Chapter 7: Iteration

- One major reason to use programming languages is to automate the process of doing iterative operations.
- It would be very tedious to re-run blocks of code many times or to run similar blocks of code where very little changes in the code from one run to the next.
- As an example, with the baseball data set, we might want to do basically the same operation on the data for every season, or to ask the same questions about each team in turn.
- *Iteration* allows us to write a small bit of code and then apply that code repeatedly across teams, seasons, players, etc.
- In general, we can iterate the operation across members of any subgroup of interest, or we can repeat the same code many times, making random changes to the operation at each run.

Vectorized Operations

- To perform the same operation repeatedly on all the different elements of a vector or on all rows (or columns) of a matrix or data frame, a natural programming approach is to use a *loop*.
- In virtually every programming language, there is a capability of writing loops.
- R is no different: Both `for` loop and `while` loop constructions are available.
- However, since R naturally stores elements as vectors, in R loops are much less efficient than *vectorized operations*.
- *Vectorized operations* take a vector as input, perform some operation on every element in the vector, and return a vector as the output.
- Similar operations can be performed on multidimensional arrays like matrices.

Iteration using `map` functions

- Another way to iterate it to apply an R function to each element of a vector (or each row or column of a matrix) using one of the `map` functions from the tidyverse.
- The result is the collection of outputs from this repeated application of the function.
- With `map`, the resulting collection of outputs is stored as a list.
- With `map_dbl`, the result is a numeric (double) vector.
- `map_lgl`, `map_int`, `map_chr` produce logical, integer, and character vectors respectively.
- The base R analogue to the `map` family of functions is the `apply` family of functions (which include `lapply`, `tapply`, etc.).

Speed Advantage of Vectorized Operations

- When vectorized operations are possible, using vectorized operations will typically be faster computationally than using iteration and loops.
- See the example comparing computation time in R.
- In general, try to avoid writing loops in R if there is another way to perform iterative operations.

Automated Iteration with `across`

- In a previous example, we looped over columns 15 to 40 to calculate a bunch of column means.
- Writing programs with such *magic numbers* like 15 and 40 here could make the code less reproducible.
- For example, if new columns are added to the data frame, or the operation is employed on another data frame, those numbers 15 and 40 might be meaningless.
- The `across` adverb can apply verbs like `summarize` and `mutate` to sets of variables that can be specified programmatically rather than via magic numbers.

Using Predicate Functions

- Columns 15 through 40 were the numeric columns in the `Teams` data frame, which is why we focused on those when computing column means.
- We can pick out the numeric columns in `across` programmatically using the *predicate function* `is.numeric` along with `where`.
- When convenient, we could also specify the names of relevant columns in `across`, or specify consecutive columns with the colon symbol `:` in between two column names.
- See the examples from the textbook on the baseball data set.

Iterating over a Single Vector

- The `map` function will apply a known function (either a built-in function or a user-written function) to each element of a vector.
- Note `map` will return the result as a list.
- The more specific variants `map_dbl`, `map_lgl`, `map_int`, `map_chr` return a vector of the specified type.
- `map_dfr` collects the results into a data frame.
- See examples on the Angels baseball franchise.

Iteration over Subgroups

- Recall the five data wrangling verbs from Chapter 4 will operate on a data frame and return a data frame.
- The `group_modify` function in the `purrr` package will apply such a verb (or any function) to *subgroups* of a data frame.
- The grouping is defined via the `group_by` function.
- Like `map_dfr`, the `group_modify` function takes a data frame and returns a data frame.
- While `map_dfr` iterates over individual elements of a vector, `group_modify` iterates over groups (subsets) of the input data frame.

Baseball Data Examples

- Recall the Pythagorean expected winning percentage formula:

$$\widehat{WPct} = \frac{1}{1 + (RA/RS)^2}$$

- Bill James suggested the exponent of 2, which gives a formula that predicts actual winning percentage fairly well.
- Would a different exponent (call it k) give a formula that fit the observed winning percentages better?
- The `nls` function can perform *nonlinear regression* and estimate the optimal value of k for a set of data.
- We can define a function to estimate this k and use `group_modify` to apply the function to various subsets (say, decades) of the historical baseball data.
- Another data example uses `group_modify` to identify league home run leaders over subsets of the historical data.

Example of associations with BMI

- Section 7.7 of the textbook has an extended example of creating scatterplots to display the association between BMI and numerous other variables in the `NHANES` data set.
- We write our own plotting function `bmi_plot` and use `map` to apply it to various columns.
- This produces a list of plot objects, but to actually see the plots, we use the `wrap_plots` functions in the `patchwork` package.