Attributes, Factors, and Matrices

In this slightly technical lesson, we'll be discussing **attributes**.

- First, we'll begin with a high-level overview of what attributes are.
- Afterward, we'll look at two very common use cases of attributes: *factors* and *matrices*.

Attributes

Every object in R can be associated with its **attributes**, where each attribute of an object has a unique name and takes on some value.

Exercise. Use `attributes()` to view the attributes of the built-in variable `mtcars`. Try modifying some of the attributes through direct assignment – does this work? When does it fail?

Exercise. Using `attr()`, write a function that "doubles" all the names of a named list (*e.g.*, `"colname"` $\rightarrow$ `"colnamecolname"`).

In general, attributes are not preserved upon object modification. However, *names*, *dimensions*, and *class* usually persist. These three attributes, being very important, have dedicated functions to access them (`names()`, `dim()`, and `class()`), which should always be used instead of `attr()`.

Now you know where names are stored!

Factors

Factors are used to represent data that can fall into one of a finite number of categories. Examples of data most naturally encoded as factors include: gender, marital status, race, nationality, and profession.

The *factor* class is built on top of integer vectors. This is the crucial insight about factors, similar to the insight about data frames being built on top of lists, and if you remember a single thing from this section, let it be that.

Factors differ from simple integer vectors in one important way: each factor is associated with a `"levels"` attribute, accessible through `levels()`. Each entry in a factor *must* be equal to one of its levels.

Exercise. Create a factor with `factor()` applied to an arbitrary vector, examine its levels, and try to assign a value to the factor that isn't one of its levels. Add that value to its levels (using `levels()`) and retry the assignment.

Exercise. Can you combine two factors with `c()`? If not, what's a reason why?

Hadley Wickham has the following to say about where factors often show up:

Sometimes when a data frame is read directly from a file, a column you'd thought would produce a numeric vector instead produces a factor. This is caused by a non-numeric value in the column, often a missing value encoded in a special way like `.` or `-`. **To remedy the situation, coerce the vector from a factor to a character vector, and then from a character to a double vector.** (Be sure to check for missing values after this process.) Of course, a much better plan is to discover what caused the problem in the first place and fix that; using the `na.strings` argument to `read.csv()` is often a good place to start.

Unfortunately, most data loading functions in R automatically convert character vectors to factors. This is suboptimal, because there's no way for those functions to know the set of all possible levels or their optimal order. Instead, use the argument `stringsAsFactors = FALSE` to suppress this behaviour, and then manually convert character vectors to factors using your knowledge of the data.[1]

While factors look (and often behave) like character vectors, they are actually integers. Be careful when treating them like strings. Some string methods (like `gsub()` and `grepl()`) will coerce factors to strings, while others (like `nchar()`) will throw an error, and still others (like `c()`) will use the underlying integer values. For this reason, **it's usually best to explicitly convert factors to character vectors if you need string-like behaviour**.

Be sure to pay attention to these details when you load data from external sources.

Advanced R, 2.2.2.3. Define `f1 = factor(letters)`, `f2 = rev(factor(letters))`, and `f3 = factor(letters, levels = rev(letters))`. How do these three factors differ?

Exercise. Suppose you're reading in data from a file that's read into a character file into a variable `fruits`, which ends up being equal to `c("apple", "grapefruit", "NA", "apple", "apple, "-", "grapefruit", "durian")`. Read the `factor()` documentation and figure out how to convert `fruits` into a factor such that the character-encoded missing values, `"NA"` and `"-"`, end up as actual `NA`s.

Exercise. Write a function that takes in an arbitrary *character vector* and returns that vector as a factor, with `NA` included in the levels of the factor if it exists in the original factor.

[1] Wickham also writes: "A global option, `options(stringsAsFactors = FALSE)`, is available to control this behaviour, but I don't recommend using it. Changing a global option may have unexpected consequences when combined with other code (either from packages, or code that you're `source()`ing), and global options make code harder to understand because they increase the number of lines you need to read to understand how a single line of code will behave."

Exercise. Write a function takes in a data frame `df` and converts the first `floor(ncol(df))` columns into factors.

Exercise. Write a function that takes in a data frame and converts every column with at most 5 unique values into a factor. (You may find `unique()` useful.)

Exercise. Write a function that takes in a data frame and, for each factor column, replaces every `NA` with the most common non-`NA` value in the column. Generate a toy dataframe to use to demonstrate that your function works.

Exercise. Write a function that takes in a data frame, with some but not all columns being factors, and expands each factor into a set of *indicator variables* within the data frame. Precisely, for each factor, replace that factor column with a number of *binary indicator variables*, having these properties:

- Every level of the factor, aside from the first level, corresponds to a new binary indicator variable.[2]
- For a particular row, each binary indicator variable takes on the value 1 if the original factor was equal to its corresponding level and 0 otherwise.
- For each binary indicator variable, its name is equal to the following strings concatenated together in order: (1) the name of the original factor, (2) an underscore (`"_"`), and (3) the name of the factor level itself.

You can assume that there are no `NA`s in the input dataframe. Test your code on the data frame `df = mtcars[1:10,]; for (n in c("cyl", "am", "carb")) df[[n]] = factor(df[[n]])`; you should obtain a result with 16 columns.

Exercise. Use `load()` to load `time.dat`, a two-column subset of the data from the National Longitudinal Study of Adolescent Health. (The function will load it into the variable `df`.) Look at the documentation for Wave II: In-Home Questionnaire, Public Use Sample and read about the two questions in the data (check the column names). Write some code to convert each column to numeric values representing "number of hours past 8:00 PM" and plot two histograms, one for each column, overlaid on top of each other.[3]

Matrices

Exercise. Run the following code: `df = data.frame(matrix(1:100, nrow=10)); df[5, 5] = NA; df[6, 6] = NA`. Figure out how `df[is.na(df)]` works. Write and test a function that takes as input a data frame `df` of purely

[2] The reason for not making a binary indicator variable for the first level of the factor is that the state of "factor is equal to its first level" can already be represented by the `length(levels()) - 1` binary indicator variables all being set to 0. As such, if we add in a binary indicator variable for the first level, it will cause problems with collinearity that breaks linear techniques (including regression).

[3] You can use string manipulation or R's inbuilt time classes. Try doing it both ways.

numeric data and a number `k`, returning a vector of every number in `df` divisible by `k`.