7.6: Sorting, Flipping and Merging Data

In this section I discuss a few useful operations that I feel are loosely related to one another: sorting a vector, sorting a data frame, binding two or more vectors together into a data frame (or matrix), and flipping a data frame (or matrix) on its side. They're all fairly straightforward tasks, at least in comparison to some of the more obnoxious data handling problems that turn up in real life.

7.6.1 Sorting a numeric or character vector

One thing that you often want to do is sort a variable. If it's a numeric variable you might want to sort in increasing or decreasing order. If it's a character vector you might want to sort alphabetically, etc. The `sort()` function provides this capability.

```
numbers <- c(2,4,3)
sort( x = numbers )
```

```
## [1] 2 3 4
```

You can ask for R to sort in decreasing order rather than increasing:

```
sort( x = numbers, decreasing = TRUE )
```

```
## [1] 4 3 2
```

And you can ask it to sort text data in alphabetical order:

```
text <- c("aardvark", "zebra", "swing")
sort( text )
```

```
## [1] "aardvark" "swing"    "zebra"
```

That's pretty straightforward. That being said, it's important to note that I'm glossing over something here. When you apply `sort()` to a character vector it doesn't strictly sort into alphabetical order. R actually has a slightly different notion of how characters are ordered (see Section 7.8.5 and Table 7.3), which is more closely related to how computers store text data than to how letters are ordered in the alphabet. However, that's a topic we'll discuss later. For now, the only thing I should note is that the `sort()` function doesn't alter the original variable. Rather, it creates a new, sorted variable as the output. So if I inspect my original `text` variable:

```
text
```

```
## [1] "aardvark" "zebra"    "swing"
```

I can see that it has remained unchanged.

7.6.2 Sorting a factor

You can also sort factors, but the story here is slightly more subtle because there's two different ways you can sort a factor: alphabetically (by label) or by factor level. The `sort()` function uses the latter. To illustrate, let's look at the two different examples. First, let's create a factor in the usual way:

```
fac <- factor( text )
fac
```

```
## [1] aardvark zebra    swing
## Levels: aardvark swing zebra
```

Now let's sort it:

```
sort(fac)
```

```
## [1] aardvark swing    zebra
## Levels: aardvark swing zebra
```

This *looks* like it's sorted things into alphabetical order, but that's only because the factor levels themselves happen to be alphabetically ordered. Suppose I deliberately define the factor levels in a non-alphabetical order:

```
fac <- factor( text, levels = c("zebra","swing","aardvark") )
fac
```

```
## [1] aardvark zebra    swing
## Levels: zebra swing aardvark
```

Now what happens when we try to sort `fac` this time? The answer:

```
sort(fac)
```

```
## [1] zebra    swing    aardvark
## Levels: zebra swing aardvark
```

It sorts the data into the numerical order implied by the factor levels, not the alphabetical order implied by the labels attached to those levels. Normally you never notice the distinction, because by default the factor levels are assigned in alphabetical order, but it's important to know the difference:

7.6.3 Sorting a data frame

The `sort()` function doesn't work properly with data frames. If you want to sort a data frame the standard advice that you'll find online is to use the `order()` function (not described in this book) to determine what order the rows should be sorted, and then use square brackets to do the shuffling. There's nothing inherently wrong with this advice, I just find it tedious. To that end, the `lsr` package includes a function called `sortFrame()` that you can use to do the sorting. The first argument to the function is named (`x`), and should correspond to the data frame that you want sorted. After that, all you do is type a list of the names of the variables that you want to use to do the sorting. For instance, if I type this:

```
sortFrame( garden, speaker, line)
```

```
##           speaker utterance line
## case.4 makka-pakka       pip    7
## case.5 makka-pakka       onk    9
## case.3   tombliboo        ee    5
## case.1  upsy-daisy       pip    1
## case.2  upsy-daisy       pip    2
```

what R does is first sort by `speaker` (factor level order). Any ties (i.e., data from the same speaker) are then sorted in order of `line` (increasing numerical order). You can use the minus sign to indicate that numerical variables should be sorted in reverse

order:

```
sortFrame( garden, speaker, -line)
```

```
##            speaker utterance line
## case.5 makka-pakka       onk    9
## case.4 makka-pakka       pip    7
## case.3   tombliboo        ee    5
## case.2  upsy-daisy       pip    2
## case.1  upsy-daisy       pip    1
```

As of the current writing, the `sortFrame()` function is under development. I've started introducing functionality to allow you to use the `-` sign to non-numeric variables or to make a distinction between sorting factors alphabetically or by factor level. The idea is that you should be able to type in something like this:

```
sortFrame( garden, -speaker)
```

and have the output correspond to a sort of the `garden` data frame in *reverse* alphabetical order (or reverse factor level order) of `speaker`. As things stand right now, this will actually work, and it will produce sensible output:

```
sortFrame( garden, -speaker)
```

```
##            speaker utterance line
## case.1  upsy-daisy       pip    1
## case.2  upsy-daisy       pip    2
## case.3   tombliboo        ee    5
## case.4 makka-pakka       pip    7
## case.5 makka-pakka       onk    9
```

However, I'm not completely convinced that I've set this up in the ideal fashion, so this may change a little bit in the future.

7.6.4 Binding vectors together

A not-uncommon task that you might find yourself needing to undertake is to combine several vectors. For instance, let's suppose we have the following two numeric vectors:

```
cake.1 <- c(100, 80, 0, 0, 0)
cake.2 <- c(100, 100, 90, 30, 10)
```

The numbers here might represent the amount of each of the two cakes that are left at five different time points. Apparently the first cake is tastier, since that one gets devoured faster. We've already seen one method for combining these vectors: we could use the `data.frame()` function to convert them into a data frame with two variables, like so:

```
cake.df <- data.frame( cake.1, cake.2 )
cake.df
```

```
##   cake.1 cake.2
## 1    100    100
## 2     80    100
## 3      0     90
## 4      0     30
## 5      0     10
```

Two other methods that I want to briefly refer to are the `rbind()` and `cbind()` functions, which will convert the vectors into a matrix. I'll discuss matrices properly in Section 7.11.1 but the details don't matter too much for our current purposes. The `cbind()` function ("column bind") produces a very similar looking output to the data frame example:

```
cake.mat1 <- cbind( cake.1, cake.2 )
cake.mat1
```

```
##      cake.1 cake.2
## [1,]    100    100
## [2,]     80    100
## [3,]      0     90
## [4,]      0     30
## [5,]      0     10
```

but nevertheless it's important to keep in mind that `cake.mat1` is a matrix rather than a data frame, and so has a few differences from the `cake.df` variable. The `rbind()` function ("row bind") produces a somewhat different output: it binds the vectors together row-wise rather than column-wise, so the output now looks like this:

```
cake.mat2 <- rbind( cake.1, cake.2 )
cake.mat2
```

```
##        [,1] [,2] [,3] [,4] [,5]
## cake.1  100   80    0    0    0
## cake.2  100  100   90   30   10
```

You can add names to a matrix by using the `rownames()` and `colnames()` functions, and I should also point out that there's a fancier function in R called `merge()` that supports more complicated "database like" merging of vectors and data frames, but I won't go into details here.

7.6.5 Binding multiple copies of the same vector together

It is sometimes very useful to bind together multiple copies of the same vector. You could do this using the `rbind` and `cbind` functions, using comands like this one

```
fibonacci <- c( 1,1,2,3,5,8 )
rbind( fibonacci, fibonacci, fibonacci )
```

```
##           [,1] [,2] [,3] [,4] [,5] [,6]
## fibonacci    1    1    2    3    5    8
## fibonacci    1    1    2    3    5    8
## fibonacci    1    1    2    3    5    8
```

but that can be pretty annoying, especially if you needs lots of copies. To make this a little easier, the `lsr` package has two additional functions `rowCopy` and `colCopy` that do the same job, but all you have to do is specify the number of copies that

you want, instead of typing the name in over and over again. The two arguments you need to specify are `x`, the vector to be copied, and `times`, indicating how many copies should be created:[117]

```
rowCopy( x = fibonacci, times = 3 )
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    1    2    3    5    8
## [2,]    1    1    2    3    5    8
## [3,]    1    1    2    3    5    8
```

Of course, in practice you don't need to name the arguments all the time. For instance, here's an example using the `colCopy()` function with the argument names omitted:

```
colCopy( fibonacci, 3 )
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    2    2    2
## [4,]    3    3    3
## [5,]    5    5    5
## [6,]    8    8    8
```

7.6.6 Transposing a matrix or data frame

```
load("./rbook-master/data/cakes.Rdata" )
cakes
```

```
##        time.1 time.2 time.3 time.4 time.5
## cake.1    100     80      0      0      0
## cake.2    100    100     90     30     10
## cake.3    100     20     20     20     20
## cake.4    100    100    100    100    100
```

And just to make sure you believe me that this is actually a matrix:

```
class( cakes )
```

```
## [1] "matrix"
```

Okay, now let's transpose the matrix:

```
cakes.flipped <- t( cakes )
cakes.flipped
```

```
##        cake.1 cake.2 cake.3 cake.4
## time.1    100    100    100    100
## time.2     80    100     20    100
## time.3      0     90     20    100
## time.4      0     30     20    100
## time.5      0     10     20    100
```

The output here is still a matrix:

```
class( cakes.flipped )
```

```
## [1] "matrix"
```

At this point you should have two questions: (1) how do we do the same thing for data frames? and (2) why should we care about this? Let's start with the how question. First, I should note that you can transpose a data frame just fine using the `t()` function, but that has the slightly awkward consequence of converting the output from a data frame to a matrix, which isn't usually what you want. It's quite easy to convert the output back again, of course,[118] but I hate typing two commands when I can do it with one. To that end, the `lsr` package has a simple "convenience" function called `tFrame()` which does exactly the same thing as `t()` but converts the output to a data frame for you. To illustrate this, let's transpose the `itng` data frame that we used earlier. Here's the original data frame:

```
itng
```

```
##        speaker utterance
## 1   upsy-daisy       pip
## 2   upsy-daisy       pip
## 3   upsy-daisy       onk
## 4   upsy-daisy       onk
## 5    tombliboo        ee
## 6    tombliboo        oo
## 7  makka-pakka       pip
## 8  makka-pakka       pip
## 9  makka-pakka       onk
## 10 makka-pakka       onk
```

and here's what happens when you transpose it using `tFrame()` :

```
tFrame( itng )
```

```
##                  V1         V2         V3         V4         V5         V6
## speaker   upsy-daisy upsy-daisy upsy-daisy upsy-daisy tombliboo tombliboo
## utterance        pip        pip        onk        onk         ee         oo
##                   V7          V8          V9         V10
## speaker   makka-pakka makka-pakka makka-pakka makka-pakka
## utterance         pip         pip         onk         onk
```

An important point to recognise is that transposing a data frame is not always a sensible thing to do: in fact, I'd go so far as to argue that it's usually *not* sensible. It depends a lot on whether the "cases" from your original data frame would make sense as variables, and to think of each of your original "variables" as cases. I think that's emphatically *not* true for our `itng` data frame, so I wouldn't advise doing it in this situation.

That being said, sometimes it really is true. For instance, had we originally stored our `cakes` variable as a data frame instead of a matrix, then it would absolutely be sensible to flip the data frame![119] There are some situations where it is useful to flip your data frame, so it's nice to know that you can do it. Indeed, that's the main reason why I have spent so much time talking about this topic. A lot of statistical tools make the assumption that the rows of your data frame (or matrix) correspond to observations, and the columns correspond to the variables. That's not unreasonable, of course, since that is a pretty standard convention. However, think about our `cakes` example here. This is a situation where you might want do an analysis of the different cakes (i.e. cakes as variables, time points as cases), but equally you might want to do an analysis where you think of the times as being the things of interest (i.e., times as variables, cakes as cases). If so, then it's useful to know how to flip a matrix or data frame around.
