7.13: Summary

Obviously, there's no real coherence to this chapter. It's just a grab bag of topics and tricks that can be handy to know about, so the best wrap up I can give here is just to repeat this list:

- Section 7.1. Tabulating data.
- Section 7.2. Transforming or recoding a variable.
- Section 7.3. Some useful mathematical functions.
- Section 7.4. Extracting a subset of a vector.
- Section 7.5. Extracting a subset of a data frame.
- Section 7.6. Sorting, flipping or merging data sets.
- Section 7.7. Reshaping a data frame.
- Section 7.8. Manipulating text.
- Section 7.9. Opening data from different file types.
- Section 7.10. Coercing data from one type to another.
- Section 7.11. Other important data types.
- Section 7.12. Miscellaneous topics.

There are a number of books out there that extend this discussion. A couple of my favourites are Spector (2008) "Data Manipulation with R" and Teetor (2011) "R Cookbook".

References

Spector, P. 2008. *Data Manipulation with R*. New York, NY: Springer.

Teetor, P. 2011. *R Cookbook*. Sebastopol, CA: O'Reilly.

103. The quote comes from *Home is the Hangman*, published in 1975.
104. As usual, you can assign this output to a variable. If you type `speaker.freq <- table(speaker)` at the command prompt R will store the table as a variable. If you then type `class(speaker.freq)` you'll see that the output is actually of class `table` . The key thing to note about a table object is that it's basically a matrix (see Section 7.11.1.
105. It's worth noting that there's also a more powerful function called `recode()` function in the `car` package that I won't discuss in this book but is worth looking into if you're looking for a bit more flexibility.
106. If you've read further into the book, and are re-reading this section, then a good example of this would be someone choosing to do an ANOVA using `age.group3` as the grouping variable, instead of running a regression using `age` as a predictor. There are sometimes good reasons for do this: for instance, if the relationship between `age` and your outcome variable is highly non-linear, and you aren't comfortable with trying to run non-linear regression! However, unless you really do have a good rationale for doing this, it's best not to. It tends to introduce all sorts of other problems (e.g., the data will probably violate the normality assumption), and you can lose a lot of power.
107. The real answer is 0: $10 for a sandwich is a total ripoff so I should go next door and buy noodles.
108. Again, I doubt that's the right "real world" answer. I suspect that most sandwich shops won't allow you to pay off your debts to them in sandwiches. But you get the idea.
109. Actually, that's a bit of a lie: the `log()` function is more flexible than that, and can be used to calculate logarithms in *any* base. The `log()` function has a `base` argument that you can specify, which has a default value of e. Thus `log10(1000)` is actually equivalent to `log(x = 1000, base = 10)` .
110. It's also worth checking out the `match()` function
111. It also works on data frames if you ever feel the need to import all of your variables from the data frame into the workspace. This can be useful at times, though it's not a good idea if you have large data sets or if you're working with multiple data sets at once. In particular, if you do this, never forget that you now have *two* copies of all your variables, one in the workspace and another in the data frame.
112. You can do this yourself using the `make.names()` function. In fact, this is itself a handy thing to know about. For example, if you want to convert the names of the variables in the `speech.by.char` list into valid R variable names, you could use a command like this: `names(speech.by.char) <- make.names(names(speech.by.char))` . However, I won't go into details here.

113. Conveniently, if you type `rownames(df) <- NULL` R will renumber all the rows from scratch. For the `df` data frame, the labels that currently run from 7 to 10 will be changed to go from 1 to 4.
114. Actually, you can make the `subset()` function behave this way by using the optional `drop` argument, but by default `subset()` does not drop, which is probably more sensible and more intuitive to novice users.
115. Specifically, recursive indexing, a handy tool in some contexts but not something that I want to discuss here.
116. Remember, `print()` is generic: see Section 4.11.
117. Note for advanced users: both of these functions are just wrappers to the `matrix()` function, which is pretty flexible in terms of the ability to convert vectors into matrices. Also, while I'm on this topic, I'll briefly mention the fact that if you're a Matlab user and looking for an equivalent of Matlab's `repmat()` function, I'd suggest checking out the `matlab` package which contains R versions of a lot of handy Matlab functions.
118. The function you need for that is called `as.data.frame()` .
119. In truth, I suspect that most of the cases when you can sensibly flip a data frame occur when all of the original variables are measurements of the same type (e.g., all variables are response times), and if so you could easily have chosen to encode your data as a matrix instead of as a data frame. But since people do sometimes prefer to work with data frames, I've written the `tFrame()` function for the sake of convenience. I don't really think it's something that is needed very often.
120. This limitation is deliberate, by the way: if you're getting to the point where you want to do something more complicated, you should probably start learning how to use `reshape()` , `cast()` and `melt()` or some of other the more advanced tools. The `wideToLong()` and `longToWide()` functions are included only to help you out when you're first starting to use R.
121. To be honest, it does bother me a little that the default value of `sep` is a space. Normally when I want to paste strings together I don't want any separator character, so I'd prefer it if the default were `sep=""` . To that end, it's worth noting that there's also a `paste0()` function, which is identical to `paste()` except that it always assumes that `sep=""` . Type `?paste` for more information about this.
122. Note that you can capture the output from `cat()` if you want to, but you have to be sneaky and use the `capture.output()` function. For example, the command `x <- capture.output(cat(hw,ng))` would work just fine.
123. Sigh. For advanced users: R actually supports two different ways of specifying regular expressions. One is the POSIX standard, the other is to use Perl-style regular expressions. The default is generally POSIX. If you understand regular expressions, that probably made sense to you. If not, don't worry. It's not important.
124. I thank Amy Perfors for this example.
125. If you're lucky.
126. You can also use the `matrix()` command itself, but I think the "binding" approach is a little more intuitive.
127. This has some interesting implications for how matrix algebra is implemented in R (which I'll admit I initially found odd), but that's a little beyond the scope of this book. However, since there will be a small proportion of readers that do care, I'll quickly outline the basic thing you need to get used to: when multiplying a matrix by a vector (or one-dimensional array) using the `\%*\%` operator R will attempt to interpret the vector (or 1D array) as either a row-vector or column-vector, depending on whichever one makes the multiplication work. That is, suppose M is the 2×3 matrix, and v is a 1×3 row vector. It is impossible to multiply Mv, since the dimensions don't conform, but you *can* multiply by the corresponding column vector, Mvt. So, if I set `v <- M[2,]` and then try to calculate `M \%*\% v` , which you'd think would fail, it actually works because R treats the one dimensional array as if it were a column vector for the purposes of matrix multiplication. Note that if both objects are one dimensional arrays/vectors, this leads to ambiguity since vvt (inner product) and vtv (outer product) yield different answers. In this situation, the `\%*\%` operator returns the inner product not the outer product. To understand all the details, check out the help documentation.
128. I should note that if you type `class(xtab.3d)` you'll discover that this is a `"table"` object rather than an `"array"` object. However, this labelling is only skin deep. The underlying data structure here is actually an array. Advanced users may wish to check this using the command `class(unclass(xtab.3d))` , but it's not important for our purposes. All I really want to do in this section is show you what the output looks like when you encounter a 3D array.
129. Date objects are coded as the number of days that have passed since January 1, 1970.
130. For advanced users: type `?double` for more information.
131. Or at least, that's the default. If all your numbers are integers (whole numbers), then you can explicitly tell R to store them as integers by adding an `L` suffix at the end of the number. That is, an assignment like `x <- 2L` tells R to assign `x` a value of 2, and to store it as an integer rather than as a binary expansion. Type `?integer` for more details.

132. For advanced users: that's a little over simplistic in two respects. First, it's a terribly imprecise way of talking about scoping. Second, it might give you the impression that all the variables in question are actually loaded into memory. That's not quite true, since that would be very wasteful of memory. Instead R has a "lazy loading" mechanism, in which what R actually does is create a "promise" to load those objects if they're actually needed. For details, check out the `delayedAssign()` function.
