# Tidy Data in R

In this lesson we will cover the basics of data in R and will do so from a somewhat opinionated viewpoint of "Tidy Data". There are other paradigms and other ways to work with data in R, but focusing on Tidy Data concepts and tools (a.k.a., The Tidyverse) gets people to a productive place the quickest. For more on data analysis using the Tidyverse, the best resource I know of is R for Data Science. The approaches we will cover are very much inspired by this book.

## Lesson Outline

- Data in R: The data frame
- Reading in data
- Tidy data

## Exercises

- Exercise 3.1
- Exercise 3.2
- Exercise 3.3

## Data in R: The data frame

Simply put, a data structure is a way for programming languages to handle storing information. Like most languages, R has several structures (vectors, matrix, lists, etc.). But R was originally built for data analysis, so the data frame, a spreadsheet like structure with rows and columns, is the most widely used and useful to learn first. In addition, the data frame (or is it data.frame) is the basis for many modern R pacakges (e.g. the tidyverse) and getting used to it will allow you to quickly build your R skills.

*Note:* It is useful to know more about the different data structures such as vectors, lists, and factors (a weird one that is for catergorical data). But that is beyond what we have time for. The best source on this information, I think, is Hadley Wickham's Data Structures Chapter in Advanced R.

*Another note:* Data types (e.g. numeric, character, logcial, etc.) are important to know about too, but details are more than we have time for. Take a look at the chapter on vectors in R for Data Science, in particular Section 20.3.

*And, yet another note:* Computers aren't very good at math. Or at least they don't deal with floating point data they way many would think. First, any value that is not an integer, is considered a "Double." These are approximations, so if we are looking to compare to doubles, we might not always get the result we are expecting. Again, R4DS is a great read on this: Section on Numeric Vectors. But also see this take from Revolution Analytics and techradar.

*Last note, I promise:* Your elementary education was wrong. In other words rounding 2.5 is 2 and not 3, but rounding 3.5 is 4. There are actually good reasons for this. Read up on the IEEE 754 standard rules on rounding.

### Build a data frame

Best way to learn what a data frame is is to look at one. Let's now build a simple data frame from scratch with the `data.frame()` function. This is mostly a teaching excercise as we will use the function very little in the excercises to come.

```r
# Our first data frame

my_df <- data.frame(names = c("joe","jenny","bob","sue"),
                    age = c(45, 27, 38,51),
                    knows_r = c(FALSE, TRUE, TRUE,FALSE),
                    stringsAsFactors = FALSE)
my_df
```

```
##   names age knows_r
## 1   joe  45   FALSE
## 2 jenny  27    TRUE
## 3   bob  38    TRUE
## 4   sue  51   FALSE
```

That created a data frame with 3 columns (names, age, knows_r) and four rows. For each row we have some information on the name of an individual (stored as a character/string), their age (stored as a numeric value), and a column indicating if they know R or not (stored as a boolean/logical).

If you've worked with data before in a spreadsheet or from a table in a database, this rectangular structure should look somewhat familiar. One way (there are many!) we can access the different parts of the data frame is like:

```r
# Use the dollar sign to get a column
my_df$age
```

```
## [1] 45 27 38 51
```

```r
# Grab a row with indexing
my_df[2,]
```

```
##   names age knows_r
## 2 jenny  27    TRUE
```

At this point, we have:

- built a data frame from scratch
- seen rows and columns
- heard about "rectangular" structure
- seen how to get a row and a column

The purpose of all this was to introduce the concept of the data frame. Moving forward we will use other tools to read in data, but the end result will be the same: a data frame with rows (i.e. observations) and columns (i.e. variables).

## Reading in data

Completely creating a data frame from scratch is useful (especially when you start writing your own functions), but more often than not data is stored in an external file that you need to read into R. These may be delimited text files, spreadsheets, relational databases, SAS files ... You get the idea. Instead of treating this subject exhaustively, we will focus just on a single file type, the `.csv` file, that is very commonly encountered and (usually) easy to create from other file types. For this, we will use the Tidyverse way to do this and use `read_csv()` from the `readr` pacakge.

The `read_csv()` function is a re-imagined version of the base R fucntion, `read.csv()`. This command assumes a header row with column names and that the delimiter is a comma. The expected no data value is NA and by default, strings are NOT converted to factors. This is a big benefit to using `read_csv()` as opposed to `read.csv()`. Additionally, `read_csv()` has some performance enhancements that make it

preferrable when working with larger data sets. In my limited experience it is about 45% faster than the base R options. For instance a ~200 MB file with hundreds of columns and a couple hundred thousand rows took ~14 seconds to read in with `read_csv()` and about 24 seconds with `read.csv()`. As a comparison at 45 seconds Excel had only opened 25% of the file!

Source files for `read_csv()` can either be on a local hard drive or, and this is pretty cool, on the web. We will be using the former for our examples and exercises. If you had a file available from a URL it would be accessed like `mydf <- read.csv("https://example.com/my_cool_file.csv")`. As an aside, paths and the use of forward vs back slash is important. R is looking for forward slashes ("/"), or unix-like paths. You can use these in place of the back slash and be fine. You can use a back slash but it needs to be a double back slash ("\\"). This is becuase the single backslash in an escape character that is used to indicate things like newlines or tabs.

For today's workshop we will focus on both grabbing data from a local file and from a URL, we already have an example of this in our `nla_analysis.R`. In that file look for the line where we use `read_csv()`

For your convenience, it looks like:

```
nla_wq_all <- read_csv("nla2007_chemical_conditionestimates_20091123.csv")
```

And now we can take a look at our data frame

```
nla_wq_all
```

```
## # A tibble: 1,252 x 51
##    SITE_ID VISIT_NO SITE_TYPE LAKE_SAMP TNT   LAT_DD LON_DD ST    EPA_REG
##    <chr>      <dbl> <chr>     <chr>     <chr>  <dbl>  <dbl> <chr> <chr>
##  1 NLA066~        1 PROB_Lake Target_S~ Targ~   49.0 -114.  MT    Region~
##  2 NLA066~        1 PROB_Lake Target_S~ Targ~   33.0  -80.0 SC    Region~
##  3 NLA066~        2 PROB_Lake Target_S~ Targ~   33.0  -80.0 SC    Region~
##  4 NLA066~        1 PROB_Lake Target_S~ Targ~   28.0  -97.9 TX    Region~
##  5 NLA066~        2 PROB_Lake Target_S~ Targ~   28.0  -97.9 TX    Region~
##  6 NLA066~        1 PROB_Lake Target_S~ Targ~   37.4 -108.  CO    Region~
##  7 NLA066~        2 PROB_Lake Target_S~ Targ~   37.4 -108.  CO    Region~
##  8 NLA066~        1 PROB_Lake Target_S~ Targ~   43.9 -115.  ID    Region~
##  9 NLA066~        2 PROB_Lake Target_S~ Targ~   43.9 -115.  ID    Region~
## 10 NLA066~        1 PROB_Lake Target_S~ Targ~   41.7  -73.1 CT    Region~
## # ... with 1,242 more rows, and 42 more variables: AREA_CAT7 <chr>,
## #   NESLAKE <chr>, STRATUM <chr>, PANEL <chr>, DSGN_CAT <chr>,
## #   MDCATY <dbl>, WGT <dbl>, WGT_NLA <dbl>, ADJWGT_CAT <chr>, URBAN <chr>,
## #   WSA_ECO3 <chr>, WSA_ECO9 <chr>, ECO_LEV_3 <dbl>, NUT_REG <chr>,
## #   NUTREG_NAME <chr>, ECO_NUTA <chr>, LAKE_ORIGIN <chr>,
## #   ECO3_X_ORIGIN <chr>, REF_CLUSTER <chr>, RT_NLA <chr>, HUC_2 <dbl>,
## #   HUC_8 <dbl>, FLAG_INFO <chr>, COMMENT_INFO <chr>, SAMPLED <chr>,
## #   SAMPLED_CHEM <chr>, INDXSAMP_CHEM <chr>, PTL <dbl>, NTL <dbl>,
## #   TURB <dbl>, ANC <dbl>, DOC <dbl>, COND <dbl>, SAMPLED_CHLA <chr>,
## #   INDXSAMP_CHLA <chr>, CHLA <dbl>, PTL_COND <chr>, NTL_COND <chr>,
## #   CHLA_COND <chr>, TURB_COND <chr>, ANC_COND <chr>, SALINITY_COND <chr>
```

### Other ways to read in data

There are many ways to read in data with R. If you have questions about this, please let Jeff know. He's happy to chat more about it. Before we move on though, I will show an example of one other way we can do this. Since Excel spreadsheets are so ubiquitous we need a reliable way to read in data stored in an excel spreadsheet. There are a variety of packages that provide this capability, but by far the best (IMHO) is `readxl` which is part of the Tidyverse. This is how we read in an File:

```r
# You'll very likely need to install it first!!!  How'd we do that?
library(readxl)
nla_wq_excel <- read_excel("nla2007_wq.xlsx")
```

This is the simplest case, but lets dig into the options to see what's possible

```
## function (path, sheet = NULL, range = NULL, col_names = TRUE,
##     col_types = NULL, na = "", trim_ws = TRUE, skip = 0, n_max = Inf,
##     guess_max = min(1000, n_max), progress = readxl_progress(),
##     .name_repair = "unique")
## NULL
```

#### An aside on colum names

If you are new to R and coming from mostly and Excel background, then you may want to think a bit more about column names than you usually might. Excel is very flexible when it comes to naming columns and this certainly has its advantages when the end user of that data is a human. However, humans don't do data analysis. Computers do. So at some point the data in that spreadsheet will likely need to be read into software that can do this analysis. To ease this process it is best to keep column names simple, without spaces, and without special characters (e.g. !, @, &, $, etc.). While it is possible to deal with these cases, it is not straightforward, especially for new users. So, when working with your data (or other people's data) take a close look at the column names if you are running into problems reading that data into R. I suggest using all lower case with separate words indicated by and underscore. Things like "chlorophyll_a" or "total_nitrogen" are good examples of decent column names.

## Exercise 3.1

For this exercise, let's read in a new dataset but this time, directly from a URL. We are still working on the `nla_analysis.R` Script

1. Add a new line of code, starting after the `read_csv` line we looked at above (on or around line 39).
2. Use the `read_csv()` function to read in "https://www.epa.gov/sites/production/files/2014-01/nla2007_sampledlakeinformation_20091113.csv", and assign the output to a data frame named `nla_sites`.
3. How many rows and columns do we have in this data frame?
4. What is stored in the fourth column of this data frame?

## Tidy data

We have learned about data frames, how to create them, and about several ways to read in external data into a data.frame. At this point there have been only a few rules applied to our data frames (which already separates them from spreadsheets) and that is our datasets must be rectangular. Beyond that we haven't disscussed how best to organize that data so that subsequent analyses are easier to accomplish. This is, in my opinion, the biggest decision we make as data analysts and it takes a lot of time to think about how best to organize data and to actually re-organize that data. Luckily, we can use an existing concept for this that will help guide our decisions and re-organization. The best concept I know of to do this is the concept of tidy data. The essence of which can be summed up as:

1. Each column is a variable
2. Each row is an observation
3. Each cell has a single value
4. The data must be rectangular

Lastly, if you want to read more about this there are several good sources:

- The previously linked R4DS Chapter on tidy data
- The original paper by Hadley Wickham
- The Tidy Data Vignette
- Really anything on the Tidyverse page
- A lot of what is in the Data Carpentry Ecology Spreadsheet Lesson is also very relevant.

Let's now see some of the basic tools for tidying data using the `tidyr` and `dplyr` packages.

### Data manipulation with `dplyr`

There are a lot of different ways to manipulate data in R, but one that is part of the core of the Tidyverse is `dplyr`. In particular, we are going to look at selecting columns, filtering data, adding new columns, grouping data, and summarizing data.

#### select

Often we get datasets that have many columns or columns that need to be re-ordered. We can accomplish both of these with `select`. Here's a quick example with the `iris` dataset. We will also be introducing the concept of the pipe: `%>%` which we will be using going forward. Let's look at some code that we can disect.

```
iris_petals <- iris %>%
  select(Species, Petal.Width, Petal.Length) %>%
  as_tibble() #the as_tibble function helps make the output look nice
iris_petals
```

```
## # A tibble: 150 x 3
##    Species Petal.Width Petal.Length
##    <fct>         <dbl>        <dbl>
##  1 setosa          0.2          1.4
##  2 setosa          0.2          1.4
##  3 setosa          0.2          1.3
##  4 setosa          0.2          1.5
##  5 setosa          0.2          1.4
##  6 setosa          0.4          1.7
##  7 setosa          0.3          1.4
##  8 setosa          0.2          1.5
##  9 setosa          0.2          1.4
## 10 setosa          0.1          1.5
## # ... with 140 more rows
```

The end result of this is a data frame, `iris_petals` that has three columns: Species, Petal.Width and Petal.Length in the order that we specified. And the syntax we are now using is "piped" in that we use the `%>%` operator to send something from before the operator (a.k.a. "to the left") to the first argument of the function after the operator (a.k.a. "to the right"). This allows us to write our code in the same order as we think of it. The best explanation of this is (again) from R For Data Science in the Piping chapter.

#### filter

The `filter()` function allows us to fiter our data that meets certain criteria. For instance, we might want to further manipulate our 3 column data frame with only one species of Iris and Petals greater than the median petal width.

```
iris_petals_virginica <- iris %>%
  select(species = Species, petal_width = Petal.Width, petal_length = Petal.Length) %>%
  filter(species == "virginica") %>%
  filter(petal_width >= median(petal_width)) %>%
  as_tibble()
iris_petals_virginica
```

```
## # A tibble: 29 x 3
##    species   petal_width petal_length
##    <fct>           <dbl>        <dbl>
##  1 virginica         2.5          6
##  2 virginica         2.1          5.9
##  3 virginica         2.2          5.8
##  4 virginica         2.1          6.6
##  5 virginica         2.5          6.1
##  6 virginica         2            5.1
##  7 virginica         2.1          5.5
##  8 virginica         2            5
##  9 virginica         2.4          5.1
## 10 virginica         2.3          5.3
## # ... with 19 more rows
```

**mutate**

Now say we have some research that suggest the ratio of the petal width and petal length is imporant. We might want to add that as a new column in our data set. The `mutate` function does this for us.

```
iris_petals_ratio <- iris %>%
  select(species = Species, petal_width = Petal.Width, petal_length = Petal.Length) %>%
  mutate(petal_ratio = petal_width/petal_length) %>%
  as_tibble()
iris_petals_ratio
```

```
## # A tibble: 150 x 4
##    species petal_width petal_length petal_ratio
##    <fct>         <dbl>        <dbl>       <dbl>
##  1 setosa          0.2          1.4       0.143
##  2 setosa          0.2          1.4       0.143
##  3 setosa          0.2          1.3       0.154
##  4 setosa          0.2          1.5       0.133
##  5 setosa          0.2          1.4       0.143
##  6 setosa          0.4          1.7       0.235
##  7 setosa          0.3          1.4       0.214
##  8 setosa          0.2          1.5       0.133
##  9 setosa          0.2          1.4       0.143
## 10 setosa          0.1          1.5       0.0667
## # ... with 140 more rows
```

## Exercise 3.2

For this exercise we will dig into our datasets and find ways to tidy them up. We first need to clean up the new data frame,`nla_sites`, that we loaded up in Exercise 3.1. Add new lines of code after the section of code that cleans up the `nla_wq` data frame. Add some comments to your script that describe what we are doing.

1. Filter out just the first visits (e.g. VISIT_NO equal to 1)
2. Select the following columns: SITE_ID, STATE_NAME, and CNTYNAME
3. Make all of our columns names lower case (Hint: Take a look at the code in nla_analysis.R where we manipulate our data)
4. Make all the character fields lower case (Hint: Take a look at the code in nla_analysis.R where we manipulate our data)
5. Keep all these changes in the `nla_sites` data frame.

### group_by and summarize

Now back to iris. What if we want to get some summary statistics of our important petal ratio metric for each of the species? Grouping the data by species, and then summarizing those groupings will let us accomplish this.

```
iris_petal_ratio_species <- iris %>%
  select(species = Species, petal_width = Petal.Width, petal_length = Petal.Length) %>%
  mutate(petal_ratio = petal_width/petal_length) %>%
  group_by(species) %>%
  summarize(mean_petal_ratio = mean(petal_ratio),
            sd_petal_ratio = sd(petal_ratio),
            median_petal_ratio = median(petal_ratio))
iris_petal_ratio_species
```

```
## # A tibble: 3 x 4
##   species    mean_petal_ratio sd_petal_ratio median_petal_ratio
##   <fct>                 <dbl>          <dbl>              <dbl>
## 1 setosa                0.168         0.0658              0.143
## 2 versicolor            0.311         0.0292              0.309
## 3 virginica             0.367         0.0502              0.375
```

### left_join

Lastly, we might also have information spread across multiple data frames. This is the same concept as having multiple tables in a relational database. There are MANY ways to combine (aka. "join) tables like this and most of them have a `dplyr` verb implemented for them. We are going to focus on one, the `left_join()`.

Instead of continuing with the `iris` data we will create some data frames to work with for these examples.

```
left_table <- data.frame(left_id = 1:6,
                         names = c("Bob", "Sue", "Jeff", "Alice", "Joe", "Betty"))
right_table <- data.frame(right_id = 1:5,
                          left_id = c(2,1,3,6,7),
                          age = c(17,26,45,32,6))
left_table
```

```
##   left_id names
## 1       1   Bob
## 2       2   Sue
## 3       3  Jeff
## 4       4 Alice
## 5       5   Joe
## 6       6 Betty
```

```
right_table
```

```
##   right_id left_id age
```

```
## 1        1       2  17
## 2        2       1  26
## 3        3       3  45
## 4        4       6  32
## 5        5       7   6
```

To combine these two tables into one we can `join` them. In particular we will use a `left_join()` which keeps all records from the first table (i.e the "left" table) and adds only the matching records in the second table (i.e. the "right" table). This is easier to grok by looking at the results.

```
left_right_table <- left_table %>%
  left_join(right_table, by = c("left_id" = "left_id"))
left_right_table
```

```
##   left_id names right_id age
## 1       1   Bob        2  26
## 2       2   Sue        1  17
## 3       3  Jeff        3  45
## 4       4 Alice       NA  NA
## 5       5   Joe       NA  NA
## 6       6 Betty        4  32
```

## Exercise 3.3

Let's now practice combining two data frames and summarizing some information in that combine data frame. We are still working on the `nla_analysis.R` script and you can add this code after that section we just completed on `nla_sites`. Don't forget your comments!

1. Use `left_join()` to combine `nla_wq` and `nla_sites` into a new data frame called `nla_2007`
2. Using `group_by()` and `summarize`, let's look at median chlorophyll per EPA Region. (Hint: There are some NA's that we will need to deal with. Use `?median` and try to figure out how to remove those when doing this calculation)
3. Re-do the above and include the minimum and maximum values.
4. Bonus: Use 'arrange() to order the output by mean chlorophyll.