   sand         0      0      1
##  5 Pasi    brush      403    278    369
##  6 Lebi    brush        0    114     60
##  7 Csexg   brush        0      0      0
##  8 Cunip   brush        0      0      0
##  9 Pasi    woods        0      0      0
## 10 Lebi    woods      106    160     10
## 11 Csexg   woods        0     10      0
## 12 Cunip   woods      190    130    240
```

The years (2012, 2013, 2014) are values of a single "Year" variable. These data are not tidy but can be made tidy with the `gather()` function.

```
beetles_tidy <- beetles %>%
  gather(key = year,
         value = abundance,
         `2012`:`2014`)

beetles_tidy
```

```
## # A tibble: 36 x 4
##    species habitat year  abundance
##    <chr>   <chr>   <chr>     <dbl>
##  1 Pasi    sand    2012          0
##  2 Lebi    sand    2012         10
##  3 Csexg   sand    2012        130
##  4 Cunip   sand    2012          0
##  5 Pasi    brush   2012        403
##  6 Lebi    brush   2012          0
##  7 Csexg   brush   2012          0
##  8 Cunip   brush   2012          0
##  9 Pasi    woods   2012          0
## 10 Lebi    woods   2012        106
## # ... with 26 more rows
```

The three years are now under a single column (the `key` argument gives this column its name), with the abundance of each species in the `abundance` column (the `value` argument gives this column its name). The 2012`:`2014 argument tells the `gather()` function which columns to gather. You'll learn more about the many ways to specify the columns when you read the text.

The trick to remember gathering is that the `key` argument will make a column filled with the column names; the `value` argument will make a column filled with the data that originally were in the separate columns. See Figure 12.2 of R4ds.

Here are similar data but with species as column names.

```
plot <- c(2,2,2,3,3,3,3,3,1,1,1,1)
sample <- c(3,6,7,1,2,3,7,10,1,3,6,10)
Pasi <- c(1,1,1,0,1,1,0,1,1,1,1,1)
Lebi <- c(0,0,0,1,0,0,0,0,0,0,0,0)
Csexg <- c(0,0,0,0,0,0,0,0,0,0,0,0)
Cunip <- c(0,0,0,0,0,0,0,0,0,0,0,0)
Gale <- c(0,0,0,0,0,0,0,0,0,0,0,0)
beetles2 <- tibble(plot, sample, Pasi, Lebi, Csexg, Cunip, Gale)
```

```
beetles2
```

```
## # A tibble: 12 x 7
##     plot sample  Pasi  Lebi Csexg Cunip  Gale
##    <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1     2      3     1     0     0     0     0
##  2     2      6     1     0     0     0     0
##  3     2      7     1     0     0     0     0
##  4     3      1     0     1     0     0     0
##  5     3      2     1     0     0     0     0
##  6     3      3     1     0     0     0     0
##  7     3      7     0     0     0     0     0
##  8     3     10     1     0     0     0     0
##  9     1      1     1     0     0     0     0
## 10     1      3     1     0     0     0     0
## 11     1      6     1     0     0     0     0
## 12     1     10     1     0     0     0     0
```

This time, each species name is a value of a single “Species” variable. The data need to be made tidy.

```
beetles2_tidy <- beetles2 %>%
  gather(key = species,
         value = abundance,
         -c(plot, sample))

beetles2_tidy
```

```
## # A tibble: 60 x 4
##     plot sample species abundance
##    <dbl>  <dbl> <chr>       <dbl>
##  1     2      3 Pasi            1
##  2     2      6 Pasi            1
##  3     2      7 Pasi            1
##  4     3      1 Pasi            0
##  5     3      2 Pasi            1
##  6     3      3 Pasi            1
##  7     3      7 Pasi            0
##  8     3     10 Pasi            1
##  9     1      1 Pasi            1
## 10     1      3 Pasi            1
## # ... with 50 more rows
```

The `tidyr` package from the `tidyverse` provides many functions besides `gather()` but we will focus on “gathering” untidy data to make it tidy.

Data transformation

Read R4ds Chapter 5: Data Transformation sections 1-4.

Cheatsheet Data Transformation

An important step in data wrangling is transforming the data to suit the needs of your analysis. “Transforming” does not mean you are changing the data to get a particular result. Instead, transformation refers to everything from summarizing your data (e.g., counts, means, standard deviations), to recording your data (e.g., 1 = “female”, 2 = “male”), to performing statistical transformations (e.g., log or arcsine transformation) that improve your analysis and interpretation.

Chapter 5 of R4ds covers many tools that you will use frequently for data analysis. This assignment will cover four of them.

- `arrange()` sorts the data in an order you specify,
- `rename()` renames columns,
- `filter()` chooses rows based on criteria you specify, and
- `select()` chooses columns based on criteria you specify.

For this assignment, you will run some of the examples in the text and anwer the questions. You will also practice importing, wrangling, and graphing of several data sets.