7.11: Other Useful Data Structures

Up to this point we have encountered several different kinds of variables. At the simplest level, we've seen numeric data, logical data and character data. However, we've also encountered some more complicated kinds of variables, namely factors, formulas, data frames and lists. We'll see a few more specialised data structures later on in this book, but there's a few more generic ones that I want to talk about in passing. None of them are central to the rest of the book (and in fact, the only one we'll even see anywhere else is the matrix), but they do crop up a fair bit in real life.

7.11.1 Matrices

In various different places in this chapter I've made reference to an R data structure called a ***matrix***, and mentioned that I'd talk a bit more about matrices later on. That time has come. Much like a data frame, a matrix is basically a big rectangular table of data, and in fact there are quite a few similarities between the two. However, there are also some key differences, so it's important to talk about matrices in a little detail. Let's start by using `rbind()` to create a small matrix:[126]

```
row.1 <- c( 2,3,1 )          # create data for row 1
row.2 <- c( 5,6,7 )          # create data for row 2
M <- rbind( row.1, row.2 )   # row bind them into a matrix
print( M )                   # and print it out...
```

```
##       [,1] [,2] [,3]
## row.1    2    3    1
## row.2    5    6    7
```

The variable `M` is a matrix, which we can confirm by using the `class()` function. Notice that, when we bound the two vectors together, R retained the names of the original variables as row names. We could delete these if we wanted by typing `rownames(M)<-NULL`, but I generally prefer having meaningful names attached to my variables, so I'll keep them. In fact, let's also add some highly unimaginative column names as well:

```
colnames(M) <- c( "col.1", "col.2", "col.3" )
print(M)
```

```
##       col.1 col.2 col.3
## row.1     2     3     1
## row.2     5     6     7
```

You can use square brackets to subset a matrix in much the same way that you can for data frames, again specifying a row index and then a column index. For instance, `M[2,3]` pulls out the entry in the 2nd row and 3rd column of the matrix (i.e., `7`), whereas `M[2,]` pulls out the entire 2nd row, and `M[,3]` pulls out the entire 3rd column. However, it's worth noting that when you pull out a column, R will print the results horizontally, not vertically. The reason for this relates to how matrices (and arrays generally) are implemented. The original matrix `M` is treated as a two-dimensional objects, containing 2 rows and 3 columns. However, whenever you pull out a single row or a single column, the result is considered to be one-dimensional. As far as R is concerned there's no real reason to distinguish between a one-dimensional object printed vertically (a column) and a one-dimensional object printed horizontally (a row), and it prints them all out horizontally.[127] There is also a way of using only a single index, but due to the internal structure to how R defines a matrix, it works very differently to what we saw previously with data frames.

The single-index approach is illustrated in Table 7.5 but I don't really want to focus on it since we'll never really need it for this book, and matrices don't play anywhere near as large a role in this book as data frames do. The reason for these differences is that for this is that, for both data frames and matrices, the "row and column" version exists to allow the human user to interact with the object in the psychologically meaningful way: since both data frames and matrices are basically just tables of data, it's the same in

each case. However, the single-index version is really a method for you to interact with the object in terms of its internal structure, and the internals for data frames and matrices are quite different.

Table 7.5: The row and column version, which is identical to the corresponding indexing scheme for a data frame of the same size.

Row	Col.1	Col.2	Col.3
Row 1	[1,1]	[1,2]	[1,3]
Row 2	[2,1]	[2,2]	[2,3]

Table 7.5: The single-index version, which is quite different to what we would get with a data frame.

Row	Col.1	Col.2	Col.3
Row 1	1	3	5
Row 2	2	4	6

The critical difference between a data frame and a matrix is that, in a data frame, we have this notion that each of the columns corresponds to a different variable: as a consequence, the columns in a data frame can be of different data types. The first column could be numeric, and the second column could contain character strings, and the third column could be logical data. In that sense, there is a fundamental asymmetry build into a data frame, because of the fact that columns represent variables (which can be qualitatively different to each other) and rows represent cases (which cannot). Matrices are intended to be thought of in a different way. At a fundamental level, a matrix really is just *one* variable: it just happens that this one variable is formatted into rows and columns. If you want a matrix of numeric data, every single element in the matrix *must* be a number. If you want a matrix of character strings, every single element in the matrix *must* be a character string. If you try to mix data of different types together, then R will either spit out an error, or quietly coerce the underlying data into a list. If you want to find out what class R secretly thinks the data within the matrix is, you need to do something like this:

```
class( M[1] )
```

```
## [1] "numeric"
```

You can't type `class(M)`, because all that will happen is R will tell you that `M` is a matrix: we're not interested in the class of the matrix itself, we want to know what class the underlying data is assumed to be. Anyway, to give you a sense of how R enforces this, let's try to change one of the elements of our numeric matrix into a character string:

```
M[1,2] <- "text"
M
```

```
##       col.1 col.2  col.3
## row.1 "2"   "text" "1"
## row.2 "5"   "6"    "7"
```

It looks as if R has coerced all of the data in our matrix into character strings. And in fact, if we now typed in `class(M[1])` we'd see that this is exactly what has happened. If you alter the contents of one element in a matrix, R will change the underlying data type as necessary.

There's only one more thing I want to talk about regarding matrices. The concept behind a matrix is very much a mathematical one, and in mathematics a matrix is a most definitely a two-dimensional object. However, when doing data analysis, we often have reasons to want to use higher dimensional tables (e.g., sometimes you need to cross-tabulate three variables against each other). You can't do this with matrices, but you can do it with ***arrays***. An array is just like a matrix, except it can have more than two dimensions if you need it to. In fact, as far as R is concerned a matrix is just a special kind of array, in much the same way that a data frame is a special kind of list. I don't want to talk about arrays too much, but I will very briefly show you an example of what

a 3D array looks like. To that end, let's cross tabulate the `speaker` and `utterance` variables from the `nightgarden.Rdata` data file, but we'll add a third variable to the cross-tabs this time, a logical variable which indicates whether or not I was still awake at this point in the show:

```
dan.awake <- c( TRUE,TRUE,TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,FALSE,FALSE )
```

Now that we've got all three variables in the workspace (assuming you loaded the `nightgarden.Rdata` data earlier in the chapter) we can construct our three way cross-tabulation, using the `table()` function.

```
xtab.3d <- table( speaker, utterance, dan.awake )
xtab.3d
```

```
## , , dan.awake = FALSE
##
##              utterance
## speaker       ee onk oo pip
##   makka-pakka  0   2  0   2
##   tombliboo    0   0  1   0
##   upsy-daisy   0   0  0   0
##
## , , dan.awake = TRUE
##
##              utterance
## speaker       ee onk oo pip
##   makka-pakka  0   0  0   0
##   tombliboo    1   0  0   0
##   upsy-daisy   0   2  0   2
```

Hopefully this output is fairly straightforward: because R can't print out text in three dimensions, what it does is show a sequence of 2D slices through the 3D table. That is, the `, , dan.awake = FALSE` part indicates that the 2D table that follows below shows the 2D cross-tabulation of `speaker` against `utterance` only for the `dan.awake = FALSE` instances, and so on.[128]

7.11.2 Ordered factors

One topic that I neglected to mention when discussing factors previously (Section 4.7 is that there are actually two different types of factor in R, unordered factors and ordered factors. An unordered factor corresponds to a nominal scale variable, and all of the factors we've discussed so far in this book have been unordered (as will all the factors used anywhere else except in this section). However, it's often very useful to explicitly tell R that your variable is *ordinal scale*, and if so you need to declare it to be an ***ordered factor***. For instance, earlier in this chapter we made use of a variable consisting of Likert scale data, which we represented as the `likert.raw` variable:

```
likert.raw
```

```
##  [1] 1 7 3 4 4 4 2 6 5 5
```

We can declare this to be an ordered factor in by using the `factor()` function, and setting `ordered = TRUE` . To illustrate how this works, let's create an ordered factor called `likert.ordinal` and have a look at it:

```
likert.ordinal <- factor( x = likert.raw,        # the raw data
                          levels = seq(7,1,-1),  # strongest agreement is 1, weakest
                          ordered = TRUE )       # and it's ordered
print( likert.ordinal )
```

```
##  [1] 1 7 3 4 4 4 2 6 5 5
## Levels: 7 < 6 < 5 < 4 < 3 < 2 < 1
```

Notice that when we print out the ordered factor, R explicitly tells us what order the levels come in. Because I wanted to order my levels in terms of *increasing* strength of agreement, and because a response of 1 corresponded to the strongest agreement and 7 to the strongest disagreement, it was important that I tell R to encode 7 as the lowest value and 1 as the largest. Always check this when creating an ordered factor: it's very easy to accidentally encode your data "upside down" if you're not paying attention. In any case, note that we can (and should) attach meaningful names to these factor levels by using the `levels()` function, like this:

```
levels( likert.ordinal ) <- c( "strong.disagree", "disagree", "weak.disagree",
                               "neutral", "weak.agree", "agree", "strong.agree" )
print( likert.ordinal )
```

```
##  [1] strong.agree    strong.disagree weak.agree      neutral
##  [5] neutral         neutral         agree           disagree
##  [9] weak.disagree   weak.disagree
## 7 Levels: strong.disagree < disagree < weak.disagree < ... < strong.agree
```

One nice thing about using ordered factors is that there are a lot of analyses for which R automatically treats ordered factors differently from unordered factors, and generally in a way that is more appropriate for ordinal data. However, since I don't discuss that in this book, I won't go into details. Like so many things in this chapter, my main goal here is to make you aware that R has this capability built into it; so if you ever need to start thinking about ordinal scale variables in more detail, you have at least some idea where to start looking!

7.11.3 Dates and times

Times and dates are very annoying types of data. To a first approximation we can say that there are 365 days in a year, 24 hours in a day, 60 minutes in an hour and 60 seconds in a minute, but that's not quite correct. The length of the solar day is not exactly 24 hours, and the length of solar year is not exactly 365 days, so we have a complicated system of corrections that have to be made to keep the time and date system working. On top of that, the measurement of time is usually taken relative to a local time zone, and most (but not all) time zones have both a standard time and a daylight savings time, though the date at which the switch occurs is not at all standardised. So, as a form of data, times and dates *suck*. Unfortunately, they're also important. Sometimes it's possible to avoid having to use any complicated system for dealing with times and dates. Often you just want to know what year something happened in, so you can just use numeric data: in quite a lot of situations something as simple as `this.year <- 2011` works just fine. If you can get away with that for your application, this is probably the best thing to do. However, sometimes you really do need to know the actual date. Or, even worse, the actual time. In this section, I'll very briefly introduce you to the basics of how R deals with date and time data. As with a lot of things in this chapter, I won't go into details because I don't use this kind of data anywhere else in the book. The goal here is to show you the basics of what you need to do if you ever encounter this kind of data in real life. And then we'll all agree never to speak of it again.

To start with, let's talk about the date. As it happens, modern operating systems are very good at keeping track of the time and date, and can even handle all those annoying timezone issues and daylight savings pretty well. So R takes the quite sensible view that it can just ask the operating system what the date is. We can pull the date using the `Sys.Date()` function:

```
today <- Sys.Date()  # ask the operating system for the date
print(today)         # display the date
```

```
## [1] "2018-12-30"
```

Okay, that seems straightforward. But, it does rather look like `today` is just a character string, doesn't it? That would be a problem, because dates really do have a numeric character to them, and it would be nice to be able to do basic addition and subtraction to them. Well, fear not. If you type in `class(today)`, R will tell you that the class of the `today` variable is `"Date"`. What this means is that, hidden underneath this text string that prints out an actual date, R actually has a numeric representation.[129] What that means is that you actually can add and subtract days. For instance, if we add 1 to `today`, R will print out the date for tomorrow:

```
today + 1
```

```
## [1] "2018-12-31"
```

Let's see what happens when we add 365 days:

```
today + 365
```

```
## [1] "2019-12-30"
```

This is particularly handy if you forget that a year is a leap year since in that case you'd probably get it wrong is doing this in your head. R provides a number of functions for working with dates, but I don't want to talk about them in any detail. I will, however, make passing mention of the `weekdays()` function which will tell you what day of the week a particular date corresponded to, which is extremely convenient in some situations:

```
weekdays( today )
```

```
## [1] "Sunday"
```

I'll also point out that you can use the `as.Date()` to convert various different kinds of data into dates. If the data happen to be strings formatted exactly according to the international standard notation (i.e., `yyyy-mm-dd`) then the conversion is straightforward, because that's the format that R expects to see by default. You can convert dates from other formats too, but it's slightly trickier, and beyond the scope of this book.

What about times? Well, times are even more annoying, so much so that I don't intend to talk about them at all in this book, other than to point you in the direction of some vaguely useful things. R itself does provide you with some tools for handling time data, and in fact there are two separate classes of data that are used to represent times, known by the odd names `POSIXct` and `POSIXlt`. You can use these to work with times if you want to, but for most applications you would probably be better off downloading the `chron` package, which provides some much more user friendly tools for working with times and dates.