Notes 03 - Data Import and Wrangling

STS 2300 (Spring 2024)

Updated: 2024-09-07

Table of Contents

Reading for Notes 03

Read Section 4.1 of ModernDive to learn about importing data from URLs and using the RStudio interface.

Read the Chapter 3 of ModernDive to learn more about data wrangling.

Learning Goals for Notes 03

- Be able to use R Studio to generate code for importing data of various types into R.
- Understand the usefulness of the pipe operator and be able to write multi-step code using it.
- Be able to subset data frames by rows and columns as desired.
- Be able to create new variables to add to a data frame.
- Understand the difference between wide and long format data and be able to convert between the two types.

Importing data into R

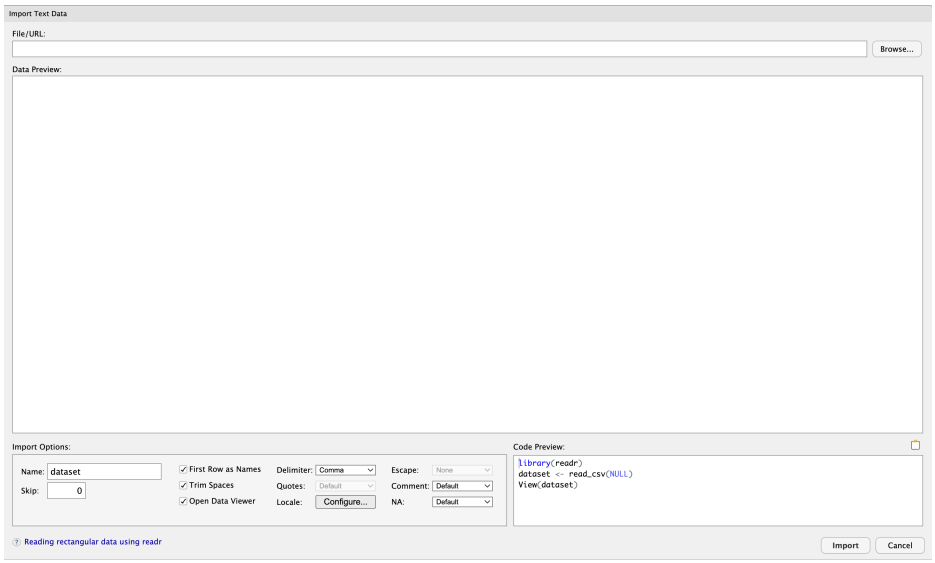
There are *many* ways to import data into R, and several functions have been created to make the process easier for different types of data. In STS 2300, we will focus on how to use the "Import Dataset" option in the Environment section. This will allow us to view a preview of the data and choose options as we go. This option also produces code so that we can read the data in the exact same way the next time without going through the menu options again.

Before importing data, I encourage you to put the data in the same folder as your R script and then do the following:

> Session –> Set Working Directory –> To Source File Location

This will make it so that the file path for the data is not specific to your computer and will allow others to run the same code.

Let's try this out in our **notes03.R** file. Download the **NC Bridges.csv** file and put it in the same folder as your **notes03.R** file. Then go to Import Dataset and choose **From Text (readr)**. This option works well for .csv files.



Import Dataset Screen

We can click **Browse…** to find the data. Once we do that, a preview will show up in the big screen in the middle. On the bottom left are options we can change, and on the bottom right is code that will read the data into R. Before you hit **Import** be sure to copy the code that is produced so that you can add it to an R script for future use.

Use the space below to add notes about reading in data this way.

The Pipe Operator (%>%)

Section 3.1 of the textbook talks about the symbol %>%, which we call the pipe operator (like how <- is the assignment operator). The pipe operator is useful when we want to perform a series of actions on a data frame. It's main purpose is to take an object on the left and "pipe it into" a function on the right.

Note: There are actually **two** pipe operators now. In some places, you may see |> used instead of %>%. For example, our supplementary textbook R for Data Science recently switched to using |>.For the purposes of our class, these two are almost identical. %>% was created first and is part of several packages in the "tidyverse" (like `dplyr`). Many R users found "piping" useful for coding and |> was eventually added to "base R".

So what is the pipe operator? To illustrate its usefulness, let's start with an example where we *don't* use the pipe operator. We'll learn what each function below is doing soon, but for now pay attention to how I have to keep creating new objects and then using that object as the first argument in my next function.

```
library(dplyr) # filter, mutate, summarize, and %>% are part of this package

# Subset the mtcars dataset to only include cars with automatic transmission
auto <- filter(mtcars, am == 0)

# Create variable for car weight in pounds (instead of 1000s of pounds)
auto <- mutate(auto, wt_lbs = 1000 * wt)

# Then calculate the mean weight for each number of cylinders
mean_wt_by_cyl <- summarize(auto, mean_wt = mean(wt_lbs), .by = cyl)

# Then arrange in ascending order by number of cylinders
mean_wt_by_cyl <- arrange(mean_wt_by_cyl, cyl)

# Look at our new dataset with averages by cylinder for automatic
transmission cars
mean_wt_by_cyl
```

```
##   cyl  mean_wt
## 1   4 2935.000
## 2   6 3388.750
## 3   8 4104.083
```

Alternatively, I could also do this all in one step, but it ends up being hard to read. In the code below, it's hard to see which arguments belong to which functions and what is ultimately being done with the code.

```
mean_wt_by_cyl <- arrange(summarize(mutate(filter(mtcars,
                                                  am == 0),
                                           wt_lbs = 1000 * wt),
                                    mean_wt = mean(wt_lbs),
                                    .by = cyl),
                          cyl)
mean_wt_by_cyl
```

```
##   cyl  mean_wt
## 1   4 2935.000
## 2   6 3388.750
## 3   8 4104.083
```

The pipe operator allows us to do all of this with more efficient and readable code that doesn't create a bunch of temporary objects along the way. Below is the same task done using pipe operators.

When we are checking if two things are equivalent (like when we use filter()), we use == to do that. A single = is used for arguments in a function.

```
mean_wt_by_cyl <- mtcars %>%
  filter(am == 0) %>%
  mutate(wt_lbs = 1000 * wt) %>%
  summarize(mean_wt = mean(wt_lbs), .by = cyl) %>%
  arrange(cyl)
mean_wt_by_cyl
```

```
##   cyl  mean_wt
## 1   4 2935.000
## 2   6 3388.750
## 3   8 4104.083
```

Notice that for each function, I'm no longer including my data frame as the first argument of the function. That's because the pipe operator tells R to take whatever we just did and put it in as the first argument of our next function. You can read the symbol %>% (or |>) as the phrase "and then".

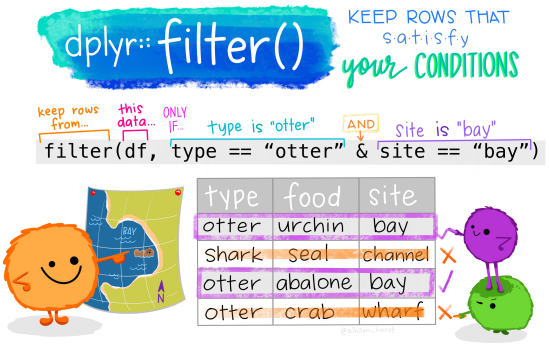
Thus, the code above:

- is creating an object called `mean_wt_by_cyl` that takes the `mtcars` dataset **and then**
- subsets it to only include automatic transmissions **and then**
- creates a new variable for weight in pounds **and then**
- calculates the mean weight for each number of cylinders **and then**
- arranges the output by number of cylinders.

Using piping will likely take some getting used to, but over time this becomes a convenient way to talk through your code as you write it.

Subsetting data by rows and columns

Using `filter()` to select rows meeting a condition



`filter()` function cartoon (from https://github.com/allisonhorst/stats-illustrations)

The `filter()` function is used to subset our data to only include certain rows. In the example above, I used it to only include cars with automatic transmissions (i.e. when the `am` variable was equal to 0). In general, our use of the filter function will look like:

```
filter(dataset_name, logical_condition, ...)
```

If we've used a pipe operator before `filter()` (like we saw above), we can leave off the first highlighted argument. After that, we just need to tell the function how to subset our data. For example, above I wrote `am == 0`. This meant that I wanted all the rows where `am` was 0. I can also use symbols like >, <, >=, <=, and `!=` (see Section 3.2 of the textbook).

If I want to have multiple conditions (e.g. automatic transmissions with 4 cylinders), I can add more conditions where the `...` is. Just separate each condition with a comma. If you want to meet *at least one* of several conditions, you can put a | in between conditions instead of a comma.

Typically, we will want to store our new filtered data frame in an object using the assignment operator. Below is an example where I subset the `mtcars` data to only include automatic transmissions (where `am` is 0), so I call my new data frame auto.

```
auto <- filter(mtcars, am == 0)
```

Practice:

- Create a subset of our NC Bridges data that only includes bridges from Alamance County (call it `alam_bridges`)
- Create a subset that includes bridges that are structurally deficient ("SD") and functionally obsolete ("SO").
- Create a subset that includes bridges in Alamance County that are either structurally deficient OR functionally obsolete.

Using `select()` to choose only certain columns

If we have more variables than we want in our data, we can use `select()` to remove some of them. In general, that might look like this:

```
select(dataset_name, variable_to_include, other_variable_to_include, ...)
```

Just like with `filter()`, we can omit the data set name if we use a pipe operator before `select()`. We can also ask for everything but a certain variable by putting a minus sign before the variable name. If we want several variables that are next to each other we can use the format `variable_a:variable_b` to return everything from `variable_a` to `variable_b`.

If I put a minus sign in front of a variable (or group of variables), it will give me all of the variables other than that one.

Practice:

- Update your `alam_bridges` data frame to only include `ROUTE`, `ACROSS`, `YEARBUILT` and `SR`

Using `mutate()` to create new variables

`mutate()` function cartoon (from https://github.com/allisonhorst/stats-illustrations)

In an example above, I used `mutate()` to convert the `wt` variable in `mtcars` (which records weight in thousands of pounds) into a new variable that calculates weight in pounds. Below is the general structure of the `mutate()` function followed by an example using `mtcars`.

```
mutate(dataset_name, newvar_name = calculation, ...)

mycars <- mutate(mtcars, wt_lbs = wt * 1000)
```

My code used the `mtcars` data (dataset_name), created a new variable called `wt_lbs`, and calculated this as `wt * 1000`. Below I'm storing the result in a new data frame called `mycars` so I don't edit the built in R data frame. (Note: If I wanted to create more than one variable, I could add other variables where the `...` is.)

Let's verify that this code did what I want...

```
mycars %>%
  select(wt, wt_lbs) %>%
  head(n = 5)
```

```
##                      wt wt_lbs
## Mazda RX4         2.620   2620
## Mazda RX4 Wag     2.875   2875
## Datsun 710        2.320   2320
## Hornet 4 Drive    3.215   3215
## Hornet Sportabout 3.440   3440
```

Practice: Using the NC Bridges data, create a new variable called `AGE` that takes 2023 minus `YEARBUILT`. Then print the year built and age for the last 10 bridges in the data.

Long vs. Wide Data

Our textbook calls long data ("tidy data") and wide data ("non-tidy data") (see Section 4.2 of ModernDive). In many cases, wide data is nicer for humans to look at, but long data is nicer for computers to interact with. The distinction will become more clear when we start visualizing data.

Below I've read in the exact same data stored in each format. The data represents the number of baby birds that were raised in nest boxes at the Museum of Life + Science in Durham, NC. To save space, I'm only displaying the first and last 3 rows of the long data.

```
birds_wide <- read.csv("https://raw.githubusercontent.com/nbussberg/STS2300-Fall2024/main/Data/nestbox_lands_wide.csv")
birds_long <- read.csv("https://raw.githubusercontent.com/nbussberg/STS2300-Fall2024/main/Data/nestbox_lands_long.csv")

birds_wide
```

```
##              Species X2012 X2013 X2014 X2015 X2016 X2017 X2018 X2019 X2020
## 1   Eastern Bluebird    20    20    27    18    19     3    16     8    NA
## 2 Carolina Chickadee     6    13     3     7     6     9    12     0    NA
## 3         House Wren     0     0     0     0     0    11     5     4    NA
##   X2021 X2022 X2023
## 1     5    17    13
## 2     0     4    14
## 3     5     3     9
```

```
head(birds_long, n = 3)
```

```
##   Year            Species Fledged
## 1 2012   Eastern Bluebird      20
## 2 2012 Carolina Chickadee       6
## 3 2012         House Wren       0
```

```
tail(birds_long, n = 3)
```

```
##    Year            Species Fledged
## 34 2023   Eastern Bluebird      13
## 35 2023 Carolina Chickadee      14
## 36 2023         House Wren       9
```

Question: Why do you think each of the years in `birds_wide` starts with an X?

Answer: This was done because object names (in this case the column vectors) cannot begin with a number. Note: we also learned objects can't have spaces in their names.

We can use the `pivot_wider()` and `pivot_longer()` functions from the `tidyr` packages to switch between formats (if we don't already have both like this example). Let's start by converting `birds_wide` to a long format.

In the code below, I'm piping my `birds_wide` data frame into the `pivot_longer()` function as the first argument. After that:

- `cols = -Species` says we will convert all of the columns other than species
- `names_to = "Year"` says that column names will go in a variable called Year
- `values_to = "Fledged"` says that the values within those columns will go into a variable called Fledged.

To save space, I'm using `head()` to only display the first five rows of my new long data frame. Notice how the year values match the first five columns from our wide data and the values match the first row (where `Species` was Eastern Bluebird).

```
library(tidyr)

birds_wide %>%
  pivot_longer(cols = -Species, names_to = "Year", values_to = "Fledged") %>%
  head(5)
```

```
## # A tibble: 5 × 3
##   Species          Year  Fledged
##   <chr>            <chr>   <int>
## 1 Eastern Bluebird X2012      20
## 2 Eastern Bluebird X2013      20
## 3 Eastern Bluebird X2014      27
## 4 Eastern Bluebird X2015      18
## 5 Eastern Bluebird X2016      19
```

There are still some issues with the Year column, but we could solve those as we learn more about R later.

To go from long to wide format, we use `pivot_wider()`. In the code below, I'm piping `birds_long` into the first argument of my `pivot_wider()` function and then: - `names_from = Year` says to make column names out of the `Year` variable - `values_from = Fledged` says to fill in the data tieh values from the `Fledged` variable.

```
birds_long %>%
  pivot_wider(names_from = Year, values_from = Fledged)
```

```
## # A tibble: 3 × 13
##   Species  `2012` `2013` `2014` `2015` `2016` `2017` `2018` `2019` `2020`
`2021`
##   <chr>     <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>
<int>
## 1 Eastern…     20     20     27     18     19      3     16      8     NA
5
## 2 Carolin…      6     13      3      7      6      9     12      0     NA
0
## 3 House W…      0      0      0      0      0     11      5      4     NA
5
## # ℹ 2 more variables: `2022` <int>, `2023` <int>
```

At the top of this section, I linked to more examples of going between wide and long data in our ModernDive textbook. You can also find examples in Sections 5.3 and 5.4 of R for Data Science and Section 6.2 of Modern Data Science with R.

Revisiting the Learning Goals for Notes 03

- Be able to use R Studio to generate code for importing data of various types into R.
  - There are several data sets in the data folder on our Moodle page. Try reading one of the Excel files into R.
- Understand the usefulness of the pipe operator and be able to write multi-step code using it.
  - What phrase do we use when reading the %>% symbol in our code?
- Be able to subset data frames by rows and columns as desired.
  - Create a subset of the `mtcars` data that only includes cars that get over 25 mpg and are manual transmission
  - Create a subset of the `mtcars` data that only includes the `mpg`, `am`, `cyl`, and `wt` variables
  - Combine the two steps above with a pipe operator and sort the data by mpg (Hint: Use `arrange()` from the example at the top.)

- Be able to create new variables to add to a data frame.
  - Using the `birds_wide` data, create a variable called Total that adds up the number of species from all of the years. (Note: Remember that NAs can't be included in this)
  - Using the Loblolly data, convert the `height` variable from feet to meters. (Note: You can divide feet by 3.281 to get meters)
- Understand the difference between wide and long format data and be able to convert between the two types.
  - Is the MTH_STS_Majors (on Github) data set wide or long format? How do you know? Convert it to the opposite format.