9

Reshaping Data

R is designed so that individual functions don't have complete flexibility with regard to their inputs. Most functions expect their input data to be arranged in a particular way, and it's the responsibility of the user of the function to make sure that the input data is in an appropriate form. So even after you've read in or created your data, it may be necessary to modify your data to suit a function you need.

The focus of this chapter will be on working with data frames, since that is the form required for the majority of the functions in R.

9.1 Modifying Data Frame Variables

Since data frames are lists, new variables can be created by simply assigning their value to a column that doesn't already exist in the data frame. Since operations in R are vectorized, transformations can be carried out without the need to use loops. For example, consider the `Loblolly` data frame which has variables for `height` and `age` for a number of trees. To create a variable called `logheight` representing the log of the `height` variable, we could use statements like

```
> Loblolly$logheight = log(Loblolly$height)
```

or

```
> Loblolly['logheight'] = log(Loblolly['height'])
```

The system's version of the `Loblolly` data frame will not be changed by these statements, but your local copy of `Loblolly` will have the new `logheight` column.

Two functions are handy to avoid the need of retyping the data frame name in order to access columns of a data frame. The `with` function can be used to evaluate any expression, first looking in a data frame of your choice to

resolve variables. For example, the `logheight` column in the previous example could be created using

```
> with(Loblolly,log(height))
```

In cases where new columns are being added to an existing data frame, the `transform` function can be used. The first argument to `transform` is a data frame, and the remaining arguments define new columns which will be returned along with all the columns of the original data frame. Each new column is defined by a `name=value` pair. So an alternate way of creating the `logheight` column in the `Loblolly` dataset would be

```
> Loblolly = transform(Loblolly,logheight = log(height))
```

Once again, the system version of `Loblolly` is unaltered, but the version in the local workspace will have the new column.

To remove a column from a data frame, set its value to `NULL`. The `subset` function (see Section 6.8) can also be useful in such situations. Negative subscripts, which extract everything except those elements specified in the negative subscripts, can also be used to create a data frame with selected rows or columns removed.

Often a similar operation needs to be performed on several columns of a data frame, with the goal being to overwrite the original versions of the variables. In cases like this, the left-hand side of the assignment statement can consist of multiple columns, as long as the expression on the right-hand side is the same size as implied by the target. For example, to convert the lengths of the four numeric variables in the `iris` dataset to inches from centimeters, we can use `sapply` to operate on all four columns at once, and assign the result back to those same columns:

```
> iris[,-5] = sapply(iris[,-5],function(x)x/2.54)
```

9.2 Recoding Variables

Often times it is necessary to create a new variable based on values of an old variable. For example, in contingency table analysis we may need to group together observations with different values, and assign them all a new value. For logistic regression, it may be necessary to change a continuous variable into one that takes on values of either 0 or 1. For simple cases, logical variables can be used directly to convert a continuous variable to a binary one. For example, using the `iris` data frame, suppose we wanted to create a new variable, `bigsepal`, which would be `TRUE` when `Sepal.Length` was greater than 6, and `FALSE` otherwise. We can simply create the appropriate logical variable:

```
> bigsepal = iris$Sepal.Length > 6
```

When a logical variable is used in a numeric context, it is automatically converted to 1 if it is `TRUE` and 0 if it is `FALSE`. Thus, logical variables can be manipulated to create categorical variables with more than two levels. Suppose we wanted to create a categorical variable called `sepalgroup`, based on `Sepal.Length`, which would be equal to 1 for lengths less than or equal to 5, 2 for lengths between 5 and 7, and 3 for lengths greater than or equal to 7. We could combine logical variables as follows:

```
> sepalgroup = 1 + (iris$Sepal.Length >= 5)
+                + (iris$Sepal.Length >= 7)
```

Note that in this case the same result could be achieved using `cut` (see Section 5.4):

```
> sepalgroup = cut(iris$Sepal.Length,c(0,5,7,10),
+                  include.lowest=TRUE,right=FALSE)
```

For some recoding tasks, the `ifelse` function may be more useful than manipulating logical variables directly. Suppose we have a variable called `group` that takes on values in the range of 1 to 5, and we wish to create a new variable that will be equal to 1 if the original variable is either 1 or 5, and equal to 2 otherwise. The `ifelse` statement accepts a logical vector as its first argument, and two other arguments: the first provides a value for the case where elements of the input logical vector are true, and the second for the case where they are false. So in this example, we could get the desired result using

```
> newgroup = ifelse(group %in% c(1,5),1,2)
```

The second and third arguments to `ifelse` will be recycled as necessary to be conformable with the input logical vector.

Note that the object returned by `ifelse` will be the same shape as the first input argument, so `ifelse` is effectively limited to cases where the desired result for each element is a scalar. If either of the second or third arguments to `ifelse` returns a vector, the return value of `ifelse` will be silently truncated to just its first element.

Calls to `ifelse` can be nested. Continuing with the previous example, if we wanted to recode values of 1 and 5 to 1, 2 and 4 to 2, and other values (in this case 3) to 3, we could use nested calls to `ifelse` as follows:

```
> newgroup = ifelse(group %in% c(1,5),1,
+                   ifelse(group %in% c(2,4),2,3))
```

Some words of warning about `ifelse` are in order. If any of the elements of the first argument to `ifelse` are `TRUE`, then all of the values in the second argument will need to be evaluated. Similarly, if any of the input elements are `FALSE`, then each value in the third argument must be evaluated. If either of the alternative values requires a large amount of computation, this may make `ifelse` surprisingly slow. Additionally, using `ifelse` with a variety of

data may result in surprises. As a simple example, suppose we have a vector, `x`, and we wish to take the logarithm of the values greater than 0, and the absolute value of values less than or equal to zero. If we happen to provide a vector with all values less than zero, there is no problem:

```
> x = c(-1.2,-3.5,-2.8,-1.1,-0.7)
> newx = ifelse(x > 0,log(x),abs(x))
> newx
[1] 1.2 3.5 2.8 1.1 0.7
```

As soon as one or more values in the vector satisfy the condition `x > 0`, warnings will appear when R tries to evaluate the logarithm of the negative numbers, even though it will never actually return them:

```
> x = c(-1.2,-3.5,-2.8,1.1,-0.7)
> newx = ifelse(x > 0,log(x),abs(x))
Warning message:
NaNs produced in: log(x)
> newx
[1] 1.20000000 3.50000000 2.80000000 0.09531018 0.70000000
```

At the expense of some additional operations, the problem can be avoided:

```
> newx = numeric(length(x))
> newx[x > 0] = log(x[x > 0])
> newx[x <= 0] = abs(x[x <= 0])
> newx
[1] 1.20000000 3.50000000 2.80000000 0.09531018 0.70000000
```

Since expressions in R return their evaluated values, yet another solution is to use `sapply` with `if/else` expressions:

```
> newx = sapply(x,function(t)if(t > 0)log(t) else abs(t))
```

9.3 The `recode` Function

A very flexible approach to recoding variables is provided by the `recode` function of the `car` package, available through CRAN. Similar to facilities in other statistical languages, the `recode` function accepts descriptions of ranges of values along with a new, constant value to be assigned to observations within those ranges. These range/value pairs are passed to `recode` as a character string, with equal signs (`=`) separating ranges and values, and semicolons (`;`) separating each range/value pair.

There are four possibilities for range/value pairs:

1. single values, for example `3='control'`
2. multiple values, for example `c(1,5)=5`

3. ranges of values, for example `5:7='middle'`. The special values `lo` and `hi` can appear in a range to represent the lowest or highest value for the variable being recoded.
4. the word `else`, representing values not covered by any other provided ranges, for example `else='not found'`.

So to recode values 1 and 5 to 1, 2 and 4 to 2, and other values to 3, we could use `recode` as follows (after loading the `car` package):

```
> newgroup = recode(group,'c(1,5)=1;c(2,4)=2;else=3')
```

9.4 Reshaping Data Frames

Often the values required for a particular operation can be found in a data frame, but they are not organized in the appropriate way. As a simple example, data for multiple groups are often stored in spreadsheets or data summaries as columns, with a separate column for each group. Most of the modeling and graphics functions in R will not be able to work with such data; they expect the values to be in a single column with an additional column that specifies the group from which the data arose. The `stack` function can reorganize datasets to have this property. As an example, suppose that data for three groups is stored in a data frame as follows:

```
> mydata = data.frame(grp1=c(12,15,19,22,25),
+                     grp2=c(18,12,42,29,44),
+                     grp3=c(8,17,22,19,31))
> mydata
  grp1 grp2 grp3
1   12   18    8
2   15   12   17
3   19   42   22
4   22   29   19
5   25   44   31
```

To perform an analysis of variance or produce histograms for each group, the data would need to rearranged using `stack`:

```
> sdata = stack(mydata)
> head(sdata)
  values  ind
1     12 grp1
2     15 grp1
3     19 grp1
4     22 grp1
5     25 grp1
6     18 grp2
```

If there were other variables in the data frame that did not need to be converted to this form, the `select=` argument to `stack` allows you to specify the variables that should be used, similar to the same argument to the `subset` function.

The `unstack` function will reorganize stacked data back to the one column per group form. To use `unstack`, a formula must be provided to explain the roles of the variables to be unstacked. To convert the `sdata` data frame back to its original form, `unstack` could be called as follows:

```
> mydata = unstack(sdata,values~ind)
> head(mydata)
  grp1 grp2 grp3
1   12   18    8
2   15   12   17
3   19   42   22
4   22   29   19
5   25   44   31
```

For more complex reorganizations, the concept of "wide" versus "long" datasets is often helpful. When there are multiple occurrences of values for a single observation, a data frame is said to be long if each occurrence is a separate row in the data frame; if all of the occurrences of values for a given observation are in the same row, then the dataset is said to be wide. The `reshape` function converts datasets between these two forms.

Perhaps the most common use of `reshape` involves repeated measures analyses, where the same variable is recorded for each observation at several different times. For some types of analysis (for example, split-plot designs), the long form is preferred; for other analyses (for example, correlation studies), the wide form is needed. For example, consider the following artificial dataset which contains observations at three different times for four subjects on variables called `x` and `y`:

```
> set.seed(17)
> obs = data.frame(subj=rep(1:4,rep(3,4)),
+                  time=rep(1:3),
+                  x=rnorm(12),y=rnorm(12))
> obs
   subj time           x           y
1     1    1 -1.01500872  1.29532187
2     1    2 -0.07963674  0.18791807
3     1    3 -0.23298702  1.59120510
                  . . .
9     3    3  0.25523700  0.68102765
10    4    1  0.36658112 -0.68203337
11    4    2  1.18078924 -0.72325674
12    4    3  0.64319207  1.67352596
```

To use `reshape` to convert the dataset to wide format, we need to provide five arguments. The first argument is the data frame to be reshaped. The next three arguments provide the names of the columns that will be involved in the reshaping. The `idvar=` argument provides the names of the variables that define the experimental unit which was repeatedly measured. In this case, it's the `subj` variable. The `v.names=` argument tells `reshape` which variables in the long format will be used to create the multiple variables in the wide format. In this example, we want both `x` and `y` be to be expanded to multiple variables, so we'd specify a vector with both those names. The `timevar=` variable tells which variable identifies the sequence number that will be used to create the multiple versions of the `v.names` variables; in this case it will be `time`. Finally, the `direction=` argument accepts values of `"wide"` or `"long"`, depending on which transformation is to be performed. Putting this all together, we can perform the conversion to wide format with the following call to `reshape`:

```
> wideobs = reshape(obs,idvar='subj',v.names=c('x','y'),
+                   timevar='time',direction='wide')
> wideobs
   subj        x.1         y.1         x.2        y.2
1     1 -1.0150087  1.29532187 -0.07963674  0.1879181
4     2 -0.8172679 -0.05517906  0.77209084  0.8384711
7     3  0.9728744  0.62595440  1.71653398  0.6335847
10    4  0.3665811 -0.68203337  1.18078924 -0.7232567
          x.3       y.3
1  -0.2329870 1.5912051
4  -0.1656119 0.1593701
7   0.2552370 0.6810276
10  0.6431921 1.6735260
```

Notice that the names of the variables are passed to `reshape`, not the actual values of the variables.

The names `x.1`, `y.1`, etc. were formed by joining together the variable names of the variables specified in the `v.names=` argument with the values of the `timevar=` variable. Any variables not specified in the `v.names=` argument are assumed to be constant for all observations with the same values as the `idvar=` variables, and a single copy of such variables will be included in the output data frame. Only the variables whose names appear in the `v.names=` argument will be converted into multiple variables, so if any variables that are in the data frame but not in the `v.names=` argument are not constant, `reshape` will print a warning message, and use the first value of such variables when converting to wide format. To prevent variables from being transferred to the output data frame, the `drop=` argument can be used to pass a vector of variable names to be ignored in the conversion.

The information about the reshaping procedure is stored as attributes in converted data frames, so once a data frame has been converted with `reshape`, it can be changed to its previous format by passing just the data frame with

no additional arguments to `reshape`. Thus, we could convert the `wideobs` data frame to its original long format as follows:

```
> obs = reshape(wideobs)
> head(obs)
    subj time           x           y
1.1    1    1 -1.01500872  1.29532187
2.1    2    1 -0.81726793 -0.05517906
3.1    3    1  0.97287443  0.62595440
4.1    4    1  0.36658112 -0.68203337
1.2    1    2 -0.07963674  0.18791807
2.2    2    2  0.77209084  0.83847112
```

As an example of converting from wide to long format, consider the `USPersonalExpenditure` dataset. Since it is stored as a matrix, we'll first convert it to a data frame, transferring the row names into a variable called `type`:

```
> usp = data.frame(type=rownames(USPersonalExpenditure),
+                       USPersonalExpenditure,row.names=NULL)
> usp
                  type  X1940  X1945 X1950 X1955 X1960
1    Food and Tobacco 22.200 44.500 59.60  73.2 86.80
2 Household Operation 10.500 15.500 29.00  36.5 46.20
3  Medical and Health  3.530  5.760  9.71  14.0 21.10
4       Personal Care  1.040  1.980  2.45   3.4  5.40
5   Private Education  0.341  0.974  1.80   2.6  3.64
```

Since `reshape` can handle multiple sets of variables, the `varying=` argument should be passed a list containing vectors with the names of the different sets of variables that should be mapped to a single variable in the long dataset. In the current example, there is only one set of variables to be mapped, so we pass a list with a vector of the appropriate variable names. Along with the `direction='long'` argument, this list will usually be enough to convert the dataset:

```
> rr = reshape(usp,varying=list(names(usp)[-1]),direction='long')
> head(rr)
                    type  time   X1940 id
1.1     Food and Tobacco     1  22.200  1
2.1  Household Operation     1  10.500  2
3.1   Medical and Health     1   3.530  3
4.1        Personal Care     1   1.040  4
5.1    Private Education     1   0.341  5
1.2     Food and Tobacco     2  44.500  1
```

By providing additional information to `reshape`, the resulting data frame can be modified to provide more useful information. For example, the automatically generated variable `id` is simply a numeric index corresponding to

the `type` variable; using `idvar='type'` will suppress its creation. The automatically generated variable `time` defaults to a set of consecutive integers; providing more meaningful values through the `times=` argument will label the values properly. Finally, the name of the column representing the values (which defaults to the first name in the `varying=` argument) can be set to a more meaningful name with the `v.names=` argument.

```
> rr=reshape(usp,varying=list(names(usp)[-1]),idvar='type',
+            times=seq(1940,1960,by=5),v.names='expend',
+            direction='long')
> head(rr)
                                        type time expend
Food and Tobacco.1940       Food and Tobacco 1940 22.200
Household Operation.1940 Household Operation 1940 10.500
Medical and Health.1940   Medical and Health 1940  3.530
Personal Care.1940             Personal Care 1940  1.040
Private Education.1940     Private Education 1940  0.341
Food and Tobacco.1945       Food and Tobacco 1945 44.500
```

In cases like this, where the desired value for time is embedded in the variable names being converted, the `split=` argument can be used to automatically determine the values for the times and names for the variables containing the values. When you use the `split=` argument, the `varying=` argument should be a vector, not a list, because `reshape` will figure out the sets of variables based on the prefixes found by splitting the variable names. The `split=` argument is passed as a list with two elements: `regexp` and `include`. The `regexp` argument provides a regular expression used to split up the names provided through the `varying=` argument. The first split piece will be used as a name for the variable containing the values, and the second split piece will be used to form values for the `time` variable that `reshape` generates. To keep the regular expression as part of the names and values that are created, the `include` argument should be set to `TRUE`. So an alternative way of reshaping the `usp` data frame, without having to explicitly provide the values of the times, would be:

```
> rr1 = reshape(usp,varying=names(usp)[-1],idvar='type',
+       split=list(regexp='X1',include=TRUE),direction='long')
> head(rr1)
                                        type time      X
Food and Tobacco.1940       Food and Tobacco 1940 22.200
Household Operation.1940 Household Operation 1940 10.500
Medical and Health.1940   Medical and Health 1940  3.530
Personal Care.1940             Personal Care 1940  1.040
Private Education.1940     Private Education 1940  0.341
Food and Tobacco.1945       Food and Tobacco 1945 44.500
```

To replace the generated row names with ones of your own choosing, use the `new.row.names=` argument.

9.5 The reshape Package

The reshape package, introduced in Section 8.6, uses the concept of "melting" a dataset (through the melt function) into a data frame which contains separate columns for each id variable, a variable column containing the name of each measured variable, and a final column named value with the variable's value. It may be noticed that this melting operation is essentially a "wide-to-long" reshaping of the data. Using the usp data frame from a previous example, we can easily convert the melted form to the long form as follows:

```
> library(reshape)
> usp = data.frame(type=rownames(USPersonalExpenditure),
+                        USPersonalExpenditure,row.names=NULL)
> musp = melt(usp)
> head(musp)
                 type variable  value
1    Food and Tobacco    X1940 22.200
2 Household Operation    X1940 10.500
3  Medical and Health    X1940  3.530
4       Personal Care    X1940  1.040
5   Private Education    X1940  0.341
6    Food and Tobacco    X1945 44.500
```

To complete the conversion, we need only remove the "X" from the variable column, rename it to time, and rename the value column to expend:

```
> musp$variable = as.numeric(sub('X','',musp$variable))
> names(musp)[2:3] = c('time','expend')
> head(musp)
                 type time expend
1    Food and Tobacco 1940 22.200
2 Household Operation 1940 10.500
3  Medical and Health 1940  3.530
4       Personal Care 1940  1.040
5   Private Education 1940  0.341
6    Food and Tobacco 1945 44.500
```

Keep in mind that variable is a factor, and that the sub function converts it to a character before operating on it; if you use it directly, you may need to pass it to as.character before processing. Since both the id variables and measure variables appear in the columns of the "long" dataset, this transformation could also be performed using

```
cast(musp,variable + type ~ .)
```

For long-to-wide conversions, recall that variables appearing to the left of the tilde in the formula passed to cast will appear in the columns of the output, while those on the right will appear in the rows. Using the simulated

data from the previous section, we put subj on the left-hand side of the formula and variable (created by the melt function) and time on the right:

```
> set.seed(17)
> obs = data.frame(subj=rep(1:4,rep(3,4)),
+                  time=rep(1:3),
+                  x=rnorm(12),y=rnorm(12))
> mobs = melt(obs)
> cast(subj ~ variable + time,data=mobs)
  subj        x_1         x_2        x_3         y_1        y_2
1    1 -1.0150087 -0.07963674 -0.2329870  1.29532187  0.1879181
2    2 -0.8172679  0.77209084 -0.1656119 -0.05517906  0.8384711
3    3  0.9728744  1.71653398  0.2552370  0.62595440  0.6335847
4    4  0.3665811  1.18078924  0.6431921 -0.68203337 -0.7232567
        y_3
1 1.5912051
2 0.1593701
3 0.6810276
4 1.6735260
```

The names of the derived columns are constructed in the order in which the right-hand-side variables are entered in the formula.

To separate each time into a separate list element, the vertical bar (|) can be used:

```
> cast(subj ~variable|time,data=mobs)
$`1`
  subj          x           y
1    1 -1.0150087  1.29532187
2    2 -0.8172679 -0.05517906
3    3  0.9728744  0.62595440
4    4  0.3665811 -0.68203337

$`2`
  subj           x          y
1    1 -0.07963674  0.1879181
2    2  0.77209084  0.8384711
3    3  1.71653398  0.6335847
4    4  1.18078924 -0.7232567

$`3`
  subj          x         y
1    1 -0.2329870 1.5912051
2    2 -0.1656119 0.1593701
3    3  0.2552370 0.6810276
4    4  0.6431921 1.6735260
```

It can be noted that this performs the same operation as the `split` function (Section 8.5), but the redundant variable (`time` in this example) is not included in the output.

Remember that the dataset that `cast` is operating on is the melted dataset, not the original one. So to create a wide data frame from the simulated data, but only including `x`, we could use

```
> cast(subj ~ variable + time,subset = variable == 'x',data=mobs)
  subj        x_1         x_2        x_3
1    1 -1.0150087 -0.07963674 -0.2329870
2    2 -0.8172679  0.77209084 -0.1656119
3    3  0.9728744  1.71653398  0.2552370
4    4  0.3665811  1.18078924  0.6431921
```

9.6 Combining Data Frames

At the most basic level, two or more data frames can be combined by rows using `rbind`, or by columns using `cbind`. For `rbind`, the data frames must have the same number of columns; for `cbind`, the data frames must have the same number of rows. Vectors or matrices passed to `cbind` will be converted to data frames, so the mode of columns passed to `cbind` will be preserved.

While `cbind` will demand that data frames and matrices are conformable (that is, they have the same number of rows), vectors passed to `cbind` will be recycled if the number of rows in the data frame or matrix is an even multiple of the length of the vector. Consider the following two data frames, one with three rows, and one with four:

```
> x = data.frame(a=c('A','B','C'),x=c(12,15,19))
> y = data.frame(a=c('D','E','F','G'),x=c(19,21,14,12))
```

We can bind a vector with two values to the second data frame, since four is an even multiple of two; R will recycle the vectors values to insure conformability:

```
> cbind(y,z=c(1,2))
  a  x z
1 D 19 1
2 E 21 2
3 F 14 1
4 G 12 2
```

When using `cbind`, duplicate column names will not be detected:

```
> cbind(x,y[1:3,])
  a  x a  x
1 A 12 D 19
2 B 15 E 21
3 C 19 F 14
```

It may be a good idea to use unique names when combining data frames in this way. An easy way to test is to pass the names of the two data frames to the `intersect` function:

```
> intersect(names(x),names(y))
[1] "a" "x"
```

When using `rbind`, the names and classes of values to be joined must match, or a variety of errors may occur. This is especially important when values in any of the columns involved are factors. Using the `data.frame` function when adding rows to a data frame can usually resolve the problem:

```
> z = rbind(x,c(a='X',x=12))
Warning message:
invalid factor level, NAs generated in:
"[<-.factor"(`*tmp*`, ri, value = "X")
> z = rbind(x,data.frame(a='X',x=12))
> levels(z$a)
[1] "A" "B" "C" "X"
```

Although the `rbind` function will demand that the names of the objects being combined agree, `cbind` does not do any such checking. To combine data frames based on the values of common variables, the `merge` function should be used. This function is designed to provide the same sort of functionality and behavior as the table joins provided by relational databases. Although `merge` is limited to operating on two data frames at a time, it can be called repeatedly to deal with more than two data frames.

The default behavior of `merge` is to join together rows of the data frames based on the values of all of the variables (columns) that the data frames have in common. (In database terminology, this is known as a natural join.) When called without any other arguments, `merge` returns only those rows which had observations in both data frames. As a simple example, consider the merge resulting from these two data frames, each of which has rows with values of the merging variable that are not found in the other data frame:

```
> x = data.frame(a=c(1,2,4,5,6),x=c(9,12,14,21,8))
> y = data.frame(a=c(1,3,4,6),y=c(8,14,19,2))
> merge(x,y)
  a  x  y
1 1  9  8
2 4 14 19
3 6  8  2
```

Although there were six unique values for `a` between the two data frames, only those rows with values of `a` in both data frames are represented in the output. To modify this, the `all=`, `all.x=`, and `all.y=` arguments can be used. Specifying `all=TRUE` will include all rows (full outer join, in database terminology), `all.x=TRUE` will include all rows from the first data frame (left outer join), and `all.y=TRUE` does the same for the second data frame (right outer join). Each case can be illustrated with the current example:

```
> merge(x,y,all=TRUE)
  a  x  y
1 1  9  8
2 2 12 NA
3 3 NA 14
4 4 14 19
5 5 21 NA
6 6  8  2
> merge(x,y,all.x=TRUE)
  a  x  y
1 1  9  8
2 2 12 NA
3 4 14 19
4 5 21 NA
5 6  8  2
> merge(x,y,all.y=TRUE)
  a  x  y
1 1  9  8
2 3 NA 14
3 4 14 19
4 6  8  2
```

Note that missing values (`NA`) are inserted in the places where data was missing from one of the data frames.

To take more control over which variables are used to merge rows of the data frame, the `by=` argument can be used. You provide the `by=` argument with a vector of the name or names of the variables that should be used for the merge. If the merging variables have different names in the data frames to be merged, the `by.x=` and `by.y=` arguments can be used.

When there are multiple rows with common values of the merging variable in either of the data frames being merged, each row will contribute one observation to the output data frame. If one of the datasets has exactly one observation for each value of the merging variable(s), the resultant merge is sometimes known as a table lookup. As a simple example, consider two datasets, one with city names and state abbreviations, and a second with state abbreviations and full state names. The goal is to create a dataset with the names of the cities along with the full state names. The following dataset represents 10 of the most expensive cities in the United States, based on housing and food costs:

```
> cities = data.frame(city=c('New York','Boston','Juneau',
+                       'Anchorage','San Diego',
+                       'Philadelphia','Los Angeles',
+                       'Fairbanks','Ann Arbor','Seattle'),
+                     state.abb= c('NY','MA','AK','AK','CA',
+                                  'PA','CA','AK','MI','WA'))
```

```
> cities
           city state.abb
1      New York        NY
2        Boston        MA
3        Juneau        AK
4     Anchorage        AK
5     San Diego        CA
6  Philadelphia        PA
7   Los Angeles        CA
8     Fairbanks        AK
9     Ann Arbor        MI
10      Seattle        WA
```

A corresponding data frame with state abbreviations and full names can be formed as follows:

```
> states = data.frame(state.abb= c('NY','MA','AK','CA',
+                                  'PA','MI','WA'),
+                state=c('New York','Massachusetts','Alaska',
+                        'California','Pennsylvania',
+                        'Michigan','Washington'))
```

Note that there is exactly one observation for each state/abbreviation combination in the `states` dataset. With this restriction in place, merging the two datasets is simple (since they have a single variable, `state.abb`, in common:

```
> merge(cities,states)
   state.abb         city         state
1         AK       Juneau        Alaska
2         AK    Anchorage        Alaska
3         AK    Fairbanks        Alaska
4         CA    San Diego    California
5         CA  Los Angeles    California
6         MA       Boston Massachusetts
7         MI    Ann Arbor      Michigan
8         NY     New York      New York
9         PA Philadelphia  Pennsylvania
10        WA      Seattle    Washington
```

The multiple observations per state in the `cities` data frame cause no problem, because there was always exactly one matching observation in the `states` data frame.

Now suppose we (foolishly) create a data frame with the zip codes for various cities using only the state abbreviation as an identifier. The problem is that there will be more than one zip code for some of the states, making it impossible for `merge` to know exactly which observations should be joined together. In cases like this, `merge` silently creates multiple observations so

that there will be an observation for each multiple occurrence of the merging variables in the merged data frame.

```
> zips = data.frame(state.abb=c('NY','MA','AK','AK','CA',
+                               'PA','CA','AK','MI','WA'),
+            zip=c('10044','02129','99801','99516','92113',
+                  '19127','90012','99709','48104','98104'))
> merge(cities,zips)
   state.abb         city   zip
1         AK       Juneau 99801
2         AK       Juneau 99516
3         AK       Juneau 99709
4         AK    Anchorage 99801
5         AK    Anchorage 99516
6         AK    Anchorage 99709
7         AK    Fairbanks 99801
8         AK    Fairbanks 99516
9         AK    Fairbanks 99709
10        CA    San Diego 92113
11        CA    San Diego 90012
12        CA  Los Angeles 92113
13        CA  Los Angeles 90012
14        MA       Boston 02129
15        MI    Ann Arbor 48104
16        NY     New York 10044
17        PA Philadelphia 19127
18        WA      Seattle 98104
```

Now there are 18 observations in the output dataset instead of the expected 10. For any state for which there were multiple observations in the `zips` data frame, `merge` has created that many observations for each observation in the `cities` dataset with that value of `state.abb`. The moral is that you should proceed with caution when you have multiple occurrences of values of a merging variable in both of the datasets being merged.

9.7 Under the Hood of `merge`

While the `merge` function will perform most common tasks regarding combining two data frames, it is occasionally useful to just find the indexes of common values in two vectors, rather than actually combining them. Internally, `merge` uses the `match` function to find these indexes. This function requires two arguments: the first is a vector of values to be matched, and the second is the vector of values that should be searched for possible matches. For those elements in the first vector that had matching values in the second,

match returns the index of the first such value in the second vector; for elements that didn't match, the default behavior of match is to return a missing value (NA). Thus, the return value from match will always be a vector of the same length as the first argument. For example, in Section 9.6, we merged the cities and states data frames based on common values of the state.abb variable. To retrieve just the indexes of the matching values, we could call match as follows:

```
> match(cities$state.abb,states$state.abb)
 [1] 1 2 3 3 4 5 4 3 6 7
```

The nomatch= argument can be used to provide a different value to be returned when a match was not found. Since subscripts of 0 are ignored, one very useful choice for this value is nomatch=0. When this value is used, the result from match can be used as an index to the second vector to find the values that actually matched. Continuing with the x and y example from Section 9.6, suppose we wanted to know which values in x$a were also present in x$b. By calling match with nomatch=0, the resulting vector can be used as an index into y$b to extract the actual values:

```
> indices = match(x$a,y$a,nomatch=0)
> y$a[indices]
[1] 1 4 6
```

It may be noted that this is equivalent to the intersect function, which currently uses match to do its work.

Finally, for the simpler case where interest is only in whether or not elements in one vector can be found in another vector, the %in% operator can be used. To produce a logical vector, the same length as x$a, which indicates which values could be found in y$a, we can use %in% as follows:

```
> x$a %in% y$a
[1]  TRUE FALSE  TRUE FALSE  TRUE
```

Like intersect, %in% is currently defined using match.