5 Advanced data management

This chapter covers

- Mathematical and statistical functions
- Character functions
- Looping and conditional execution
- User-written functions
- Ways to aggregate and reshape data

In chapter 4, we reviewed the basic techniques used for managing datasets within R. In this chapter, we'll focus on advanced topics. The chapter is divided into three basic parts. In the first part we'll take a whirlwind tour of R's many functions for mathematical, statistical, and character manipulation. To give this section relevance, we begin with a data management problem that can be solved using these functions. After covering the functions themselves, we'll look at one possible solution to the data management problem.

Next, we cover how to write your own functions to accomplish data management and analysis tasks. First, you'll explore ways of controlling program flow, including looping and conditional statement execution. Then we'll investigate the structure of user-written functions and how to invoke them once created.

Then, we'll look at ways of aggregating and summarizing data, along with methods of reshaping and restructuring datasets. When aggregating data, you

can specify the use of any appropriate built-in or user-written function to accomplish the summarization, so the topics you learned in the first two parts of the chapter will provide a real benefit.

5.1 A data management challenge

To begin our discussion of numerical and character functions, let's consider a data management problem. A group of students have taken exams in Math, Science, and English. You want to combine these scores in order to determine a single performance indicator for each student. Additionally, you want to assign an A to the top 20 percent of students, a B to the next 20 percent, and so on. Finally, you want to sort the students alphabetically. The data are presented in table 5.1.

Table 5.1 Student exam data

Student	Math	Science	English
John Davis	502	95	25
Angela Williams	600	99	22
Bullwinkle Moose	412	80	18
David Jones	358	82	15
Janice Markhammer	495	75	20
Cheryl Cushing	512	85	28
Reuven Ytzrhak	410	80	15
Greg Knox	625	95	30
Joel England	573	89	27
Mary Rayburn	522	86	18

Looking at this dataset, several obstacles are immediately evident. First, scores on the three exams aren't comparable. They have widely different means and standard deviations, so averaging them doesn't make sense. You must transform the exam scores into comparable units before combining them. Second, you'll need a method of determining a student's percentile rank on this score in order to assign a grade. Third, there's a single field for names, complicating the task of sorting students. You'll need to break apart their names into first name and last name in order to sort them properly.

Each of these tasks can be accomplished through the judicious use of R's numerical and character functions. After working through the functions described in the next section, we'll consider a possible solution to this data management challenge.

5.2 Numerical and character functions

In this section, we'll review functions in R that can be used as the basic building blocks for manipulating data. They can be divided into numerical (mathematical, statistical, probability) and character functions. After we review each type, I'll show you how to apply functions to the columns (variables) and rows (observations) of matrices and data frames (see section 5.2.6).

5.2.1 Mathematical functions

Table 5.2 lists common mathematical functions along with short examples.

Table 5.2 Mathematical functions

Function	Description
`abs(x)`	Absolute value `abs(-4)` returns 4.
`sqrt(x)`	Square root `sqrt(25)` returns 5. This is the same as `25^(0.5)`.
`ceiling(x)`	Smallest integer not less than *x* `ceiling(3.475)` returns 4.
`floor(x)`	Largest integer not greater than *x* `floor(3.475)` returns 3.
`trunc(x)`	Integer formed by truncating values in *x* toward 0 `trunc(5.99)` returns 5.
`round(x, digits=n)`	Round *x* to the specified number of decimal places `round(3.475, digits=2)` returns 3.48.
`signif(x, digits=n)`	Round *x* to the specified number of significant digits `signif(3.475, digits=2)` returns 3.5.
`cos(x), sin(x), tan(x)`	Cosine, sine, and tangent `cos(2)` returns –0.416.
`acos(x), asin(x), atan(x)`	Arc-cosine, arc-sine, and arc-tangent `acos(-0.416)` returns 2.
`cosh(x), sinh(x), tanh(x)`	Hyperbolic cosine, sine, and tangent `sinh(2)` returns 3.627.
`acosh(x), asinh(x), atanh(x)`	Hyperbolic arc-cosine, arc-sine, and arc-tangent `asinh(3.627)` returns 2.

`log(x,base=n)`
`log(x)`
`log10(x)`
	Logarithm of *x* to the base *n*
	For convenience
	`log(x)` is the natural logarithm.
	`log10(x)` is the common logarithm.
	`log(10)` returns 2.3026.
	`log10(10)` returns 1.

Table 5.2 Mathematical functions *(continued)*

Function	Description
`exp(x)`	Exponential function `exp(2.3026)` returns 10.

Data transformation is one of the primary uses for these functions. For example, you often transform positively skewed variables such as income to a log scale before further analyses. Mathematical functions will also be used as components in formulas, in plotting functions (for example, `x versus sin(x)`) and in formatting numerical values prior to printing.

The examples in table 5.2 apply mathematical functions to scalars (individual numbers). When these functions are applied to numeric vectors, matrices, or data frames, they operate on each individual value. For example, `sqrt(c(4, 16, 25))` returns `c(2, 4, 5)`.

5.2.2 Statistical functions

Common statistical functions are presented in table 5.3. Many of these functions have optional parameters that affect the outcome. For example:

```
y <- mean(x)
```

provides the arithmetic mean of the elements in object x, and

```
z <- mean(x, trim = 0.05, na.rm=TRUE)
```

provides the trimmed mean, dropping the highest and lowest 5 percent of scores and any missing values. Use the `help()` function to learn more about each function and its arguments.

Table 5.3 Statistical functions

Function	Description
`mean(x)`	Mean `mean(c(1,2,3,4))` returns 2.5.
`median(x)`	Median `median(c(1,2,3,4))` returns 2.5.
`sd(x)`	Standard deviation `sd(c(1,2,3,4))` returns 1.29.
`var(x)`	Variance `var(c(1,2,3,4))` returns 1.67.
`mad(x)`	Median absolute deviation `mad(c(1,2,3,4))` returns 1.48.

Table 5.3 Statistical functions *(continued)*

Function	Description

`quantile(x, probs)`	Quantiles where *x* is the numeric vector where quantiles are desired and *probs* is a numeric vector with probabilities in [0,1].
	`# 30th and 84th percentiles of x`
	`y <- quantile(x, c(.3,.84))`

`range(x)`	Range `x <- c(1,2,3,4)` `range(x)` returns c(1,4). `diff(range(x))` returns 3.
`sum(x)`	Sum `sum(c(1,2,3,4))` returns 10.
`diff(x, lag=n)`	Lagged differences, with `lag` indicating which lag to use. The default lag is 1. `x<- c(1, 5, 23, 29)` `diff(x)` returns `c(4, 18, 6)`.
`min(x)`	Minimum `min(c(1,2,3,4))` returns 1.
`max(x)`	Maximum `max(c(1,2,3,4))` returns 4.
`scale(x, center=TRUE, scale=TRUE)`	Column center `(center=TRUE)` or standardize `(center=TRUE, scale=TRUE)` data object *x*. An example is given in listing 5.6.

To see these functions in action, look at the next listing. This listing demonstrates two ways to calculate the mean and standard deviation of a vector of numbers.

Listing 5.1 Calculating the mean and standard deviation

```
> x <- c(1,2,3,4,5,6,7,8)

> mean(x)                    <-- Short way
[1] 4.5
> sd(x)
[1] 2.449490

> n <- length(x)             <-- Long way
> meanx <- sum(x)/n
> css <- sum((x - meanx)^2)
> sdx <- sqrt(css / (n-1))
> meanx
[1] 4.5
> sdx
[1] 2.449490
```

It's instructive to view how the corrected sum of squares (`css`) is calculated in the second approach:

1. `x` equals `c(1, 2, 3, 4, 5, 6, 7, 8)` and mean x equals 4.5 (`length(x)` returns the number of elements in `x`).
2. `(x - meanx)` subtracts `4.5` from each element of `x`, resulting in `c(-3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5)`.
3. `(x - meanx)^2` squares each element of `(x - meanx)`, resulting in `c(12.25, 6.25, 2.25, 0.25, 0.25, 2.25, 6.25, 12.25)`.
4. `sum((x - meanx)^2)` sums each of the elements of `(x - meanx)^2)`, resulting in 42.

Writing formulas in R has much in common with matrix manipulation languages such as MATLAB (we'll look more specifically at solving matrix algebra problems in appendix E).

STANDARDIZING DATA

By default, the `scale()` function standardizes the specified columns of a matrix or data frame to a mean of 0 and a standard deviation of 1:

```
newdata <- scale(mydata)
```

To standardize each column to an arbitrary mean and standard deviation, you can use code similar to the following:

```
newdata <- scale(mydata)*SD + M
```

where `M` is the desired mean and `SD` is the desired standard deviation. Using the `scale()` function on non-numeric columns will produce an error. To standardize a specific column rather than an entire matrix or data frame, you can use code such as

```
newdata <- transform(mydata, myvar = scale(myvar)*10+50)
```

This code standardizes the variable `myvar` to a mean of 50 and standard deviation of 10. We'll use the `scale()` function in the solution to the data management challenge in section 5.3.

5.2.3 Probability functions

You may wonder why probability functions aren't listed with the statistical functions (it was really bothering you, wasn't it?). Although probability functions are statistical by definition, they're unique enough to deserve their own section. Probability functions are often used to generate simulated data with known characteristics and to calculate probability values within user-written statistical functions.

In R, probability functions take the form

```
[dpqr]distribution_abbreviation()
```

where the first letter refers to the aspect of the *distribution* returned:

- `d` = density
- `p` = distribution function
- `q` = quantile function
- `r` = random generation (random deviates)

The common probability functions are listed in table 5.4.

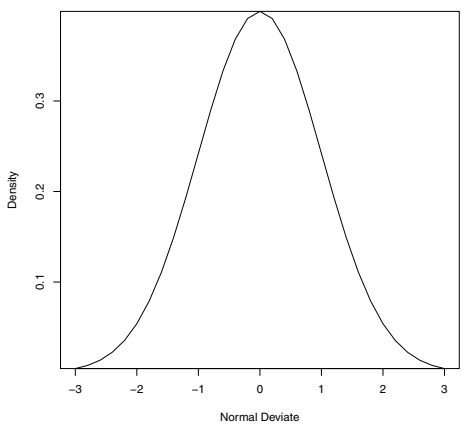
Table 5.4 Probability distributions

Distribution	Abbreviation	Distribution	Abbreviation
Beta	`beta`	Logistic	`logis`
Binomial	`binom`	Multinomial	`multinom`
Cauchy	`cauchy`	Negative binomial	`nbinom`
Chi-squared (noncentral)	`chisq`	Normal	`norm`
Exponential	`exp`	Poisson	`pois`
F	`f`	Wilcoxon Signed Rank	`signrank`
Gamma	`gamma`	T	`t`
Geometric	`geom`	Uniform	`unif`
Hypergeometric	`hyper`	Weibull	`weibull`
Lognormal	`lnorm`	Wilcoxon Rank Sum	`wilcox`

To see how these work, let's look at functions related to the normal distribution. If you don't specify a mean and a standard deviation, the standard normal distribution is assumed (mean=0, sd=1). Examples of the density (`dnorm`), distribution (`pnorm`), quantile (`qnorm`) and random deviate generation (`rnorm`) functions are given in table 5.5.

Table 5.5 Normal distribution functions

Problem	Solution

Plot the standard normal curve on the interval [–3,3] (see below)
 
 	
	`x <- pretty(c(-3,3), 30)`
	`y <- dnorm(x)`
	`plot(x, y,`
	`  type = "l",`
	`  xlab = "Normal Deviate",`
	`  ylab = "Density",`
	`  yaxs = "i"`
	`)`

What is the area under the standard normal curve to the left of z=1.96?	`pnorm(1.96)` equals 0.975

Table 5.5 Normal distribution functions *(continued)*

Problem	Solution
What is the value of the 90th percentile of a normal distribution with a mean of 500 and a standard deviation of 100?	`qnorm(.9, mean=500, sd=100)` equals 628.16
Generate 50 random normal deviates with a mean of 50 and a standard deviation of 10.	`rnorm(50, mean=50, sd=10)`

Don't worry if the plot function options are unfamiliar. They're covered in detail in chapter 11; `pretty()` is explained in table 5.7 later in this chapter.

SETTING THE SEED FOR RANDOM NUMBER GENERATION

Each time you generate pseudo-random deviates, a different seed, and therefore different results, are produced. To make your results reproducible, you can specify the seed explicitly, using the `set.seed()` function. An example is given in the next listing. Here, the `runif()` function is used to generate pseudo-random numbers from a uniform distribution on the interval 0 to 1.

Listing 5.2 Generating pseudo-random numbers from a uniform distribution

```
> runif(5)
[1] 0.8725344 0.3962501 0.6826534 0.3667821 0.9255909
> runif(5)
[1] 0.4273903 0.2641101 0.3550058 0.3233044 0.6584988
> set.seed(1234)
> runif(5)
[1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154
> set.seed(1234)
> runif(5)
[1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154
```

By setting the seed manually, you're able to reproduce your results. This ability can be helpful in creating examples you can access at a future time and share with others.

GENERATING MULTIVARIATE NORMAL DATA

In simulation research and Monte Carlo studies, you often want to draw data from multivariate normal distribution with a given mean vector and covariance matrix. The `mvrnorm()` function in the MASS package makes this easy. The function call is

```
mvrnorm(n, mean, sigma)
```

where *n* is the desired sample size, *mean* is the vector of means, and *sigma* is the variance-covariance (or correlation) matrix. In listing 5.3 you'll sample 500 observations from a three-variable multivariate normal distribution with

Mean Vector	230.7	146.7	3.6
Covariance Matrix	15360.8	6721.2	-47.1
	6721.2	4700.9	-16.5
	-47.1	-16.5	0.3

Listing 5.3 Generating data from a multivariate normal distribution

```
> library(MASS)
> options(digits=3)
> set.seed(1234)                                    <- 1 Set random number seed

> mean <- c(230.7, 146.7, 3.6)                      <- 2 Specify mean vector, covariance matrix
> sigma <- matrix(c(15360.8, 6721.2, -47.1,
                    6721.2, 4700.9, -16.5,
                     -47.1,  -16.5,   0.3), nrow=3, ncol=3)

> mydata <- mvrnorm(500, mean, sigma)               <- 3 Generate data
> mydata <- as.data.frame(mydata)
> names(mydata) <- c("y","x1","x2")

> dim(mydata)                                       <- 4 View results
[1] 500 3
> head(mydata, n=10)
       y    x1   x2
1   98.8  41.3 4.35
2  244.5 205.2 3.57
3  375.7 186.7 3.69
4  -59.2  11.2 4.23
5  313.0 111.0 2.91
6  288.8 185.1 4.18
7  134.8 165.0 3.68
8  171.7  97.4 3.81
9  167.3 101.0 4.01
10 121.1  94.5 3.76
```

In listing 5.3, you set a random number seed so that you can reproduce the results at a later time (1). You specify the desired mean vector and variance-covariance matrix (2), and generate 500 pseudo-random observations (3). For convenience, the results are converted from a matrix to a data frame, and the variables are given names. Finally, you confirm that you have 500 observations and 3 variables, and print out the first 10 observations (4). Note that because a correlation matrix is also a covariance matrix, you could've specified the correlations structure directly.

The probability functions in R allow you to generate simulated data, sampled from distributions with known characteristics. Statistical methods that rely on simulated data have grown exponentially in recent years, and you'll see several examples of these in later chapters.

5.2.4 Character functions

Although mathematical and statistical functions operate on numerical data, character functions extract information from textual data, or reformat textual data for printing and reporting. For example, you may want to concatenate a person's first name and last name, ensuring that the first letter of each is capitalized. Or you may want to count the instances of obscenities in open-ended feedback. Some of the most useful character functions are listed in table 5.6.

Table 5.6 Character functions

Function	Description
`nchar(x)`	Counts the number of characters of *x* `x <- c("ab", "cde", "fghij")` `length(x)` returns 3 (see table 5.7). `nchar(x[3])` returns 5.
`substr(x, start, stop)`	Extract or replace substrings in a character vector. `x <- "abcdef"` `substr(x, 2, 4)` returns "bcd". `substr(x, 2, 4) <- "22222"` (*x* is now `"a222ef"`).
`grep(pattern, x, ignore.case=FALSE, fixed=FALSE)`	Search for pattern in *x*. If `fixed=FALSE`, then *pattern* is a regular expression. If `fixed=TRUE`, then *pattern* is a text string. Returns matching indices. `grep("A", c("b","A","c"), fixed=TRUE)` returns 2.
`sub(pattern, replacement, x, ignore.case=FALSE, fixed=FALSE)`	Find *pattern* in *x* and substitute with *replacement* text. If `fixed=FALSE` then *pattern* is a regular expression. If `fixed=TRUE` then *pattern* is a text string. `sub("\\s",".","Hello There")` returns `Hello.There`. Note `"\s"` is a regular expression for finding whitespace; use `"\\s"` instead because `"\"` is R's escape character (see section 1.3.3).
`strsplit(x, split, fixed=FALSE)`	Split the elements of character vector *x* at *split*. If `fixed=FALSE`, then *pattern* is a regular expression. If `fixed=TRUE`, then *pattern* is a text string. `y <- strsplit("abc", "")` returns a 1-component, 3-element list containing `"a" "b" "c"`. `unlist(y)[2]` and `sapply(y, "[", 2)` both return "b".
`paste(..., sep="")`	Concatenate strings after using `sep` string to separate them. `paste("x", 1:3, sep="")` returns `c("x1", "x2", "x3")`. `paste("x",1:3,sep="M")` returns `c("xM1","xM2" "xM3")`. `paste("Today is", date())` returns `Today is Thu Jun 25 14:17:32 2011` (I changed the date to appear more current.)
`toupper(x)`	Uppercase `toupper("abc")` returns "ABC".
`tolower(x)`	Lowercase `tolower("ABC")` returns "abc".

Note that the functions `grep()`, `sub()`, and `strsplit()` can search for a text string (`fixed=TRUE`) or a regular expression (`fixed=FALSE`) (`FALSE` is the default). Regular expressions provide a clear and concise syntax for matching a pattern of text. For example, the regular expression

```
^[hc]?at
```

matches any string that starts with 0 or one occurrences of `h` or `c`, followed by `at`. The expression therefore matches hat, cat, and at, but not bat. To learn more, see the *regular expression* entry in Wikipedia.

5.2.5 Other useful functions

The functions in table 5.7 are also quite useful for data management and manipulation, but they don't fit cleanly into the other categories.

Table 5.7 Other useful functions

Function	Description
`length(x)`	Length of object *x*. `x <- c(2, 5, 6, 9)` `length(x)` returns 4.
`seq(from, to, by)`	Generate a sequence. `indices <- seq(1,10,2)` `indices` is `c(1, 3, 5, 7, 9)`.
`rep(x, n)`	Repeat *x* *n* times. `y <- rep(1:3, 2)` `y` is `c(1, 2, 3, 1, 2, 3)`.
`cut(x, n)`	Divide continuous variable *x* into factor with *n* levels. To create an ordered factor, include the option `ordered_result = TRUE`.
`pretty(x, n)`	Create pretty breakpoints. Divides a continuous variable *x* into `n` intervals, by selecting *n+1* equally spaced rounded values. Often used in plotting.

`cat(… , file = "myfile", append = FALSE)`	Concatenates the objects in … and outputs them to the screen or to a file (if one is declared) .
	`firstname <- c("Jane")`
	`cat("Hello" , firstname, "\n").`

The last example in the table demonstrates the use of escape characters in printing. Use `\n` for new lines, `\t` for tabs, `\'` for a single quote, `\b` for backspace, and so forth (type `?Quotes` for more information). For example, the code

```
name <- "Bob"
cat( "Hello", name, "\b.\n", "Isn\'t R", "\t", "GREAT?\n")
```

produces

```
Hello Bob.
 Isn't R         GREAT?
```

Note that the second line is indented one space. When `cat` concatenates objects for output, it separates each by a space. That's why you include the backspace (`\b`) escape character before the period. Otherwise it would have produced "Hello Bob ."

How you apply the functions you've covered so far to numbers, strings, and vectors is intuitive and straightforward, but how do you apply them to matrices and data frames? That's the subject of the next section.

5.2.6 Applying functions to matrices and data frames

One of the interesting features of R functions is that they can be applied to a variety of data objects (scalars, vectors, matrices, arrays, and data frames). The following listing provides an example.

Listing 5.4 Applying functions to data objects

```
> a <- 5
> sqrt(a)
[1] 2.236068
> b <- c(1.243, 5.654, 2.99)
> round(b)
[1] 1 6 3
> c <- matrix(runif(12), nrow=3)
> c
       [,1]  [,2]  [,3]  [,4]
[1,] 0.4205 0.355 0.699 0.323
[2,] 0.0270 0.601 0.181 0.926
[3,] 0.6682 0.319 0.599 0.215
> log(c)
       [,1]   [,2]   [,3]   [,4]
[1,] -0.866 -1.036 -0.358 -1.130
[2,] -3.614 -0.508 -1.711 -0.077
[3,] -0.403 -1.144 -0.513 -1.538
> mean(c)
[1] 0.444
```

Notice that the mean of matrix c in listing 5.4 results in a scalar (0.444). The `mean()` function took the average of all 12 elements in the matrix. But what if you wanted the 3 row means or the 4 column means?

R provides a function, `apply()`, that allows you to `apply` an arbitrary function to any dimension of a matrix, array, or data frame. The format for the apply function is

```
apply(x, MARGIN, FUN, ...)
```

where *x* is the data object, *MARGIN* is the dimension index, *FUN* is a function you specify, and ... are any parameters you want to pass to *FUN*. In a matrix or data frame `MARGIN=1` indicates rows and `MARGIN=2` indicates columns. Take a look at the examples in listing 5.5.

Listing 5.5 Applying a function to the rows (columns) of a matrix

```
> mydata <- matrix(rnorm(30), nrow=6)
> mydata
         [,1]   [,2]    [,3]   [,4]   [,5]
[1,]  0.71298  1.368 -0.8320 -1.234 -0.790
[2,] -0.15096 -1.149 -1.0001 -0.725  0.506
[3,] -1.77770  0.519 -0.6675  0.721 -1.350
[4,] -0.00132 -0.308  0.9117 -1.391  1.558
[5,] -0.00543  0.378 -0.0906 -1.485 -0.350
[6,] -0.52178 -0.539 -1.7347  2.050  1.569
> apply(mydata, 1, mean)
[1] -0.155 -0.504 -0.511  0.154 -0.310  0.165
> apply(mydata, 2, mean)
[1] -0.2907  0.0449 -0.5688 -0.3442  0.1906
> apply(mydata, 2, mean, trim=0.2)
[1] -0.1699  0.0127 -0.6475 -0.6575  0.2312
```

1 Generate data

2 Calculate row means

3 Calculate column means

4 Calculate trimmed column means

You start by generating a 6 x 5 matrix containing random normal variates 1. Then you calculate the 6 row means 2, and 5 column means 3. Finally, you calculate trimmed column means (in this case, means based on the middle 60 percent of the data, with the bottom 20 percent and top 20 percent of values discarded) 4.

Because *FUN* can be any R function, including a function that you write yourself (see section 5.4), `apply()` is a powerful mechanism. While `apply()` applies a function over the margins of an array, `lapply()` and `sapply()` apply a function over a list. You'll see an example of `sapply` (which is a user-friendly version of `lapply`) in the next section.

You now have all the tools you need to solve the data challenge in section 5.1, so let's give it a try.

5.3 A solution for our data management challenge

Your challenge from section 5.1 is to combine subject test scores into a single performance indicator for each student, grade each student from A to F based on their relative standing (top 20 percent, next 20 percent, etc.), and sort the roster by students' last name, followed by first name. A solution is given in the following listing.

Listing 5.6 A solution to the learning example

```
> options(digits=2)

> Student <- c("John Davis", "Angela Williams", "Bullwinkle Moose",
               "David Jones", "Janice Markhammer", "Cheryl Cushing",
               "Reuven Ytzrhak", "Greg Knox", "Joel England",
               "Mary Rayburn")
> Math <- c(502, 600, 412, 358, 495, 512, 410, 625, 573, 522)
> Science <- c(95, 99, 80, 82, 75, 85, 80, 95, 89, 86)
> English <- c(25, 22, 18, 15, 20, 28, 15, 30, 27, 18)
> roster <- data.frame(Student, Math, Science, English,
                       stringsAsFactors=FALSE)

> z <- scale(roster[,2:4])
> score <- apply(z, 1, mean)
> roster <- cbind(roster, score)
```

Obtain performance scores

```
> y <- quantile(score, c(.8,.6,.4,.2))
> roster$grade[score >= y[1]] <- "A"
> roster$grade[score < y[1] & score >= y[2]] <- "B"
> roster$grade[score < y[2] & score >= y[3]] <- "C"
> roster$grade[score < y[3] & score >= y[4]] <- "D"
> roster$grade[score < y[4]] <- "F"

> name <- strsplit((roster$Student), " ")
> lastname <- sapply(name, "[", 2)
> firstname <- sapply(name, "[", 1)
> roster <- cbind(firstname,lastname, roster[,-1])

> roster <- roster[order(lastname,firstname),]
```

Grade students

Extract last and first names

Sort by last and first names

```
> roster
    Firstname   Lastname Math Science English score grade
6      Cheryl    Cushing  512      85      28  0.35     C
1        John      Davis  502      95      25  0.56     B
9        Joel    England  573      89      27  0.70     B
4       David      Jones  358      82      15 -1.16     F
8        Greg       Knox  625      95      30  1.34     A
5      Janice Markhammer  495      75      20 -0.63     D
3   Bullwinkle      Moose  412      80      18 -0.86     D
10       Mary    Rayburn  522      86      18 -0.18     C
2      Angela   Williams  600      99      22  0.92     A
7      Reuven    Ytzrhak  410      80      15 -1.05     F
```

The code is dense so let's walk through the solution step by step:

Step 1. The original student roster is given. The `options(digits=2)` limits the number of digits printed after the decimal place and makes the printouts easier to read.

```
> options(digits=2)
> roster
             Student Math Science English
1         John Davis  502      95      25
2    Angela Williams  600      99      22
3   Bullwinkle Moose  412      80      18
4         David Jones  358      82      15
5  Janice Markhammer  495      75      20
6     Cheryl Cushing  512      85      28
7     Reuven Ytzrhak  410      80      15
8          Greg Knox  625      95      30
9        Joel England  573      89      27
10       Mary Rayburn  522      86      18
```

Step 2. Because the Math, Science, and English tests are reported on different scales (with widely differing means and standard deviations), you need to make them comparable before combining them. One way to do this is to standardize the variables so that each test is reported in standard deviation units, rather than in their original scales. You can do this with the `scale()` function:

```
> z <- scale(roster[,2:4])
> z
        Math Science English
```

```
 [1,]  0.013   1.078   0.587
 [2,]  1.143   1.591   0.037
 [3,] -1.026  -0.847  -0.697
 [4,] -1.649  -0.590  -1.247
 [5,] -0.068  -1.489  -0.330
 [6,]  0.128  -0.205   1.137
 [7,] -1.049  -0.847  -1.247
 [8,]  1.432   1.078   1.504
 [9,]  0.832   0.308   0.954
[10,]  0.243  -0.077  -0.697
```

Step 3. You can then get a performance score for each student by calculating the row means using the `mean()` function and adding it to the roster using the `cbind()` function:

```
> score <- apply(z, 1, mean)
> roster <- cbind(roster, score)
> roster
            Student   Math  Science English  score
1         John Davis  502       95      25    0.559
2    Angela Williams  600       99      22    0.924
3   Bullwinkle Moose  412       80      18   -0.857
4        David Jones  358       82      15   -1.162
5  Janice Markhammer  495       75      20   -0.629
6     Cheryl Cushing  512       85      28    0.353
7     Reuven Ytzrhak  410       80      15   -1.048
8          Greg Knox  625       95      30    1.338
9        Joel England  573       89      27    0.698
10       Mary Rayburn  522       86      18   -0.177
```

Step 4. The `quantile()` function gives you the percentile rank of each student's performance score. You see that the cutoff for an A is 0.74, for a B is 0.44, and so on.

```
> y <- quantile(roster$score, c(.8,.6,.4,.2))
> y
  80%   60%   40%   20%
 0.74  0.44 -0.36 -0.89
```

Step 5. Using logical operators, you can recode students' percentile ranks into a new categorical grade variable. This creates the variable `grade` in the `roster` data frame.

```
> roster$grade[score >= y[1]] <- "A"
> roster$grade[score < y[1] & score >= y[2]] <- "B"
> roster$grade[score < y[2] & score >= y[3]] <- "C"
> roster$grade[score < y[3] & score >= y[4]] <- "D"
> roster$grade[score < y[4]] <- "F"
> roster
             Student Math Science English  score grade
1         John Davis  502       95      25  0.559     B
2    Angela Williams  600       99      22  0.924     A
3   Bullwinkle Moose  412       80      18 -0.857     D
4        David Jones  358       82      15 -1.162     F
5  Janice Markhammer  495       75      20 -0.629     D
6     Cheryl Cushing  512       85      28  0.353     C
7     Reuven Ytzrhak  410       80      15 -1.048     F
8          Greg Knox  625       95      30  1.338     A
```

```
9       Joel England  573      89      27  0.698     B
10      Mary Rayburn  522      86      18 -0.177     C
```

Step 6. You'll use the `strsplit()` function to break student names into first name and last name at the space character. Applying `strsplit()` to a vector of strings returns a list:

```
> name <- strsplit((roster$Student), " ")
> name

[[1]]
[1] "John"  "Davis"

[[2]]
[1] "Angela"   "Williams"

[[3]]
[1] "Bullwinkle" "Moose"

[[4]]
[1] "David" "Jones"

[[5]]
[1] "Janice"     "Markhammer"

[[6]]
[1] "Cheryl"  "Cushing"

[[7]]
[1] "Reuven"  "Ytzrhak"

[[8]]
[1] "Greg" "Knox"

[[9]]
[1] "Joel"    "England"

[[10]]
[1] "Mary"    "Rayburn"
```

Step 7. You can use the `sapply()` function to take the first element of each component and put it in a firstname vector, and the second element of each component and put it in a lastname vector. `"["` is a function that extracts part of an object—here the first or second component of the list `name`. You'll use `cbind()` to add them to the roster. Because you no longer need the student variable, you'll drop it (with the –1 in the roster index).

```
> Firstname <- sapply(name, "[", 1)
> Lastname <- sapply(name, "[", 2)
> roster <- cbind(Firstname, Lastname, roster[,-1])
> roster
    Firstname   Lastname Math Science English  score grade
1        John      Davis  502      95      25  0.559     B
2      Angela   Williams  600      99      22  0.924     A
3  Bullwinkle      Moose  412      80      18 -0.857     D
```

```
4       David      Jones  358      82      15 -1.162     F
5      Janice Markhammer  495      75      20 -0.629     D
6      Cheryl     Cushing  512      85      28  0.353     C
7      Reuven     Ytzrhak  410      80      15 -1.048     F
8        Greg        Knox  625      95      30  1.338     A
9        Joel     England  573      89      27  0.698     B
10       Mary     Rayburn  522      86      18 -0.177     C
```

Step 8. Finally, you can sort the dataset by first and last name using the `order()` function:

```
> roster[order(Lastname,Firstname),]
    Firstname   Lastname  Math  Science  English  score   grade
6      Cheryl     Cushing  512       85       28     0.35     C
1        John       Davis  502       95       25     0.56     B
9        Joel     England  573       89       27     0.70     B
4       David       Jones  358       82       15    -1.16     F
8        Greg        Knox  625       95       30     1.34     A
5      Janice Markhammer  495       75       20    -0.63     D
3  Bullwinkle      Moose  412       80       18    -0.86     D
10       Mary     Rayburn  522       86       18    -0.18     C
2      Angela   Williams  600       99       22     0.92     A
7      Reuven     Ytzrhak  410       80       15    -1.05     F
```

Voilà! Piece of cake!

There are many other ways to accomplish these tasks, but this code helps capture the flavor of these functions. Now it's time to look at control structures and user-written functions.

5.4 Control flow

In the normal course of events, the statements in an R program are executed sequentially from the top of the program to the bottom. But there are times that you'll want to execute some statements repetitively, while only executing other statements if certain conditions are met. This is where control-flow constructs come in.

R has the standard control structures you'd expect to see in a modern programming language. First you'll go through the constructs used for conditional execution, followed by the constructs used for looping.

For the syntax examples throughout this section, keep the following in mind:

- *statement* is a single R statement or a compound statement (a group of R statements enclosed in curly braces { } and separated by semicolons).
- *cond* is an expression that resolves to true or false.
- *expr* is a statement that evaluates to a number or character string.
- *seq* is a sequence of numbers or character strings.

After we discuss control-flow constructs, you'll learn how to write your functions.

5.4.1 Repetition and looping

Looping constructs repetitively execute a statement or series of statements until a condition isn't true. These include the `for` and `while` structures.

FOR

The `for` loop executes a statement repetitively until a variable's value is no longer contained in the sequence `seq`. The syntax is

```
for (var in seq) statement
```

In this example

```
for (i in 1:10)  print("Hello")
```

the word Hello is printed 10 times.

WHILE

A `while` loop executes a statement repetitively until the condition is no longer true. The syntax is

```
while (cond) statement
```

In a second example, the code

```
i <- 10
while (i > 0) {print("Hello"); i <- i - 1}
```

once again prints the word Hello 10 times. Make sure that the statements inside the brackets modify the `while` condition so that sooner or later it's no longer true—otherwise the loop will never end! In the previous example, the statement

```
i <- i - 1
```

subtracts 1 from object `i` on each loop, so that after the tenth loop it's no longer larger than 0. If you instead added 1 on each loop, R would never stop saying Hello. This is why `while` loops can be more dangerous than other looping constructs.

Looping in R can be inefficient and time consuming when you're processing the rows or columns of large datasets. Whenever possible, it's better to use R's built-in numerical and character functions in conjunction with the `apply` family of functions.

5.4.2 Conditional execution

In conditional execution, a statement or statements are only executed if a specified condition is met. These constructs include `if-else`, `ifelse`, and `switch`.

IF-ELSE

The `if-else` control structure executes a statement if a given condition is true. Optionally, a different statement is executed if the condition is false. The syntax is

```
if (cond) statement
if (cond) statement1 else statement2
```

Here are examples:

```
if (is.character(grade)) grade <- as.factor(grade)
if (!is.factor(grade)) grade <- as.factor(grade) else print("Grade already
    is a factor")
```

In the first instance, if grade is a character vector, it's converted into a factor. In the second instance, one of two statements is executed. If grade isn't a factor (note the `!` symbol), it's turned into one. If it is a factor, then the message is printed.

IFELSE

The `ifelse` construct is a compact and vectorized version of the `if-else` construct. The syntax is

```
ifelse(cond, statement1, statement2)
```

The first statement is executed if *cond* is `TRUE`. If *cond* is `FALSE`, the second statement is executed. Here are examples:

```
ifelse(score > 0.5, print("Passed"), print("Failed"))
outcome <- ifelse (score > 0.5, "Passed", "Failed")
```

Use `ifelse` when you want to take a binary action or when you want to input and output vectors from the construct.

SWITCH

`switch` chooses statements based on the value of an expression. The syntax is

```
switch(expr, ...)
```

where . . . represents statements tied to the possible outcome values of *expr*. It's easiest to understand how `switch` works by looking at the example in the following listing.

Listing 5.7 A `switch` example

```
> feelings <- c("sad", "afraid")
> for (i in feelings)
    print(
      switch(i,
        happy  = "I am glad you are happy",
        afraid = "There is nothing to fear",
        sad    = "Cheer up",
        angry  = "Calm down now"
      )
    )

[1] "Cheer up"
[1] "There is nothing to fear"
```

This is a silly example but shows the main features. You'll learn how to use `switch` in user-written functions in the next section.

5.5 User-written functions

One of R's greatest strengths is the user's ability to add functions. In fact, many of the functions in R are functions of existing functions. The structure of a function looks like this:

```
myfunction <- function(arg1, arg2, ... ){
  statements
  return(object)
}
```

Objects in the function are local to the function. The object returned can be any data type, from scalar to list. Let's take a look at an example.

Say you'd like to have a function that calculates the central tendency and spread of data objects. The function should give you a choice between parametric (mean and standard deviation) and nonparametric (median and median absolute deviation) statistics. The results should be returned as a named list. Additionally, the user should have the choice of automatically printing the results, or not. Unless otherwise specified, the function's default behavior should be to calculate parametric statistics and not print the results. One solution is given in the following listing.

Listing 5.8 `mystats()`: a user-written function for summary statistics

```
mystats <- function(x, parametric=TRUE, print=FALSE) {
  if (parametric) {
    center <- mean(x); spread <- sd(x)
  } else {
    center <- median(x); spread <- mad(x)
  }
  if (print & parametric) {
    cat("Mean=", center, "\n", "SD=", spread, "\n")
  } else if (print & !parametric) {
    cat("Median=", center, "\n", "MAD=", spread, "\n")
  }
  result <- list(center=center, spread=spread)
  return(result)
}
```

To see your function in action, first generate some data (a random sample of size 500 from a normal distribution):

```
set.seed(1234)
x <- rnorm(500)
```

After executing the statement

```
y <- mystats(x)
```

`y$center` will contain the mean (0.00184) and `y$spread` will contain the standard deviation (1.03). No output is produced. If you execute the statement

```
y <- mystats(x, parametric=FALSE, print=TRUE)
```

`y$center` will contain the median (–0.0207) and `y$spread` will contain the median absolute deviation (1.001). In addition, the following output is produced:

```
Median= -0.0207
MAD= 1
```

Next, let's look at a user-written function that uses the `switch` construct. This function gives the user a choice regarding the format of today's date. Values that are assigned to parameters in the function declaration are taken as defaults. In the `mydate()` function, `long` is the default format for dates if `type` isn't specified:

```
mydate <- function(type="long") {
  switch(type,
    long =  format(Sys.time(), "%A %B %d %Y"),
    short = format(Sys.time(), "%m-%d-%y"),
    cat(type, "is not a recognized type\n")
   )
}
```

Here's the function in action:

```
> mydate("long")
[1] "Thursday December 02 2010"
> mydate("short")
[1] "12-02-10"
> mydate()
[1] "Thursday December 02 2010"
> mydate("medium")
medium is not a recognized type
```

Note that the `cat()` function is only executed if the entered type doesn't match `"long"` or `"short"`. It's usually a good idea to have an expression that catches user-supplied arguments that have been entered incorrectly.

Several functions are available that can help add error trapping and correction to your functions. You can use the function `warning()` to generate a warning message, `message()` to generate a diagnostic message, and `stop()` to stop execution of the current expression and carry out an error action. See each function's online help for more details.

> **TIP** Once you start writing functions of any length and complexity, access to good debugging tools becomes important. R has a number of useful built-in functions for debugging, and user-contributed packages are available that provide additional functionality. An excellent resource on this topic is Duncan Murdoch's "Debugging in R" (http://www.stats.uwo.ca/faculty/murdoch/software/debuggingR).

After creating your own functions, you may want to make them available in every session. Appendix B describes how to customize the R environment so that user-written functions are loaded automatically at startup. We'll look at additional examples of user-written functions in chapters 6 and 8.

You can accomplish a great deal using the basic techniques provided in this section. If you'd like to explore the subtleties of function writing, or want to write professional-level code that you can distribute to others, I recommend two excellent books that you'll find in the References section at the end of this book: Venables & Ripley (2000) and Chambers (2008). Together, they provide a significant level of detail and breadth of examples.

Now that we've covered user-written functions, we'll end this chapter with a discussion of data aggregation and reshaping.

5.6 Aggregation and restructuring

R provides a number of powerful methods for aggregating and reshaping data. When you aggregate data, you replace groups of observations with summary statistics based on those observations. When you reshape data, you alter the structure (rows and columns) determining how the data is organized. This section describes a variety of methods for accomplishing these tasks.

In the next two subsections, we'll use the `mtcars` data frame that's included with the base installation of R. This dataset, extracted from *Motor Trend* magazine (1974), describes the design and performance characteristics (number of cylinders, displacement, horsepower, mpg, and so on) for 34 automobiles. To learn more about the dataset, see `help(mtcars)`.

5.6.1 Transpose

The transpose (reversing rows and columns) is perhaps the simplest method of reshaping a dataset. Use the `t()` function to transpose a matrix or a data frame. In the latter case, row names become variable (column) names. An example is presented in the next listing.

Listing 5.9 Transposing a dataset

```
> cars <- mtcars[1:5,1:4]
> cars
                   mpg cyl disp  hp
Mazda RX4         21.0   6  160 110
Mazda RX4 Wag     21.0   6  160 110
Datsun 710        22.8   4  108  93
Hornet 4 Drive    21.4   6  258 110
Hornet Sportabout 18.7   8  360 175
> t(cars)
     Mazda RX4 Mazda RX4 Wag Datsun 710 Hornet 4 Drive Hornet Sportabout
mpg         21            21       22.8           21.4              18.7
cyl          6             6        4.0            6.0               8.0
disp       160           160      108.0          258.0             360.0
hp         110           110       93.0          110.0             175.0
```

Listing 5.9 uses a subset of the `mtcars` dataset in order to conserve space on the page. You'll see a more flexible way of transposing data when we look at the `shape` package later in this section.

5.6.2 Aggregating data

It's relatively easy to collapse data in R using one or more `by` variables and a defined function. The format is

```
aggregate(x, by, FUN)
```

where *x* is the data object to be collapsed, *by* is a list of variables that will be crossed to form the new observations, and *FUN* is the scalar function used to calculate summary statistics that will make up the new observation values.

As an example, we'll aggregate the `mtcars` data by number of cylinders and gears, returning means on each of the numeric variables (see the next listing).

Listing 5.10 Aggregating data

```
> options(digits=3)
> attach(mtcars)
> aggdata <-aggregate(mtcars, by=list(cyl,gear), FUN=mean, na.rm=TRUE)
> aggdata
  Group.1 Group.2  mpg cyl disp  hp drat   wt qsec  vs   am gear carb
1       4       3 21.5   4  120  97 3.70 2.46 20.0 1.0 0.00    3 1.00
2       6       3 19.8   6  242 108 2.92 3.34 19.8 1.0 0.00    3 1.00
3       8       3 15.1   8  358 194 3.12 4.10 17.1 0.0 0.00    3 3.08
4       4       4 26.9   4  103  76 4.11 2.38 19.6 1.0 0.75    4 1.50
5       6       4 19.8   6  164 116 3.91 3.09 17.7 0.5 0.50    4 4.00
6       4       5 28.2   4  108 102 4.10 1.83 16.8 0.5 1.00    5 2.00
7       6       5 19.7   6  145 175 3.62 2.77 15.5 0.0 1.00    5 6.00
8       8       5 15.4   8  326 300 3.88 3.37 14.6 0.0 1.00    5 6.00
```

In these results, Group.1 represents the number of cylinders (4, 6, or 8) and Group.2 represents the number of gears (3, 4, or 5). For example, cars with 4 cylinders and 3 gears have a mean of 21.5 miles per gallon (mpg).

When you're using the `aggregate()` function, the `by` variables must be in a list (even if there's only one). You can declare a custom name for the groups from within the list, for instance, using `by=list(Group.cyl=cyl, Group.gears=gear)`. The function specified can be any built-in or user-provided function. This gives the aggregate command a great deal of power. But when it comes to power, nothing beats the `reshape` package.

5.6.3 The reshape package

The `reshape` package is a tremendously versatile approach to both restructuring and aggregating datasets. Because of this versatility, it can be a bit challenging to learn. We'll go through the process slowly and use a small dataset so that it's clear what's happening. Because `reshape` isn't included in the standard installation of R, you'll need to install it one time, using `install.packages("reshape")`.

Basically, you'll "melt" data so that each row is a unique ID-variable combination. Then you'll "cast" the melted data into any shape you desire. During the cast, you can aggregate the data with any function you wish.

The dataset you'll be working with is shown in table 5.8.

Table 5.8 The original dataset (`mydata`)

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

In this dataset, the *measurements* are the values in the last two columns (5, 6, 3, 5, 6, 1, 2, and 4). Each measurement is uniquely identified by a combination of ID variables (in this case ID, Time, and whether the measurement is on X1 or X2). For example, the measured value 5 in the first row is

uniquely identified by knowing that it's from observation (ID) 1, at Time 1, and on variable X1.

MELTING

When you melt a dataset, you restructure it into a format where each measured variable is in its own row, along with the ID variables needed to uniquely identify it. If you melt the data from table 5.8, using the following code

```
library(reshape)
md <- melt(mydata, id=(c("id", "time")))
```

you end up with the structure shown in table 5.9.

Note that you must specify the variables needed to uniquely identify each measurement (ID and Time) and that the variable indicating the measurement variable names (X1 or X2) is created for you automatically.

Now that you have your data in a melted form, you can recast it into any shape, using the `cast()` function.

Table 5.9 The melted dataset

ID	Time	Variable	Value
1	1	X1	5
1	2	X1	3
2	1	X1	6
2	2	X1	2
1	1	X2	6
1	2	X2	5
2	1	X2	1
2	2	X2	4

CASTING

The `cast()` function starts with melted data and reshapes it using a formula that you provide and an (optional) function used to aggregate the data. The format is

```
newdata <- cast(md, formula, FUN)
```

where *md* is the melted data, *formula* describes the desired end result, and *FUN* is the (optional) aggregating function. The formula takes the form

```
rowvar1 + rowvar2 + … ~ colvar1 + colvar2 + …
```

In this formula, *rowvar1 + rowvar2 + …* define the set of crossed variables that define the rows, and *colvar1 + colvar2 + …* define the set of crossed variables that define the columns. See the examples in figure 5.1.

Because the formulas on the right side (d, e, and f) don't include a function, the data is reshaped. In contrast, the examples on the left side (a, b, and c) specify the mean as an aggregating function. Thus the data are not only reshaped but aggregated as well. For example, (a) gives the means on X1 and X2 averaged over time for each observation. Example (b) gives the mean scores of X1 and X2 at Time 1 and Time 2, averaged over observations. In (c) you have the mean score for each observation at Time 1 and Time 2, averaged over X1 and X2.

As you can see, the flexibility provided by the `melt()` and `cast()` functions is amazing. There are many times when you'll have to reshape or aggregate your data prior to analysis. For example, you'll typically need to place your data in what's called

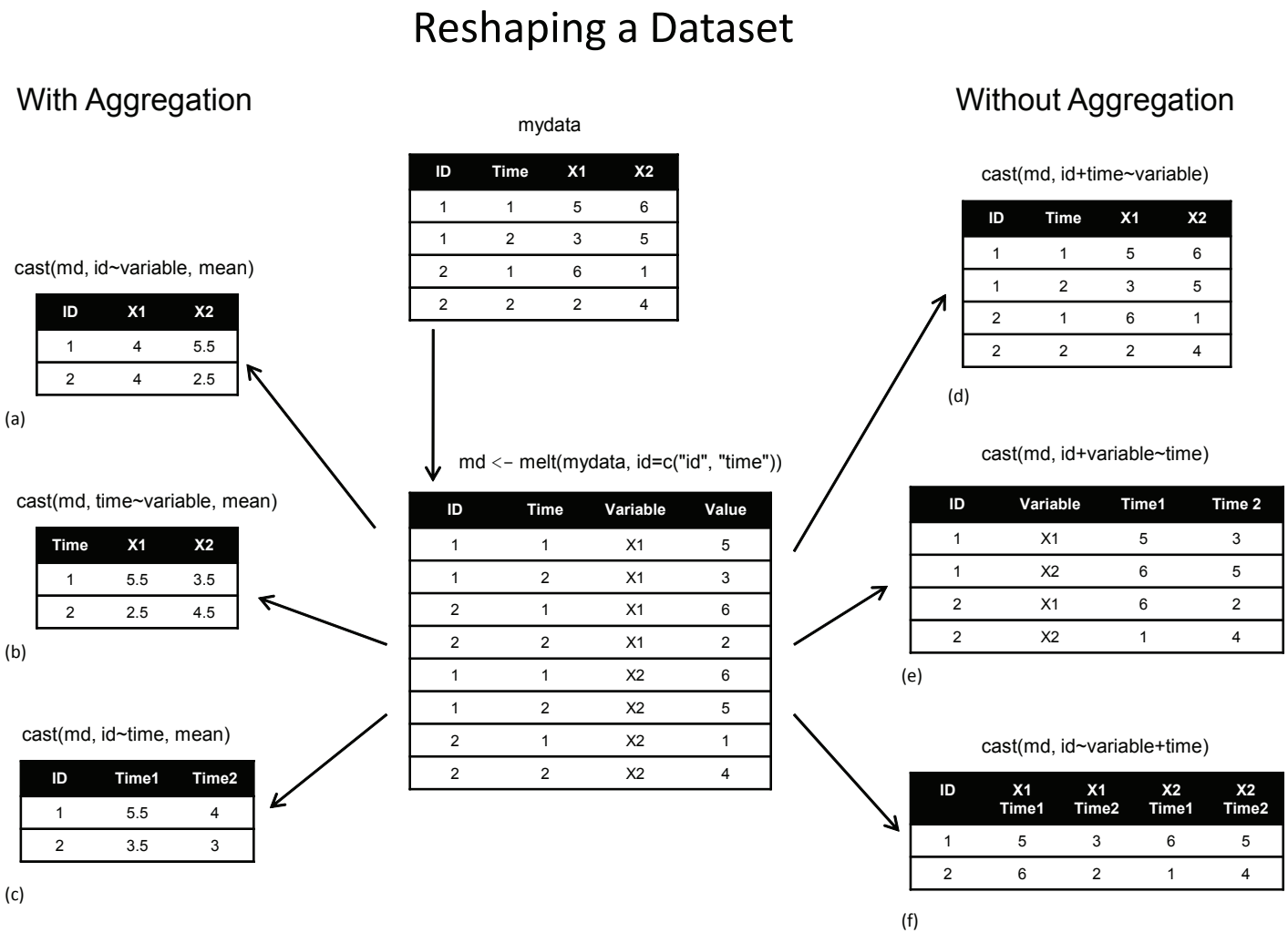


Figure 5.1 **Reshaping data with the `melt()` and `cast()` functions**

"long format" resembling table 5.9 when analyzing repeated measures data (data where multiple measures are recorded for each observation). See section 9.6 for an example.

5.7 Summary

This chapter reviewed dozens of mathematical, statistical, and probability functions that are useful for manipulating data. We saw how to apply these functions to a wide range of data objects, including vectors, matrices, and data frames. We learned to use control-flow constructs for looping and branching to execute some statements repetitively and execute other statements only when certain conditions are met. You then had a chance to write your own functions and apply them to data. Finally, we explored ways of collapsing, aggregating, and restructuring your data.

Now that you've gathered the tools you need to get your data into shape (no pun intended), we're ready to bid part 1 goodbye and enter the exciting world of data analysis! In upcoming chapters, we'll begin to explore the many statistical and graphical methods available for turning data into information.