CHAPTER 6
Data Transformations

Introduction

This chapter is all about the *apply* functions: `apply`, `lapply`, `sapply`, `tapply`, `mapply`; and their cousins, `by` and `split`. These functions let you take data in great gulps and process the whole gulp at once. Where traditional programming languages use loops, R uses vectorized operations and the apply functions to crunch data in batches, greatly streamlining the calculations.

Defining Groups Via a Factor

An important idiom of R is using a factor to define a group. Suppose we have a vector and a factor, both of the same length, that were created as follows:

```
> v <- c(40,2,83,28,58)
> f <- factor(c("A","C","C","B","C"))
```

We can visualize the vector elements and factors levels side by side, like this:

Vector	Factor
40	A
2	C
83	A
28	B
58	C

The factor level identifies the group of each vector element: 40 and 83 are in group A; 28 is in group B; and 2 and 58 are in group C.

In this book, I refer to such factors as *grouping factors*. They effectively slice and dice our data by putting them into groups. This is powerful because processing data in groups occurs often in statistics when comparing group means, comparing group proportions, performing ANOVA analysis, and so forth.

This chapter has recipes that use grouping factors to split vector elements into their respective groups (Recipe 6.1), apply a function to groups within a vector (Recipe 6.5), and apply a function to groups of rows within a data frame (Recipe 6.6). In other chapters, the same idiom is used to test group means (Recipe 9.19), perform one-way ANOVA analysis (Recipe 11.20), and plot data points by groups (Recipe 10.4), among other uses.

6.1 Splitting a Vector into Groups

Problem

You have a vector. Each element belongs to a different group, and the groups are identified by a grouping factor. You want to split the elements into the groups.

Solution

Suppose the vector is `x` and the factor is `f`. You can use the `split` function:

```
> groups <- split(x, f)
```

Alternatively, you can use the `unstack` function:

```
> groups <- unstack(data.frame(x,f))
```

Both functions return a list of vectors, where each vector contains the elements for one group.

The `unstack` function goes one step further: if all vectors have the same length, it converts the list into a data frame.

Discussion

The `Cars93` dataset contains a factor called `Origin` that has two levels, `USA` and `non-USA`. It also contains a column called `MPG.city`. We can split the MPG data according to origin as follows:

```
> library(MASS)
> split(Cars93$MPG.city, Cars93$Origin)
$USA
 [1] 22 19 16 19 16 16 25 25 19 21 18 15 17 17 20 23 20 29 23 22 17 21 18 29 20
[26] 31 23 22 22 24 15 21 18 17 18 23 19 24 23 18 19 23 31 23 19 19 19 28

$non-USA
 [1] 25 18 20 19 22 46 30 24 42 24 29 22 26 20 17 18 18 29 28 26 18 17 20 19 29
[26] 18 29 24 17 21 20 33 25 23 39 32 25 22 18 25 17 21 18 21 20
```

The usefulness of splitting is that we can analyze the data by group. This example computes the median MPG for each group:

```
> g <- split(Cars93$MPG.city, Cars93$Origin)
> median(g[[1]])
[1] 20
> median(g[[2]])
[1] 22
```

See Also

See the "Introduction" to this chapter for more about grouping factors. Recipe 6.5 shows another way to apply functions, such as `median`, to each group. The `unstack` function can perform other, more powerful transformations beside this; see the help page.

6.2 Applying a Function to Each List Element

Problem

You have a list, and you want to apply a function to each element of the list.

Solution

Use either the `lapply` function or the `sapply` function, depending upon the desired form of the result. `lapply` always returns the results in list, whereas `sapply` returns the results in a vector if that is possible:

```
> lst <- lapply(lst, fun)
> vec <- sapply(lst, fun)
```

Discussion

These functions will call your function (*fun*, in the solution example) once for every element on your list. Your function should expect one argument, an element from the list. The `lapply` and `sapply` functions will collect the returned values. `lapply` collects them into a list and returns the list.

The "`s`" in "`sapply`" stands for "simplify." The function tries to simplify the results into a vector or matrix. For that to happen, all the returned values must have the same length. If that length is 1 then you get a vector; otherwise, you get a matrix. If the lengths vary, simplification is impossible and you get a list.

Let's say I teach an introductory statistics class four times and administer comparable final exams each time. Here are the exam scores from the four semesters:

```
> scores
$S1
 [1] 89 85 85 86 88 89 86 82 96 85 93 91 98 87 94 77 87 98 85 89
[21] 95 85 93 93 97 71 97 93 75 68 98 95 79 94 98 95

$S2
 [1]  60  98  94  95  99  97 100  73  93  91  98  86  66  83  77
```

```
[16]  97  91  93  71  91  95 100  72  96  91  76 100  97  99  95
[31]  97  77  94  99  88 100  94  93  86

$S3
 [1] 95 86 90 90 75 83 96 85 83 84 81 98 77 94 84 89 93 99 91 77
[21] 95 90 91 87 85 76 99 99 97 97 97 77 93 96 90 87 97 88

$S4
 [1] 67 93 63 83 87 97 96 92 93 96 87 90 94 90 82 91 85 93 83 90
[21] 87 99 94 88 90 72 81 93 93 94 97 89 96 95 82 97
```

Each semester starts with 40 students but, alas, not everyone makes it to the finish line; hence each semester has a different number of scores. We can count this number with the `length` function: `lapply` will return a list of lengths, and `sapply` will return a vector of lengths:

```
> lapply(scores, length)
$S1
[1] 36

$S2
[1] 39

$S3
[1] 38

$S4
[1] 36

> sapply(scores, length)
S1 S2 S3 S4
36 39 38 36
```

We can see the mean and standard deviation of the scores just as easily:

```
> sapply(scores, mean)
      S1       S2       S3       S4
88.77778 89.79487 89.23684 88.86111
> sapply(scores, sd)
       S1        S2        S3        S4
 7.720515 10.543592  7.178926  8.208542
```

If the called function returns a vector, `sapply` will form the results into a matrix. The `range` function, for example, returns a two-element vector:

```
> sapply(scores, range)
     S1  S2 S3 S4
[1,] 68  60 75 63
[2,] 98 100 99 99
```

If the called function returns a structured object, such as a list, then you will need to use `lapply` rather than `sapply`. Structured objects cannot be put into a vector. Suppose we want to perform a *t* test on every semester. The `t.test` function returns a list, so we must use `lapply`:

```
> tests <- lapply(scores, t.test)
```

Now the result `tests` is a list: it is a list of `t.test` results. We can use `sapply` to extract elements from the `t.test` results, such as the bounds of the confidence interval:

```
> sapply(tests, function(t) t$conf.int)
          S1       S2       S3       S4
[1,] 86.16553 86.37703 86.87719 86.08374
[2,] 91.39002 93.21271 91.59650 91.63848
```

See Also

See Recipe 2.12.

6.3 Applying a Function to Every Row

Problem

You have a matrix. You want to apply a function to every row, calculating the function result for each row.

Solution

Use the `apply` function. Set the second argument to 1 to indicate row-by-row application of a function:

```
> results <- apply(mat, 1, fun)     # mat is a matrix, fun is a function
```

The `apply` function will call `fun` once for each row, assemble the returned values into a vector, and then return that vector.

Discussion

Suppose your matrix `long` is longitudinal data. Each row contains data for one subject, and the columns contain the repeated observations over time:

```
> long
         trial1    trial2     trial3     trial4     trial5
Moe   -1.8501520 -1.406571 -1.0104817 -3.7170704 -0.2804896
Larry  0.9496313  1.346517 -0.1580926  1.6272786  2.4483321
Curly -0.5407272 -1.708678 -0.3480616 -0.2757667 -1.2177024
```

You could calculate the average observation for each subject by applying the `mean` function to the rows. The result is a vector:

```
> apply(long, 1, mean)
       Moe      Larry      Curly
-1.6529530  1.2427334 -0.8181872
```

Note that `apply` uses the `rownames` from your matrix to identify the elements of the resulting vector, which is handy.

The function being called (`fun`, described previously) should expect one argument, a vector, which will be one row from the matrix. The function can return a scalar or a vector. In the vector case, `apply` assembles the results into a matrix. The `range` function returns a vector of two elements, the minimum and the maximum, so applying it to `long` produces a matrix:

```
> apply(long, 1, range)
            Moe      Larry      Curly
[1,] -3.7170704 -0.1580926 -1.7086779
[2,] -0.2804896  2.4483321 -0.2757667
```

You can employ this recipe on data frames as well. It works if the data frame is homogeneous—either all numbers or all character strings. When the data frame has columns of different types, extracting vectors from the rows isn't sensible because vectors must be homogeneous.

6.4 Applying a Function to Every Column

Problem

You have a matrix or data frame, and you want to apply a function to every column.

Solution

For a matrix, use the `apply` function. Set the second argument to 2, which indicates column-by-column application of the function:

```
> results <- apply(mat, 2, fun)
```

For a data frame, use the `lapply` or `sapply` functions. Either one will apply a function to successive columns of your data frame. Their difference is that `lapply` assembles the return values into a list whereas `sapply` assembles them into a vector:

```
> lst <- lapply(dfrm, fun)
> vec <- sapply(dfrm, fun)
```

You can use `apply` on data frames, too, but only if the data frame is homogeneous (i.e., either all numeric values or all character strings).

Discussion

The `apply` function is intended for processing a matrix. In Recipe 6.3 we used `apply` to process the rows of a matrix. This is the same situation, but now we are processing the columns. The second argument of `apply` determines the direction:

- 1 means process row by row.
- 2 means process column by column.

This is more mnemonic than it looks. We speak of matrices in "rows and columns", so rows are first and columns second; 1 and 2, respectively.

A data frame is a more complicated data structure than a matrix, so there are more options. You can simply use `apply`, in which case R will convert your data frame to a matrix and then apply your function. That will work if your data frame contains only numeric data or character data. It probably will not work if you have mixed data types. In that case, R will force all columns to have identical types, likely performing an unwanted conversion as a result.

Fortunately, there is an alternative. Recall that a data frame is a kind of list: it is a list of the columns of the data frame. You can use `lapply` and `sapply` to process the columns, as described in Recipe 6.2:

```
> lst <- lapply(dfrm, fun)          # Returns a list
> vec <- sapply(dfrm, fun)          # Returns a vector
```

The function `fun` should expect one argument: a column from the data frame.

I often use this recipe to check the types of columns in data frames. The `batch` column of this data frame seems to contain numbers:

```
> head(batches)
  batch clinic dosage   shrinkage
1     1     IL      3 -0.11810714
2     3     IL      4 -0.29932107
3     2     IL      4 -0.27651716
4     1     IL      5 -0.18925825
5     2     IL      2 -0.06804804
6     3     NJ      5 -0.38279193
```

But printing the classes of the columns reveals it to be a factor instead:

```
> sapply(batches, class)
    batch    clinic    dosage shrinkage
 "factor"  "factor" "integer" "numeric"
```

A cool example of this recipe is removing low-correlation variables from a set of predictors. Suppose that `resp` is a response variable and `pred` is a data frame of predictor variables. Suppose further that we have too many predictors and therefore want to select the top 10 as measured by correlation with the response.

The first step is to calculate the correlation between each variable and `resp`. In R, that's a one-liner:

```
> cors <- sapply(pred, cor, y=resp)
```

The `sapply` function will call the function `cor` for every column in `pred`. Note that we gave a third argument, `y=resp`, to `sapply`. Any arguments beyond the second one are passed to `cor`. Every time that `sapply` calls `cor`, the first argument will be a column and the second argument will be `y=resp`. With those arguments, the function call will be `cor(`*`column`*`,y=resp)`, which calculates the correlation between the given column and `resp`.

The result from `sapply` is a vector of correlations, one for each column. We use the `rank` function to find the positions of the correlations that have the largest magnitude:

```
> mask <- (rank(-abs(cors)) <= 10)
```

This expression is a comparison (`<=`), so it returns a vector of logical values. It is cleverly constructed so that the top 10 correlations have corresponding `TRUE` values and all others are `FALSE`. Using that vector of logical values, we can select just those columns from the data frame:

```
> best.pred <- pred[,mask]
```

At this point, we can regress `resp` against `best.pred`, knowing that we have chosen the predictors with the highest correlations:

```
> lm(resp ~ best.pred)
```

That's pretty good for four lines of code.

See Also

See Recipes 5.22, 6.2, and 6.3.

6.5 Applying a Function to Groups of Data

Problem

Your data elements occur in groups. You want to process the data by groups—for example, summing by group or averaging by group.

Solution

Create a grouping factor (of the same length as your vector) that identifies the group of each corresponding datum. Then use the `tapply` function, which will apply a function to each group of data:

```
> tapply(x, f, fun)
```

Here, *x* is a vector, *f* is a grouping factor, and *fun* is a function. The function should expect one argument, which is a vector of elements taken from *x* according to their group.

Discussion

Suppose I have a vector with the populations of the 16 largest cities in the greater Chicago metropolitan area, taken from the data frame called `suburbs`:

```
> attach(suburbs)
> pop
 [1] 2853114   90352  171782   94487  102746  106221  147779   76031   70834
[10]   72616   74239   83048   67232   75386   63348   91452
```

We can easily compute sums and averages for all the cities:

```
> sum(pop)
[1] 4240667
> mean(pop)
[1] 265041.7
```

What if we want the sum and average broken out by county? We will need a factor, say `county`, the same length as `pop`, where each level of the factor gives the corresponding county (there are two Lake counties: one in Illinois and one in Indiana) :

```
> county
 [1] Cook     Kenosha  Kane     Kane     Lake(IN) Kendall  DuPage   Cook
 [9] Will     Cook     Cook     Lake(IN) Cook     Cook     Cook     Lake(IL)
Levels: Cook DuPage Kane Kendall Kenosha Lake(IL) Lake(IN) Will
```

Now I can use the `county` factor and `tapply` function to process items in groups. The `tapply` function has three main parameters: the vector of data, the factor that defines the groups, and a function. It will extract each group, apply the function to each group, and return a vector with the collected results. This example shows summing the populations by county:

```
> tapply(pop,county,sum)
    Cook   DuPage     Kane  Kendall  Kenosha Lake(IL) Lake(IN)     Will
 3281966   147779   266269   106221    90352    91452   185794    70834
```

The next example computes average populations by county:

```
> tapply(pop,county,mean)
    Cook   DuPage     Kane  Kendall  Kenosha Lake(IL) Lake(IN)     Will
468852.3 147779.0 133134.5 106221.0  90352.0  91452.0  92897.0  70834.0
```

The function given to `tapply` should expect a single argument: a vector containing all the members of one group. A good example is the `length` function, which takes a vector parameter and returns the vector's length. Use it to count the number of cities in each county:

```
> tapply(pop,county,length)
    Cook   DuPage     Kane  Kendall  Kenosha Lake(IL) Lake(IN)     Will
       7        1        2        1        1        1        2        1
```

In most cases you will use functions that return a scalar and `tapply` will collect the returned scalars into a vector. Your function can return complex objects, too, in which case `tapply` will return them in a list. See the `tapply` help page for details.

See Also

See this chapter's "Introduction" for more about grouping factors.

6.6 Applying a Function to Groups of Rows

Problem

You want to apply a function to groups of rows within a data frame.

Solution

Define a grouping factor—that is, a factor with one level (element) for every row in your data frame—that identifies the data groups.

For each such group of rows, the `by` function puts the rows into a temporary data frame and calls your function with that argument. The `by` function collects the returned values into a list and returns the list:

```
> by(dfrm, fact, fun)
```

Here, *`dfrm`* is the data frame, *`fact`* is the grouping factor, and *`fun`* is a function. The function should expect one argument, a data frame.

Discussion

The advantage of the `by` function is that it calls your function with a data frame, which is useful if your function handles data frames in a special way. For instance, the `print`, `summary`, and `mean` functions perform special processing for data frames.

Suppose you have a data frame from clinical trials, called `trials`, where the dosage was randomized to study its effect:

```
> trials
   sex      pre dose1 dose2     post
1    F 5.931640     2     1 3.162600
2    F 4.496187     1     2 3.293989
3    M 6.161944     1     1 4.446643
4    F 4.322465     2     1 3.334748
5    M 4.153510     1     1 4.429382
.
. (etc.)
.
```

The data includes a factor for the subject's sex, so `by` can split the data according to sex and call `summary` for the two groups. The result is two summaries, one for men and one for women:

```
> by(trials, trials$sex, summary)
trials$sex: F
 sex        pre             dose1           dose2           post
 F:7   Min.   :4.156   Min.   :1.000   Min.   :1.000   Min.   :2.886
 M:0   1st Qu.:4.409   1st Qu.:1.000   1st Qu.:1.000   1st Qu.:3.075
       Median :4.895   Median :1.000   Median :2.000   Median :3.163
       Mean   :5.020   Mean   :1.429   Mean   :1.571   Mean   :3.174
       3rd Qu.:5.668   3rd Qu.:2.000   3rd Qu.:2.000   3rd Qu.:3.314
       Max.   :5.932   Max.   :2.000   Max.   :2.000   Max.   :3.389
```

```
------------------------------------------------------------
trials$sex: M
 sex        pre              dose1            dose2            post
 F:0   Min.   :3.998   Min.   :1.000   Min.   :1.000   Min.   :3.738
 M:9   1st Qu.:4.773   1st Qu.:1.000   1st Qu.:1.000   1st Qu.:3.800
       Median :5.110   Median :2.000   Median :1.000   Median :4.194
       Mean   :5.189   Mean   :1.556   Mean   :1.444   Mean   :4.148
       3rd Qu.:5.828   3rd Qu.:2.000   3rd Qu.:2.000   3rd Qu.:4.429
       Max.   :6.658   Max.   :2.000   Max.   :2.000   Max.   :4.517
```

We can also build linear models of `post` as a function of the dosages, with one model for men and one model for women:

```
> models <- by(trials, trials$sex, function(df) lm(post~pre+dose1+dose2, data=df))
```

Observe that the parameter to our function is a data frame, so we can use it as the `data` argument of `lm`. The result is a two-element list of linear models. When we print the list, we see a model for each sex:

```
> print(models)
trials$sex: F

Call:
lm(formula = post ~ pre + dose1 + dose2, data = df)

Coefficients:
(Intercept)          pre        dose1        dose2
    4.30804     -0.08161     -0.16225     -0.31354

------------------------------------------------------------
trials$sex: M

Call:
lm(formula = post ~ pre + dose1 + dose2, data = df)

Coefficients:
(Intercept)          pre        dose1        dose2
    5.29981     -0.02713     -0.36851     -0.30323
```

We have a list of models, so we can apply the `confint` function to each list element and see the confidence intervals for each model's coefficients:

```
> lapply(models, confint)
$F
                2.5 %      97.5 %
(Intercept)  3.0841733  5.53191431
pre         -0.2950747  0.13184560
dose1       -0.4711773  0.14667409
dose2       -0.6044273 -0.02264593

$M
                2.5 %      97.5 %
(Intercept)  4.8898433  5.70978218
pre         -0.1070276  0.05277108
dose1       -0.4905828 -0.24644057
dose2       -0.4460200 -0.16043211
```

In this case, we see that `pre` is not significant for either model because its confidence interval contains zero; in contrast, `dose1` and `dose2` are significant for both models. The key fact, however, is that the models have significantly different intercepts, alerting us to a potentially different response for men and women.

See Also

See this chapter's "Introduction" for more about grouping factors. See Recipe 6.2 for more about `lapply`.

6.7 Applying a Function to Parallel Vectors or Lists

Problem

You have a function, say `f`, that takes multiple arguments. You want to apply the function element-wise to vectors and obtain a vector result. Unfortunately, the function is not *vectorized*; that is, it works on scalars but not on vectors.

Solution

Use the `mapply` function. It will apply the function `f` to your arguments element-wise:

```
> mapply(f, vec1, vec2, ..., vecN)
```

There must be one vector for each argument expected by `f`. If the vector arguments are of unequal length, the Recycling Rule is applied.

The `mapply` function also works with list arguments:

```
> mapply(f, list1, list2, ..., listN)
```

Discussion

The basic operators of R, such as $x + y$, are vectorized; this means that they compute their result element-by-element and return a vector of results. Also, many R functions are vectorized.

Not all functions are vectorized, however, and those that are not work only on scalars. Using vector arguments produces errors at best and meaningless results at worst. In such cases, the `mapply` function can effectively vectorize the function for you.

Consider the `gcd` function from Recipe 2.12, which takes two arguments:

```
> gcd <- function(a,b) {
+     if (b == 0) return(a)
+     else return(gcd(b, a %% b))
+ }
```

If we apply `gcd` to two vectors, the result is wrong answers and a pile of error messages:

```
> gcd(c(1,2,3), c(9,6,3))
[1] 1 2 0
Warning messages:
1: In if (b == 0) return(a) else return(gcd(b, a%%b)) :
  the condition has length > 1 and only the first element will be used
2: In if (b == 0) return(a) else return(gcd(b, a%%b)) :
  the condition has length > 1 and only the first element will be used
3: In if (b == 0) return(a) else return(gcd(b, a%%b)) :
  the condition has length > 1 and only the first element will be used
```

The function is not vectorized, but we can use `mapply` to vectorize it. This gives the element-wise GCDs between two vectors:

```
> mapply(gcd, c(1,2,3), c(9,6,3))
[1] 1 2 3
```