Study Material. Do not distribute.

9 Functions

Functions are the R objects that evaluate a set of input arguments and return an output value. This chapter explains how to create and use functions in R.

The Function Keyword

In R, function objects are defined with this syntax:

```
function(arguments) body
```

where ***arguments*** is a set of symbol names (and, optionally, default values) that will be defined within the body of the function, and ***body*** is an R expression. Typically, the body is enclosed in curly braces, but it does not have to be if the body is a single expression. For example, the following two definitions are equivalent:

```
f <- function(x,y) x + y
f <- function(x,y) {x + y}
```

Arguments

A function definition in R includes the names of arguments. Optionally, it may include default values. If you specify a default value for an argument, then the argument is considered optional:

```
> f <- function(x, y) {x + y}
> f(1,2)
[1] 3
> g <- function(x, y=10) {x + y}
> g(1)
[1] 11
```

Study Material. Do not distribute.

If you do not specify a default value for an argument, and you do not specify a value when calling the function, you will get an error if the function attempts to use the argument:[1]

```
> f(1)
Error in f(1) :
  element 2 is empty;
   the part of the args list of '+' being evaluated was:
   (x, y)
```

In a function call, you may override the default value:

```
> g(1, 2)
[1] 3
```

In R, it is often convenient to specify a variable-length argument list. You might want to pass extra arguments to another function, or you may want to write a function that accepts a variable number of arguments. To do this in R, you specify an ellipsis (`...`) in the arguments to the function.[2]

As an example, let's create a function that prints the first argument and then passes all the other arguments to the `summary` function. To do this, we will create a function that takes one argument: `x`. The arguments specification also includes an ellipsis to indicate that the function takes other arguments. We can then call the `summary` function with the ellipsis as its argument:

```
> v <- c(sqrt(1:100))
> f <- function(x,...) {print(x); summary(...)}
> f("Here is the summary for v.", v, digits=2)
[1] "Here is the summary for v."
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     5.1     7.1     6.7     8.7    10.0
```

Notice that all of the arguments after `x` were passed to `summary`.

1. Note that you will get an error only if you try to use the uninitialized argument within the function; you could easily write a function that simply doesn't reference the argument, and it will work fine. Additionally, there are other ways to check whether an argument has been initialized from inside the body of a function. For example, the following function works identically to the function `g` shown above (which included a default value for `y` in its definition):

```
> h <- function(x,y) {
+   args <- as.list(match.call())
+   if (is.null(args$y)) {
+     y <- 10
+   }
+   x + y
+ }
```

In practice, you should specify default values in the function signature to make your functions as clear and easy to read as possible.

2. You might remember from Chapter 7 that "`...`" is a special type of object in R. The only place you can manipulate this object is inside the body of a function. In this context, it means "all the other arguments for the function."

Study Material. Do not distribute.

It is also possible to read the arguments from the variable-length argument list. To do this, you can convert the object `...` to a list within the body of the function. As an example, let's create a function that simply sums all its arguments:

```
> addemup <- function(x,...) {
+    args <- list(...)
+    for (a in args) x <- x + a
+    x
+ }
> addemup(1, 1)
[1] 2
> addemup(1, 2, 3, 4, 5)
[1] 15
```

You can also directly refer to items within the list `...` through the variables `..1`, `..2`, to `..9`. Use ..1 for the first item, ..2 for the second, and so on. Named arguments are valid symbols within the body of the function. For more information about the scope within which variables are defined, see Chapter 8.

Return Values

In an R function, you may use the `return` function to specify the value returned by the function. For example:

```
> f <- function(x) {return(x^2 + 3)}
> f(3)
[1] 12
```

However, R will simply return the last evaluated expression as the result of a function. So it is common to omit the `return` statement:

```
> f <- function(x) {x^2 + 3}
> f(3)
[1] 12
```

In some cases, an explicit return value may lead to cleaner code.

Functions as Arguments

Many functions in R can take other functions as arguments. For example, many modeling functions accept an optional argument that specifies how to handle missing values; this argument is usually a function for processing the input data.

As an example of a function that takes another function as an argument, let's look at `sapply`. The `sapply` function iterates through each element in a vector, applying another function to each element in the vector and returning the results. Here is a simple example:

```
> a <- 1:7
> sapply(a, sqrt)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
```

Study Material. Do not distribute.

This is a toy example; you could have calculated the same quantity with the expression `sqrt(1:7)`. However, there are many useful functions that don't work properly on a vector with more than one element; `sapply` provides a simple way to extend such a function to work on a vector. Related functions allow you to summarize every element in a data structure or to perform more complicated calculations. See "Summarizing Functions" on page 190 for information on related functions.

Anonymous Functions

So far, we've mostly seen named functions in R. However, because functions are just objects in R, it is possible to create functions that do not have names. These are called *anonymous functions*. Anonymous functions are usually passed as arguments to other functions. If you're new to functional languages, this concept might seem strange, so let's start with a very simple example.

We will define a function that takes another function as its argument and then applies that function to the number 3. Let's call the function `apply.to.three`, and we will call the argument `f`:

```
> apply.to.three <- function(f) {f(3)}
```

Now let's call `apply.to.three` with an anonymous function assigned to argument `f`. As an example, let's create a simple function that takes one argument and multiplies that argument by 7:

```
> apply.to.three(function(x) {x * 7})
[1] 21
```

Here's how this works. When the R interpreter evaluates the expression `apply.to.three(function(x) {x * 7})`, it assigns the argument `f` to the anonymous function `function(x) {x * 7}`. The interpreter then begins evaluating the expression `f(3)`. The interpreter assigns `3` to the argument `x` for the anonymous function. Finally, the interpreter evaluates the expression `3 * 7` and returns the result.

Anonymous functions are a very powerful tool used in many places in R. Above, we used the `sapply` function to apply a named function to every element in an array. You can also pass an anonymous function as an argument to `sapply`:

```
> a <- c(1, 2, 3, 4, 5)
> sapply(a, function(x) {x + 1})
[1] 2 3 4 5 6
```

This family of functions is a good alternative to control structures. *Control structures* are language features like if-then statements, loops, and go-to statements. For example, suppose that you had a vector of numerical values and wanted to calculate the square of each element. You could do this using a loop:

```
> v <- 1:20
> w <- NULL
> for (i in 1:length(v)) {w[i] <- v[i]^2}
> w
[1] 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400
```

However, you can do the same thing using an "apply" statement like this:

Study Material. Do not distribute.

```
> v <- 1:20
> w <- sapply(v, function(i) {i^2})
> w
[1] 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400
```

I think it's more clear what the second code snippet does: it applies the function to each element in v. (Additionally, the apply function will be faster. See "Lookup Performance in R" on page 509 for more information.

By the way, it is possible to define an anonymous function and apply it directly to an argument. Here's an example:

```
> (function(x) {x+1})(1)
[1] 2
```

Notice that the function object needs to be enclosed in parentheses. This is because function calls, expressions of the form `f(arguments)`, have very high precedence in R.[3]

Properties of Functions

R includes a set of functions for getting more information about function objects. To see the set of arguments accepted by a function, use the `args` function. The `args` function returns a function object with `NULL` as the body. Here are a few examples:

```
> args(sin)
function (x)
NULL
> args(`?`)
function (e1, e2)
NULL
> args(args)
function (name)
NULL
> args(lm)
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
NULL
```

3. If you omit the parentheses in this example, you will not initially get an error:

```
> function(x) {x+1}(1)
function(x) {x+1}(1)
```

This is because you will have created an object that is a function taking one argument (`x`) with the body `{x+1}(1)`. There is no error generated because the body is not evaluated. If you were to assign this object to a symbol (so that you can easily apply it to an argument and see what it does), you will find that this function attempts to call a function returned by evaluating the expression `{x + 1}`. In order not to get an error or an input of class c, you would need to register a generic function that took as input an object of class c (`x` in this expression) and a numerical value (1 in this expression) and returned a function object. So omitting the parentheses is not wrong; it is a valid R expression. However, this is almost certainly not what you meant to write.

Study Material. Do not distribute.

If you would like to manipulate the list of arguments with R code, then you may find the `formals` function more useful. The `formals` function will return a pairlist object, with a pair for every argument. The name of each pair will correspond to each argument name in the function. When a default value is defined, the corresponding value in the pairlist will be set to that value. When no default is defined, the value will be `NULL`. The `formals` function is available only for functions written in R (objects of type `closure`) and not for built-in functions.

Here is a simple example of using `formals` to extract information about the arguments to a function:

```
> f <- function(x, y=1, z=2) {x + y + z}
> f.formals <- formals(f)
> f.formals
$x


$y
[1] 1

$z
[1] 2

> f.formals$x

> f.formals$y
[1] 1
> f.formals$z
[1] 2
```

You may also use `formals` on the left-hand side of an assignment statement to change the formal argument for a function. For example:

```
> f.formals$y <- 3
> formals(f) <- f.formals
> args(f)
function (x, y = 3, z = 2)
NULL
```

R provides a convenience function called `alist` to construct an argument list. You simply specify the argument list as if you were defining a function. (Note that for an argument with no default, you do not need to include a value but still need to include the equals sign.)

```
> f <- function(x, y=1, z=2) {x + y + z}
> formals(f) <- alist(x=, y=100, z=200)
> f
function (x, y = 100, z = 200)
{
    x + y + z
}
```

R provides a similar function called `body` that can be used to return the body of a function:

Study Material. Do not distribute.

```
> body(f)
{
    x + y + z
}
```

Like the `formals` function, the `body` function may be used on the left-hand side of an assignment statement:

```
> f
function (x, y = 3, z = 2)
{
    x + y + z
}
> body(f) <- expression({x * y * z})
> f
function (x, y = 3, z = 2)
{
    x * y * z
}
```

Note that the body of a function has type `expression`, so when you assign a new value it must have the type `expression`.

Argument Order and Named Arguments

When you specify a function in R, you assign a name to each argument in the function. Inside the body of the function, you can access the arguments by name. For example, consider the following function definition:

```
> addTheLog <- function(first, second) {first + log(second)}
```

This function takes two arguments, called `first` and `second`. Inside the body of the function, you can refer to the arguments by these names.

When you call a function in R, you can specify the arguments in three different ways (in order of priority):

1. Exact names. The arguments will be assigned to *full* names explicitly given in the argument list. Full argument names are matched first:

   ```
   > addTheLog(second=exp(4), first=1)
   [1] 5
   ```

2. Partially matching names. The arguments will be assigned to *partial* names explicitly given in the arguments list:

   ```
   > addTheLog(s=exp(4), f=1)
   [1] 5
   ```

3. Argument order. The arguments will be assigned to names in the order in which they were given:

   ```
   > addTheLog(1, exp(4))
   [1] 5
   ```

Study Material. Do not distribute.

When you are using generic functions, you cannot specify the argument name of the object on which the generic function is being called. You can still specify names for other arguments.

When possible, it's a good practice to use exact argument names. Specifying full argument names does require extra typing, but it makes your code easier to read and removes ambiguity.

Partial names are a deprecated feature because they can lead to confusion. As an example, consider the following function:

```
> f <- function(arg1=10, arg2=20) {
+     print(paste("arg1:", arg1))
+     print(paste("arg2:", arg2))
+ }
```

When you call this function with one ambiguous argument, it will cause an error:

```
> f(arg=1)
Error in f(arg = 1) : argument 1 matches multiple formal arguments
```

However, when you specify two arguments, the ambiguous argument could refer to either of the other arguments:

```
> f(arg=1, arg2=2)
[1] "arg1: 1"
[1] "arg2: 2"
> f(arg=1, arg1=2)
[1] "arg1: 2"
[1] "arg2: 1"
```

Side Effects

All functions in R return a value. Some functions also do other things: change variables in the current environment (or in other environments), plot graphics, load or save files, or access the network. These operations are called *side effects*.

Changes to Other Environments

We have already seen some examples of functions with side effects. In Chapter 8, we showed how to directly access symbols and objects in an environment (or in parent environments). We also showed how to access objects on the call stack.

An important function that causes side effects is the `<<-` operator. This operator takes the following form: *`var <<- value`*. This operator will cause the interpreter to first search through the current environment to find the symbol *`var`*. If the interpreter does not find the symbol *`var`* in the current environment, then the interpreter will next search through the parent environment. The interpreter will recursively search through environments until it either finds the symbol *`var`* or reaches the global environment. If it reaches the global environment before the symbol *`var`* is found, then R will assign *`value`* to *`var`* in the global environment.

Here is an example that compares the behavior of the `<-` assignment operator and the `<<-` operator:

Study Material. Do not distribute.

```
> x
Error: object "x" not found
> doesnt.assign.x <- function(i) {x <- i}
> doesnt.assign.x(4)
> x
Error: object "x" not found
> assigns.x <- function(i) {x <<- i}
> assigns.x(4)
> x
[1] 4
```

Input/Output

R does a lot of stuff, but it's not completely self-contained. If you're using R, you'll probably want to load data from external files (or from the Internet) and save data to files. These input/output (I/O) actions are side effects, because they do things other than just return an object. We'll talk about these functions extensively in Chapter 11.

Graphics

Graphics functions are another example of side effects in R. Graphics functions may return objects, but they also plot graphics (either on screen or to files). We'll talk about these functions in Chapters 13 and 14.

Study Material. Do not distribute.

Study Material. Do not distribute.

12
Preparing Data

Back in my freshman year of college, I was planning to be a biochemist. I spent hours and hours in the lab: mixing chemicals in test tubes, putting samples in different machines, and analyzing the results. Over time, I grew frustrated because I found myself spending weeks in the lab doing manual work and just a few minutes planning experiments or analyzing results. After a year, I gave up on chemistry and became a computer scientist, thinking that I would spend less time on preparation and testing and more time on analysis.

Unfortunately for me, I chose to do data mining work professionally. Everyone loves building models, drawing charts, and playing with cool algorithms. Unfortunately, most of the time you spend on data analysis projects is spent on preparing data for analysis. I'd estimate that 80% of the effort on a typical project is spent on finding, cleaning, and preparing data for analysis. Less than 5% of the effort is devoted to analysis. (The rest of the time is spent on writing up what you did.)

If you're new to data analysis, you're probably wondering what the big deal is about preparing data. Suppose that you are getting some data off of your company's web servers, or out of a financial database, or from electronic patient records. It all came from computers, so it's perfect, right?

In practice, data is almost never stored in the right form for analysis. Even when data is in the right form, there are often surprises in the data. It takes a lot of work to pull together a usable data set. This chapter explains how to prepare data for analysis with R.

Combining Data Sets

Let's start with one of the most common obstacles to data analysis: working with data that's stored in two different places. For example, suppose that you wanted to look at batting statistics for baseball players by age. In most baseball data sources (like the Baseball Databank data), player information (like ages) is kept in different files from performance data (like batting statistics). So you would need to combine

Study Material. Do not distribute.

two files to do this analysis. This section discusses several tools in R used for combining data sets.

Pasting Together Data Structures

R provides several functions that allow you to paste together multiple data structures into a single structure.

Paste

The simplest of these functions is `paste`. The `paste` function allows you to concatenate multiple character vectors into a single vector. (If you concatenate a vector of another type, it will be coerced to a character vector first.)

```
> x <- c("a", "b", "c", "d", "e")
> y <- c("A", "B", "C", "D", "E")
> paste(x,y)
[1] "a A" "b B" "c C" "d D" "e E"
```

By default, values are separated by a space; you can specify another separator (or none at all) with the `sep` argument:

```
> paste(x, y, sep="-")
[1] "a-A" "b-B" "c-C" "d-D" "e-E"
```

If you would like all of values in the returned vector to be concatenated with one another (to return just a single value), then specify a value for the `collapse` argument. The value of `collapse` will be used as the separator in this value:

```
> paste(x, y, sep="-", collapse="#")
[1] "a-A#b-B#c-C#d-D#e-E"
```

rbind and cbind

Sometimes, you would like to bind together multiple data frames or matrices. You can do this with the `rbind` and `cbind` functions. The `cbind` function will combine objects by adding columns. You can picture this as combining two tables horizontally. As an example, let's start with the data frame for the top five salaries in the NFL in 2008:[1]

```
> top.5.salaries
  name.last name.first     team position   salary
1   Manning     Peyton    Colts       QB 18700000
2     Brady        Tom Patriots       QB 14626720
3    Pepper     Julius Panthers       DE 14137500
4    Palmer     Carson  Bengals       QB 13980000
5   Manning        Eli   Giants       QB 12916666
```

Now let's create a new data frame with two more columns (a year and a rank):

```
> year <- c(2008, 2008, 2008, 2008, 2008)
> rank <- c(1, 2, 3, 4, 5)
```

1. Salary data is from *http://sportsillustrated.cnn.com/football/nfl/salaries/2008/all.html*. The salary numbers are cap numbers, not cash salaries.

Study Material. Do not distribute.

```
> more.cols <- data.frame(year, rank)
> more.cols
  year rank
1 2008    1
2 2008    2
3 2008    3
4 2008    4
5 2008    5
```

Finally, let's put together these two data frames:

```
> cbind(top.5.salaries, more.cols)
  name.last name.first     team position   salary year rank
1   Manning     Peyton    Colts       QB 18700000 2008    1
2     Brady        Tom Patriots       QB 14626720 2008    2
3    Pepper     Julius Panthers       DE 14137500 2008    3
4    Palmer     Carson  Bengals       QB 13980000 2008    4
5   Manning        Eli   Giants       QB 12916666 2008    5
```

The `rbind` function will combine objects by adding rows. You can picture this as combining two tables vertically.

As an example, suppose that you had a data frame with the top five salaries (as shown above) and a second data frame with the next three salaries:

```
> top.5.salaries
  name.last name.first     team position   salary
1   Manning     Peyton    Colts       QB 18700000
2     Brady        Tom Patriots       QB 14626720
3    Pepper     Julius Panthers       DE 14137500
4    Palmer     Carson  Bengals       QB 13980000
5   Manning        Eli   Giants       QB 12916666
> next.three
  name.last name.first    team position   salary
6     Favre      Brett Packers       QB 12800000
7    Bailey      Champ Broncos       CB 12690050
8  Harrison     Marvin   Colts       WR 12000000
```

You could combine these into a single data frame using the `rbind` function:

```
> rbind(top.5.salaries, next.three)
  name.last name.first     team position   salary
1   Manning     Peyton    Colts       QB 18700000
2     Brady        Tom Patriots       QB 14626720
3    Pepper     Julius Panthers       DE 14137500
4    Palmer     Carson  Bengals       QB 13980000
5   Manning        Eli   Giants       QB 12916666
6     Favre      Brett  Packers       QB 12800000
7    Bailey      Champ  Broncos       CB 12690050
8  Harrison     Marvin    Colts       WR 12000000
```

An extended example

To show how to fetch and combine together data and build a data frame for analysis, we'll use an example from the previous chapter: stock quotes. Yahoo! Finance allows you to download CSV files with stock quotes for a single ticker.

Study Material. Do not distribute.

Suppose that you wanted a single data set with stock quotes for multiple securities (say, the 30 stocks in the Dow Jones Industrial Average). You would need a way to bind together the data returned by the query into a single data set. Let's write a function that can return historical stock quotes for multiple securities in a single data frame. First, let's write a function that assembles the URL for the CSV file and then fetches a data frame with the contents.

Here is what this function will do. First, it will define the URL. (I determined the format of the URL by trial and error: I tried fetching CSV files from Yahoo! Finance with different ticker symbols and different date ranges until I knew how to construct the queries.) We will use the `paste` function to put together all these different character values. Next, we will fetch the URL with the `read.csv` function, assigning the data frame to the symbol `tmp`. The data frame has most of the information we want but doesn't include the ticker symbol. So we will use the `cbind` function to attach a vector of ticker symbols to the data frame. (By the way, the function uses `Date` objects to represent the date. I also used the current date as the default value for `to`, and the date one year ago as the default value for `from`.)

Here is the function:

```
get.quotes <- function(ticker,
                       from=(Sys.Date()-365),
                       to=(Sys.Date()),
                       interval="d") {

 # define parts of the URL
 base <- "http://ichart.finance.yahoo.com/table.csv?";
 symbol <- paste("s=", ticker, sep="");

 # months are numbered from 00 to 11, so format the month correctly
 from.month <- paste("&a=",
  formatC(as.integer(format(from,"%m"))-1,width=2,flag="0"),
  sep="");
 from.day <- paste("&b=", format(from,"%d"), sep="");
 from.year <- paste("&c=", format(from,"%Y"), sep="");
 to.month <- paste("&d=",
  formatC(as.integer(format(to,"%m"))-1,width=2,flag="0"),
  sep="");
 to.day <- paste("&e=", format(to,"%d"), sep="");
 to.year <- paste("&f=", format(to,"%Y"), sep="");
 inter <- paste("&g=", interval, sep="");
 last <- "&ignore=.csv";

 # put together the url
 url <- paste(base, symbol, from.month, from.day, from.year,
              to.month, to.day, to.year, inter, last, sep="");

 # get the file
 tmp <- read.csv(url);

 # add a new column with ticker symbol labels
 cbind(symbol=ticker,tmp);
}
```

Study Material. Do not distribute.

Now let's write a function that returns a data frame with quotes from multiple securities. This function will simply call `get.quotes` once for every ticker in a vector of tickers and bind together the results using `rbind`:

```
get.multiple.quotes <- function(tkrs,
                                from=(Sys.Date()-365),
                                to=(Sys.Date()),
                                interval="d") {
    tmp <- NULL;
    for (tkr in tkrs) {
        if (is.null(tmp))
            tmp <- get.quotes(tkr,from,to,interval)
        else tmp <- rbind(tmp,get.quotes(tkr,from,to,interval))
        }
    tmp
}
```

Finally, let's define a vector with the set of ticker symbols in the Dow Jones Industrial Average and then build a data frame with data from all 30 tickers:

```
>  dow.tickers <- c("MMM", "AA", "AXP", "T", "BAC", "BA", "CAT", "CVX",
+                  "CSCO", "KO", "DD", "XOM", "GE", "HPQ", "HD", "INTC",
+                  "IBM", "JNJ", "JPM", "KFT", "MCD", "MRK", "MSFT", "PFE",
+                  "PG", "TRV", "UTX", "VZ", "WMT", "DIS")
> # date on which I ran this code
> Sys.Date()
[1] "2012-01-08"
> dow30 <- get.multiple.quotes(dow30.tickers)
```

We'll return to this data set below.data

Merging Data by Common Fields

As an example, let's return to the Baseball Databank database that we used in "Importing Data From Databases" on page 156. In this database, player information is stored in the `Master` table. Players are uniquely identified by the column `playerID`:

```
> dbListFields(con,"Master")
 [1] "lahmanID"     "playerID"     "managerID"    "hofID"
 [5] "birthYear"    "birthMonth"   "birthDay"     "birthCountry"
 [9] "birthState"   "birthCity"    "deathYear"    "deathMonth"
[13] "deathDay"     "deathCountry" "deathState"   "deathCity"
[17] "nameFirst"    "nameLast"     "nameNote"     "nameGiven"
[21] "nameNick"     "weight"       "height"       "bats"
[25] "throws"       "debut"        "finalGame"    "college"
[29] "lahman40ID"   "lahman45ID"   "retroID"      "holtzID"
[33] "bbrefID"
```

Batting information is stored in the `Batting` table. Players are uniquely identified by `playerID` in this table as well:

```
> dbListFields(con, "Batting")
 [1] "playerID"  "yearID"    "stint"     "teamID"    "lgID"
 [6] "G"         "G_batting" "AB"        "R"         "H"
[11] "2B"        "3B"        "HR"        "RBI"       "SB"
[16] "CS"        "BB"        "SO"        "IBB"       "HBP"
[21] "SH"        "SF"        "GIDP"      "G_old"
```

Study Material. Do not distribute.

Suppose that you wanted to show batting statistics for each player along with his name and age. To do this, you would need to merge data from the two tables. In R, you can do this with the merge function:

```
> batting <- dbGetQuery(con, "SELECT * FROM Batting")
> master <- dbGetQuery(con, "SELECT * FROM Master")
> batting.w.names <- merge(batting, master)
```

In this case, there was only one common variable between the two tables: playerID:

```
> intersect(names(batting), names(master))
[1] "playerID"
```

By default, merge uses common variables between the two data frames as the merge keys. So, in this case, we did not have to specify any more arguments to merge. Let's take a closer look at the arguments to merge (for data frames):

```
merge(x, y, by = , by.x = , by.y = , all = , all.x = , all.y = ,
      sort = , suffixes = , incomparables = , ...)
```

Here is a description of the arguments to merge.

Argument	Description	Default
x	One of the two data frames to combine.	
y	One of the two data frames to combine.	
by	A vector of character values corresponding to column names.	intersect(names(x), names(y))
by.x	A vector of character values corresponding to column names in x. Overrides the list given in by.	by
by.y	A vector of character values corresponding to column names in y. Overrides the list given in by.	by
all	A logical value specifying whether rows from each data frame should be included even if there is no match in the other data frame. This is equivalent to an OUTER JOIN in a database. (Equivalent to all.x=TRUE and all.y=TRUE.)	FALSE
all.x	A logical value specifying whether rows from data frame x should be included even if there is no match in the other data frame. This is equivalent to x LEFT OUTER JOIN y in a database.	all
all.y	A logical value specifying whether rows from data frame x should be included even if there is no match in the other data frame. This is equivalent to x RIGHT OUTER JOIN y in a database.	all
sort	A logical value that specifies whether the results should be sorted by the by columns.	TRUE
suffixes	A character vector with two values. If there are columns in x and y with the same name that are not used in the by list, they will be renamed with the suffixes given by this argument.	suffixes = c(“.x”, “.y”)

Study Material. Do not distribute.

Argument	Description	Default
incomparables	A list of variables that cannot be matched.	NULL

By default, `merge` is equivalent to a NATURAL JOIN in SQL. You can specify other columns to make it use `merge` like an INNER JOIN. You can specify values of ALL to get the same results as OUTER or FULL joins. If there are no matching field names, or if `by` is of length 0 (or `by.x` and `by.y` are of length 0), then `merge` will return the full Cartesian product of `x` and `y`.

Transformations

Sometimes, there will be some variables in your source data that aren't quite right. This section explains how to change a variable in a data frame.

Reassigning Variables

One of the most convenient ways to redefine a variable in a data frame is to use the assignment operator. For example, suppose that you wanted to change the type of a variable in the `dow30` data frame that we created above. When `read.csv` imported this data, it interpreted the "Date" field as a character string and converted it to a factor:

```
> class(dow30$Date)
[1] "factor"
```

Factors are fine for some things, but we could better represent the date field as a `Date` object. (That would create a proper ordering on dates and allow us to extract information from them.) Luckily, Yahoo! Finance prints dates in the default date format for R, so we can just transform these values into `Date` objects using `as.Date` (see the help file for `as.Date` for more information). So let's change this variable within the data frame to use `Date` objects:

```
> dow30$Date <- as.Date(dow30$Date)
> class(dow30$Date)
[1] "Date"
```

It's also possible to make other changes to data frames. For example, suppose that we wanted to define a new midpoint variable that is the mean of the high and low price. We can add this variable with the same notation:

```
> dow30$mid <- (dow30$High + dow30$Low) / 2
> names(dow30)
[1] "symbol"    "Date"      "Open"      "High"      "Low"
[6] "Close"     "Volume"    "Adj.Close" "mid"
```

The Transform Function

A convenient function for changing variables in a data frame is the `transform` function. Formally, `transform` is defined as:

```
transform(`_data`, ...)
```

Study Material. Do not distribute.

Notice that there aren't any named arguments for this function. To use `transform`, you specify a data frame (as the first argument) and a set of expressions that use variables within the data frame. The `transform` function applies each expression to the data frame and then returns the final data frame.

For example, suppose that we wanted to perform the two transformations listed above: changing the Date column to a Date format, and adding a new midpoint variable. We could do this with `transform` using the following expression:

```
> dow30.transformed <- transform(dow30, Date=as.Date(Date),
+   mid = (High + Low) / 2)
> names(dow30.transformed)
[1] "symbol"    "Date"      "Open"      "High"      "Low"
[6] "Close"     "Volume"    "Adj.Close" "mid"
> class(dow30.transformed$Date)
[1] "Date"
```

Applying a Function to Each Element of an Object

When transforming data, one common operation is to apply a function to a set of objects (or each part of a composite object) and return a new set of objects (or a new composite object). The base R library includes a set of different functions for doing this.

Applying a function to an array

To apply a function to parts of an array (or matrix), use the `apply` function:

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

`Apply` accepts three arguments: `X` is the array to which a function is applied, `FUN` is the function, and `MARGIN` specifies the dimensions to which you would like to apply a function. Optionally, you can specify arguments to `FUN` as addition arguments to apply arguments to `FUN`.) To show how this works, here's a simple example. Let's create a matrix with five rows of four elements, corresponding to the numbers between 1 and 20:

```
> x <- 1:20
> dim(x) <- c(5, 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

Now let's show how `apply` works. We'll use the function `max` because it's easy to look at the matrix above and see where the results came from.

First, let's select the maximum element of each row. (These are the values in the rightmost column: 16, 17, 18, 19, and 20.) To do this, we will specify `X=x`, `MARGIN=1` (rows are the first dimension), and `FUN=max`:

Study Material. Do not distribute.

```
> apply(X=x, MARGIN=1, FUN=max)
[1] 16 17 18 19 20
```

To do the same thing for columns, we simply have to change the value of `MARGIN`:

```
> apply(X=x, MARGIN=2, FUN=max)
[1]  5 10 15 20
```

As a slightly more complex example, we can also use `MARGIN` to apply a function over multiple dimensions. (We'll switch to the function `paste` to show which elements were included.) Consider the following three-dimensional array:

```
> x <- 1:27
> dim(x) <- c(3, 3, 3)
> x
, , 1

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18

, , 3

     [,1] [,2] [,3]
[1,]   19   22   25
[2,]   20   23   26
[3,]   21   24   27
```

Let's start by looking at which values are grouped for each value of `MARGIN`:

```
> apply(X=x, MARGIN=1, FUN=paste, collapse=",")
[1] "1,4,7,10,13,16,19,22,25" "2,5,8,11,14,17,20,23,26"
[3] "3,6,9,12,15,18,21,24,27"
> apply(X=x, MARGIN=2, FUN=paste, collapse=",")
[1] "1,2,3,10,11,12,19,20,21" "4,5,6,13,14,15,22,23,24"
[3] "7,8,9,16,17,18,25,26,27"
> apply(X=x, MARGIN=3, FUN=paste, collapse=",")
[1] "1,2,3,4,5,6,7,8,9"          "10,11,12,13,14,15,16,17,18"
[3] "19,20,21,22,23,24,25,26,27"
```

Let's do something more complicated. Let's select `MARGIN=c(1, 2)` to see which elements are selected:

```
> apply(X=x, MARGIN=c(1,2), FUN=paste, collapse=",")
     [,1]      [,2]      [,3]
[1,] "1,10,19" "4,13,22" "7,16,25"
[2,] "2,11,20" "5,14,23" "8,17,26"
[3,] "3,12,21" "6,15,24" "9,18,27"
```

Study Material. Do not distribute.

This is the equivalent of doing the following: for each value of *i* between 1 and 3 and each value of *j* between 1 and 3, calculate `FUN` of `x[i][j][1]`, `x[i][j][2]`, `x[i][j][3]`.

Applying a function to a list or vector

To apply a function to each element in a vector or a list and return a list, you can use the function `lapply`. The function `lapply` requires two arguments: an object `X` and a function `FUNC`. (You may specify additional arguments that will be passed to `FUNC`.) Let's look at a simple example of how to use `lapply`:

```
> x <- as.list(1:5)
> lapply(x,function(x) 2^x)
[[1]]
[1] 2

[[2]]
[1] 4

[[3]]
[1] 8

[[4]]
[1] 16

[[5]]
[1] 32
```

You can apply a function to a data frame, and the function will be applied to each vector in the data frame. For example:

```
> d <- data.frame(x=1:5, y=6:10)
> d
  x  y
1 1  6
2 2  7
3 3  8
4 4  9
5 5 10
> lapply(d,function(x) 2^x)
$x
[1]  2  4  8 16 32

$y
[1]   64  128  256  512 1024
> lapply(d,FUN=max)
$x
[1] 5

$y
[1] 10
```

Sometimes, you might prefer to get a vector, matrix, or array instead of a list. To do this, use the `sapply` function. This function works exactly the same way as `apply`, except that it returns a vector or matrix (when appropriate):

Study Material. Do not distribute.

```
> sapply(d, FUN=function(x) 2^x)
      x    y
[1,]  2   64
[2,]  4  128
[3,]  8  256
[4,] 16  512
[5,] 32 1024
```

Another related function is `mapply`, the "multivariate" version of `sapply`:

```
mapply(FUN, ..., MoreArgs = , SIMPLIFY = , USE.NAMES = )
```

Here is a description of the arguments to `mapply`.

Argument	Description	Default
FUN	The function to apply.	
...	A set of vectors over which FUN should be applied.	
MoreArgs	A list of additional arguments to pass to FUN.	
SIMPLIFY	A logical value indicating whether to simplify the returned array.	TRUE
USE.NAMES	A logical value indicating whether to use names for returned values. Names are taken from the values in the first vector (if it is a character vector) or from the names of elements in that vector.	TRUE

This function will apply `FUN` to the first element of each vector, then to the second, and so on, until it reaches the last element.

Here is a simple example of `mapply`:

```
> mapply(paste,
+        c(1, 2, 3, 4, 5),
+        c("a", "b", "c", "d", "e"),
+        c("A", "B", "C", "D", "E"),
+        MoreArgs=list(sep="-"))
[1] "1-a-A" "2-b-B" "3-c-C" "4-d-D" "5-e-E"
```

the plyr library

At this point, you're probably confused by all the different apply functions. They all accept different arguments, they're named inconsistently, and they work differently. Luckily, you don't have to remember any of the details of these function if you use the `plyr` package.

The `plyr` package contains a set of 12 logically named functions for applying another function to an R data object and returning the results. Each of these functions takes an array, data frame, or list as input and returns an array, data frame, list, or nothing as output. (You can choose to discard the results.) Here's a table of the most useful functions:

Study Material. Do not distribute.

Input	Array Output	Data Frame Output	List Output	Discard Output
Array	`aaply`	`adply`	`alply`	`a_ply`
Data Frame	`daply`	`ddply`	`dlply`	`d_ply`
List	`laply`	`ldply`	`llply`	`l_ply`

All of these functions accept the following arguments:

Argument	Description	Default
.data	The input data object	
.fun	The function to apply to the data	`NULL`
.progress	The type of progress bar (created with `create_progress`); choices include `"none"`, `"text"`, `"tk"`, and `"win"`	`"none"`
.expand	If .data is a dataframe, controls how output is expanded; choose `.expand=TRUE` for 1d output, `.expand=FALSE` for nd.	TRUE
.parallel	Specifies whether to apply the function in parallel (through foreach)	FALSE
...	Other arguments passed to `.fun`	

Other arguments depend on the input and output. If the input is an array, then these arguments are available:

Argument	Description	Default
.margins	A vector describing the subscripts to split up data by	

If the input is a data frame, then these arguments are available:

Argument	Description	Default
.drop (or .drop_i for daply)	Specifies whether to drop combinations of variables that do not appear in the data input	TRUE
.variables	Specifies a set of variables by which to split the data frame	
.drop_o (for daply only)	Specifies whether to drop extra dimensions in the output for dimensions of length 1	TRUE

If the output is dropped, then this argument is available:

Argument	Description	Default
.print	Specifies whether to print each output value	FALSE

Let's try to re-create some of our examples from above using `plyr`:

```
> # (1) input list, output list
> lapply(d, function(x) 2^x)
$x
[1]  2  4  8 16 32
```

Study Material. Do not distribute.

```
$y
[1]   64  128  256  512 1024
> # equivalent is llply
> llply(.data=d, .fun=function(x) 2^x)
$x
[1]  2  4  8 16 32

$y
[1]   64  128  256  512 1024
> # (2) input is an array, output is a vector
> apply(X=x,MARGIN=1, FUN=paste, collapse=",")
[1] "1,4,7,10,13,16,19,22,25" "2,5,8,11,14,17,20,23,26"
[3] "3,6,9,12,15,18,21,24,27"
> # equivalent (but note labels)
> aaply(.data=x,.margins=1, .fun=paste, collapse=",")
                        1                         2
"1,4,7,10,13,16,19,22,25" "2,5,8,11,14,17,20,23,26"
                        3
"3,6,9,12,15,18,21,24,27"
> # (3) Data frame in, matrix out
> t(sapply(d, FUN=function(x) 2^x))
  [,1] [,2] [,3] [,4] [,5]
x    2    4    8   16   32
y   64  128  256  512 1024
> # equivalent (but note the additional labels)
> aaply(.data=d, .fun=function(x) 2^x, .margins=2)

X1   1   2   3   4    5
  x  2   4   8  16   32
  y 64 128 256 512 1024
```

Binning Data

Another common data transformation is to group a set of observations into bins based on the value of a specific variable. For example, suppose you had some time series data where time was measured in days, but you wanted to summarize the data by month. There are several functions available for binning numeric data in R.

Shingles

We briefly mentioned shingles in "Shingles" on page 95. Shingles are a way to represent intervals in R. They can be overlapping, like roof shingles (hence the name). They are used extensively in the `lattice` package, when you want to use a numeric value as a conditioning value.

To create shingles in R, use the `shingle` function:

```
shingle(x, intervals=sort(unique(x)))
```

To specify where to separate the bins, use the `intervals` argument. You can use a numeric vector to indicate the breaks or a two-column matrix, where each row represents a specific interval.

Study Material. Do not distribute.

To create shingles where the number of observations is the same in each bin, you can use the `equal.count` function:

```
equal.count(x, ...)
```

Cut

The function `cut` is useful for taking a continuous variable and splitting it into discrete pieces. Here is the default form of `cut` for use with numeric vectors:

```
# numeric form
cut(x, breaks, labels = NULL,
    include.lowest = FALSE, right = TRUE, dig.lab = 3,
    ordered_result = FALSE, ...)
```

There is also a version of `cut` for manipulating `Date` objects:

```
# Date form
cut(x, breaks, labels = NULL, start.on.monday = TRUE,
    right = FALSE, ...)
```

The `cut` function takes a numeric vector as input and returns a factor. Each level in the factor corresponds to an interval of values in the input vector. Here is a description of the arguments to `cut`.

Argument	Description	Default
x	A numeric vector (to convert to a factor).	
breaks	Either a single integer value specifying the number of break points or a numeric vector specifying the set of break points.	
labels	Labels for the levels in the output factor.	`NULL`
include.lowest	A logical value indicating if a value equal to the lowest point in the range (if `right=TRUE`) in a range should be included in a given bucket. If `right=FALSE` indicates whether a value equal to the highest point in the range should be included.	`FALSE`
right	A logical value that specifies whether intervals should be closed on the right and open on the left. (For `right=FALSE`, intervals will be open on the right and closed on the left.)	`TRUE`
dig.lab	Number of digits used when generating labels (if labels are not explicitly specified).	`3`
ordered_results	A logical value indicating whether the result should be an ordered factor.	`FALSE`

For example, suppose that you wanted to count the number of players with batting averages in certain ranges. To do this, you could use the `cut` function and the `table` function:

```
> # load in the example data
> library(nutshell)
> data(batting.2008)
> # first, add batting average to the data frame:
> batting.2008.AB <- transform(batting.2008, AVG = H/AB)
> # now, select a subset of players with over 100 AB (for some
> # statistical significance):
> batting.2008.over100AB <- subset(batting.2008.AB, subset=(AB > 100))
> # finally, split the results into 10 bins:
> battingavg.2008.bins <- cut(batting.2008.over100AB$AVG,breaks=10)
```

Study Material. Do not distribute.

```
> table(battingavg.2008.bins)
battingavg.2008.bins
(0.137,0.163] (0.163,0.189] (0.189,0.215]  (0.215,0.24]  (0.24,0.266]
            4             6            24            67           121
(0.266,0.292] (0.292,0.318] (0.318,0.344]  (0.344,0.37]  (0.37,0.396]
          132            70            11             5             2
```

Combining Objects with a Grouping Variable

Sometimes you would like to combine a set of similar objects (either vectors or data frames) into a single data frame, with a column labeling the source. You can do this with the `make.groups` function in the `lattice` package:

```
library(lattice)
make.groups(...)
```

For example, let's combine three different vectors into a data frame:

```
> hat.sizes <- seq(from=6.25, to=7.75, by=.25)
> pants.sizes <- c(30, 31, 32, 33, 34, 36, 38, 40)
> shoe.sizes <- seq(from=7, to=12)
> make.groups(hat.sizes, pants.sizes, shoe.sizes)
              data       which
hat.sizes1    6.25   hat.sizes
hat.sizes2    6.50   hat.sizes
hat.sizes3    6.75   hat.sizes
hat.sizes4    7.00   hat.sizes
hat.sizes5    7.25   hat.sizes
hat.sizes6    7.50   hat.sizes
hat.sizes7    7.75   hat.sizes
pants.sizes1 30.00 pants.sizes
pants.sizes2 31.00 pants.sizes
pants.sizes3 32.00 pants.sizes
pants.sizes4 33.00 pants.sizes
pants.sizes5 34.00 pants.sizes
pants.sizes6 36.00 pants.sizes
pants.sizes7 38.00 pants.sizes
pants.sizes8 40.00 pants.sizes
shoe.sizes1   7.00  shoe.sizes
shoe.sizes2   8.00  shoe.sizes
shoe.sizes3   9.00  shoe.sizes
shoe.sizes4  10.00  shoe.sizes
shoe.sizes5  11.00  shoe.sizes
shoe.sizes6  12.00  shoe.sizes
```

Subsets

Often, you'll be provided with too much data. For example, suppose that you were working with patient records at a hospital. You might want to analyze healthcare records for patients between 5 and 13 years of age who were treated for asthma during the past 3 years. To do this, you need to take a subset of the data and not examine the whole database.

Other times, you might have too much relevant data. For example, suppose that you were looking at a logistics operation that fills billions of orders every year. R can

Study Material. Do not distribute.

hold only a certain number of records in memory and might not be able to hold the entire database. In most cases, you can get statistically significant results with a tiny fraction of the data; even millions of orders might be too many.

Bracket Notation

One way to take a subset of a data set is to use the bracket notation. As you may recall, you can select rows in a data frame by providing a vector of logical values. If you can write a simple expression describing the set of rows to select from a data frame, you can provide this as an index.

For example, suppose that we wanted to select only batting data from 2008. The column `batting.w.names$yearID` contains the year associated with each row, so we could calculate a vector of logical values describing which rows to keep with the expression `batting.w.names$yearID==2008`. Now we just have to index the data frame `batting.w.names` with this vector to select only rows for the year 2008:

```
> batting.w.names.2008 <- batting.w.names[batting.w.names$yearID==2008,]
> summary(batting.w.names.2008$yearID)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   2008    2008    2008    2008    2008    2008 
```

Similarly, we can use the same notation to select only certain columns. Suppose that we wanted to keep only the variables `nameFirst`, `nameLast`, `AB`, `H`, and `BB`. We could provide these in the brackets as well:

```
> batting.w.names.2008.short <-
+    batting.w.names[batting.w.names$yearID==2008,
+    c("nameFirst", "nameLast", "AB", "H", "BB")]
```

subset Function

As an alternative, you can use the `subset` function to select a subset of rows and columns from a data frame (or matrix):

```
subset(x, subset, select, drop = FALSE, ...)
```

There isn't anything you can do with `subset` that you can't do with the bracket notation, but using `subset` can lead to more readable code. `Subset` allows you to use variable names from the data frame when selecting subsets, saving some typing. Here is a description of the arguments to `subset`.

Argument	Description	Default
x	The object from which to calculate a subset.	
subset	A logical expression that describes the set of rows to return.	
select	An expression indicating which columns to return.	
drop	Passed to `[`.	`FALSE`

As an example, let's recreate the same data sets we created above using `subset`:

Study Material. Do not distribute.

```
> batting.w.names.2008 <- subset(batting.w.names, yearID==2008)
> batting.w.names.2008.short <- subset(batting.w.names, yearID==2008,
+     c("nameFirst","nameLast","AB","H","BB"))
```

Random Sampling

Often, it is desirable to take a random sample of a data set. Sometimes, you might have too much data (for statistical reasons or for performance reasons). Other times, you simply want to split your data into different parts for modeling (usually into training, testing, and validation subsets).

One of the simplest ways to extract a random sample is with the `sample` function. The `sample` function returns a random sample of the elements of a vector:

```
sample(x, size, replace = FALSE, prob = NULL)
```

Argument	Description	Default
x	The object from which the sample is taken	
size	An integer value specifying the sample size	
replace	A logical value indicating whether to sample with, or without, replacement	FALSE
prob	A vector of probabilities for selecting each item	NULL

Somewhat nonintuitively, when applied to a data frame, `sample` will return a random sample of the columns. (Remember that a data frame is implemented as a list of vectors, so `sample` is just taking a random sample of the elements of the list.) So you need to be a little more clever when you use `sample` with a data frame.

To take a random sample of the observations in a data set, you can use `sample` to create a random sample of row numbers and then select these row numbers using an index operator. For example, let's take a random sample of five elements from the `batting.2008` data set:

```
> batting.2008[sample(1:nrow(batting.2008), 5), ]
       playerID yearID stint teamID lgID   G G_batting  AB  R  H 2B 3B
90648 izturma01   2008     1    LAA   AL  79        79 290 44 78 14  2
90280 benoijo01   2008     1    TEX   AL  44         3   0  0  0  0  0
90055 percitr01   2008     1    TBA   AL  50         4   0  0  0  0  0
91085  getzch01   2008     1    CHA   AL  10        10   7  2  2  0  0
90503 willijo03   2008     1    FLO   NL 102       102 351 54 89 21  5
      HR RBI SB CS BB SO IBB HBP SH SF GIDP G_old
90648  3  37 11  2 26 27   0   1  2  2    9    79
90280  0   0  0  0  0  0   0   0  0  0    0     3
90055  0   0  0  0  0  0   0   0  0  0    0     4
91085  0   1  1  1  0  1   0   0  0  0    0    10
90503 15  51  3  2 48 82   2  14  1  2    7   102
```

You can also use this technique to select a more complicated random subset. For example, suppose that you wanted to randomly select statistics for three teams. You could do this as follows:

```
> batting.2008$teamID <- as.factor(batting.2008$teamID)
> levels(batting.2008$teamID)
```

Study Material. Do not distribute.

```
 [1] "ARI" "ATL" "BAL" "BOS" "CHA" "CHN" "CIN" "CLE" "COL" "DET" "FLO"
[12] "HOU" "KCA" "LAA" "LAN" "MIL" "MIN" "NYA" "NYN" "OAK" "PHI" "PIT"
[23] "SDN" "SEA" "SFN" "SLN" "TBA" "TEX" "TOR" "WAS"
> # example of sample
> sample(levels(batting.2008$teamID), 3)
[1] "ATL" "TEX" "DET"
> # usage example (note that it's a different random sample of teams)
> batting.2008.3teams <- batting.2008[is.element(batting.2008$teamID,
+      sample(levels(batting.2008$teamID), 3)), ]
> # check to see that sample only has three teams
> summary(batting.2008.3teams$teamID)
ARI ATL BAL BOS CHA CHN CIN CLE COL DET FLO HOU KCA LAA LAN MIL MIN
  0   0   0   0   0   0  48   0   0   0   0   0   0  41   0  44   0
NYA NYN OAK PHI PIT SDN SEA SFN SLN TBA TEX TOR WAS
  0   0   0   0   0   0   0   0   0   0   0   0   0
```

This function is good for data sources where you simply want to take a random sample of all the observations, but often you might want to do something more complicated, like stratified sampling, cluster sampling, maximum entropy sampling, or other more sophisticated methods. You can find many of these methods in the `sampling` package. For an example using this package to do stratified sampling, see "Machine Learning Algorithms for Classification" on page 477.

Summarizing Functions

Often, you are provided with data that is too fine grained for your analysis. For example, you might be analyzing data about a website. Suppose that you wanted to know the average number of pages delivered to each user. To find the answer, you might need to look at every HTTP transaction (every request for content), grouping together requests into sessions and counting the number of requests. R provides a number of different functions for summarizing data, aggregating records together to build a smaller data set.

tapply, aggregate

The `tapply` function is a very flexible function for summarizing a vector `X`. You can specify which subsets of `X` to summarize, as well as the function used for summarization:

```
tapply(X, INDEX, FUN = , ..., simplify = )
```

Here are the arguments to `tapply`.

Argument	Description	Default
X	The object on which to apply the function (usually a vector).	
INDEX	A list of factors that specify different sets of values of X over which to calculate FUN, each the same length as X.	
FUN	The function applied to elements of X.	NULL
...	Optional arguments are passed to FUN.	

Study Material. Do not distribute.

Argument	Description	Default
simplify	If `simplify=TRUE`, then if FUN returns a scalar, then `tapply` returns an array with the mode of the scalar. If `simplify=FALSE`, then `tapply` returns a list.	TRUE

For example, we can use `tapply` to sum the number of home runs by team:

```
> tapply(X=batting.2008$HR, INDEX=list(batting.2008$teamID), FUN=sum)
ARI ATL BAL BOS CHA CHN CIN CLE COL DET FLO HOU KCA LAA LAN MIL MIN
159 130 172 173 235 184 187 171 160 200 208 167 120 159 137 198 111
NYA NYN OAK PHI PIT SDN SEA SFN SLN TBA TEX TOR WAS
180 172 125 214 153 154 124  94 174 180 194 126 117
```

You can also apply a function that returns multiple items, such as `fivenum` (which returns a vector containing the minimum, lower-hinge, median, upper-hinge, and maximum values) to the data. For example, here is the result of applying `fivenum` to the batting averages of each player, aggregated by league:

```
> tapply(X=(batting.2008$H/batting.2008$AB),
+   INDEX=list(batting.2008$lgID),FUN=fivenum)
$AL
[1] 0.0000000 0.1758242 0.2487923 0.2825485 1.0000000

$NL
[1] 0.0000000 0.0952381 0.2172524 0.2679739 1.0000000
```

You can also use `tapply` to calculate summaries over multiple dimensions. For example, we can calculate the mean number of home runs per player by league and batting hand:

```
> tapply(X=(batting.2008$HR),
+   INDEX=list(batting.w.names.2008$lgID,
+     batting.w.names.2008$bats),
+   FUN=mean)
          B        L        R
AL 3.058824 3.478495 3.910891
NL 3.313433 3.400000 3.344902
```

(As a side note, there is no equivalent to `tapply` in the `plyr` package.)

A function closely related to `tapply` is `by`. The `by` function works the same way as `tapply`, except that it works on data frames. The `INDEX` argument is replaced by an `INDICES` argument. Here is an example:

```
> by(batting.2008[, c("H", "2B", "3B", "HR")],
+   INDICES=list(batting.w.names.2008$lgID,
+     batting.w.names.2008$bats), FUN=mean)
: AL
: B
         H         2B         3B         HR
29.0980392  5.4901961  0.8431373  3.0588235
-----------------------------------------------------
: NL
: B
         H         2B         3B         HR
29.2238806  6.4776119  0.6865672  3.3134328
-----------------------------------------------------
```

Study Material. Do not distribute.

```
: AL
: L
        H         2B         3B         HR
32.4301075  6.7258065  0.5967742  3.4784946
-----------------------------------------------------
: NL
: L
       H        2B        3B        HR
31.888372  6.283721  0.627907  3.400000
-----------------------------------------------------
: AL
: R
        H         2B         3B         HR
34.2549505  7.0495050  0.6460396  3.9108911
-----------------------------------------------------
: NL
: R
        H         2B         3B         HR
29.9414317  6.1822126  0.6290672  3.3449024
```

Another option for summarization is the function aggregate. Here is the form of aggregate when applied to data frames:

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

Aggregate can also be applied to time series and takes slightly different arguments:

```
aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,
          ts.eps = getOption("ts.eps"), ...)
```

Here is a description of the arguments to aggregate.

Argument	Description	Default
x	The object to aggregate	
by	A list of grouping elements, each as long as x	
FUN	A scalar function used to compute the summary statistic	no default for data frames; for time series, FUN=SUM
nfrequency	Number of observations per unit of time	1
ndeltat	Fraction of the sampling period between successive observations	1
ts.eps	Tolerance used to decide if nfrequency is a submultiple of the original frequency	getOption("ts.eps")
...	Further arguments passed to FUN	

For example, we can use aggregate to summarize batting statistics by team:

```
> aggregate(x=batting.2008[, c("AB", "H", "BB", "2B", "3B", "HR")],
+    by=list(batting.2008$teamID), FUN=sum)
   Group.1   AB    H  BB  2B 3B  HR
1      ARI 5409 1355 587 318 47 159
2      ATL 5604 1514 618 316 33 130
3      BAL 5559 1486 533 322 30 172
4      BOS 5596 1565 646 353 33 173
5      CHA 5553 1458 540 296 13 235
```

Study Material. Do not distribute.

```
6       CHN 5588 1552 636 329 21 184
7       CIN 5465 1351 560 269 24 187
8       CLE 5543 1455 560 339 22 171
9       COL 5557 1462 570 310 28 160
10      DET 5641 1529 572 293 41 200
11      FLO 5499 1397 543 302 28 208
12      HOU 5451 1432 449 284 22 167
13      KCA 5608 1507 392 303 28 120
14      LAA 5540 1486 481 274 25 159
15      LAN 5506 1455 543 271 29 137
16      MIL 5535 1398 550 324 35 198
17      MIN 5641 1572 529 298 49 111
18      NYA 5572 1512 535 289 20 180
19      NYN 5606 1491 619 274 38 172
20      OAK 5451 1318 574 270 23 125
21      PHI 5509 1407 586 291 36 214
22      PIT 5628 1454 474 314 21 153
23      SDN 5568 1390 518 264 27 154
24      SEA 5643 1498 417 285 20 124
25      SFN 5543 1452 452 311 37  94
26      SLN 5636 1585 577 283 26 174
27      TBA 5541 1443 626 284 37 180
28      TEX 5728 1619 595 376 35 194
29      TOR 5503 1453 521 303 32 126
30      WAS 5491 1376 534 269 26 117
```

Aggregating Tables with rowsum

Sometimes, you would simply like to calculate the sum of certain variables in an object, grouped together by a grouping variable. To do this in R, use the `rowsum` function:

```
rowsum(x, group, reorder = TRUE, ...)
```

For example, we can use `rowsum` to summarize batting statistics by team:

```
> rowsum(batting.2008[,c("AB", "H", "BB", "2B", "3B", "HR")],
+   group=batting.2008$teamID)
      AB    H  BB X2B X3B  HR
ARI 5409 1355 587 318  47 159
ATL 5604 1514 618 316  33 130
BAL 5559 1486 533 322  30 172
BOS 5596 1565 646 353  33 173
CHA 5553 1458 540 296  13 235
CHN 5588 1552 636 329  21 184
CIN 5465 1351 560 269  24 187
CLE 5543 1455 560 339  22 171
COL 5557 1462 570 310  28 160
DET 5641 1529 572 293  41 200
FLO 5499 1397 543 302  28 208
HOU 5451 1432 449 284  22 167
KCA 5608 1507 392 303  28 120
LAA 5540 1486 481 274  25 159
LAN 5506 1455 543 271  29 137
MIL 5535 1398 550 324  35 198
MIN 5641 1572 529 298  49 111
```

Study Material. Do not distribute.

```
NYA 5572 1512 535 289  20 180
NYN 5606 1491 619 274  38 172
OAK 5451 1318 574 270  23 125
PHI 5509 1407 586 291  36 214
PIT 5628 1454 474 314  21 153
SDN 5568 1390 518 264  27 154
SEA 5643 1498 417 285  20 124
SFN 5543 1452 452 311  37  94
SLN 5636 1585 577 283  26 174
TBA 5541 1443 626 284  37 180
TEX 5728 1619 595 376  35 194
TOR 5503 1453 521 303  32 126
WAS 5491 1376 534 269  26 117
```

Counting Values

Often, it can be useful to count the number of observations that take on each possible value of a variable. R provides several functions for doing this.

The simplest function for counting the number of observations that take on a value is the `tabulate` function. This function counts the number of elements in a vector that take on each integer value and returns a vector with the counts.

As an example, suppose that you wanted to count the number of players who hit 0 HR, 1 HR, 2 HR, 3 HR, and so on. You could do this with the `tabulate` function:

```
> HR.cnts <- tabulate(batting.w.names.2008$HR)
> # tabulate doesn't label results, so let's add names:
> names(HR.cnts) <- 0:(length(HR.cnts) - 1)
> HR.cnts
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
92 63 45 20 15 26 23 21 22 15 15 18 12 10 12  4  9  3  3 13  9  7 10
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
 4  8  2  5  2  4  0  1  6  6  3  1  2  4  1  0  0  0  0  0  0  0  0
46 47
 0  1
```

A related function (for categorical values) is `table`. Suppose that you are presented with some data that includes a few categorical values (encoded as factors in R) and wanted to count how many observations in the data had each categorical value. To do this, you can use the `table` function:

```
table(..., exclude = if (useNA == "no") c(NA, NaN), useNA = c("no",
    "ifany", "always"), dnn = list.names(...), deparse.level = 1)
```

The `table` function returns a table object showing the number of observations that have each possible categorical value.[2] Here are the arguments to `table`.

2. If you are familiar with SAS, you can think of `table` as the equivalent to `PROC FREQ`.

Study Material. Do not distribute.

Argument	Description	Default
...	A set of factors (or objects that can be coerced into factors).	
exclude	Levels to remove from factors.	`if (useNA == "no") c(NA, NaN)`
useNA	Indicates whether to include NA values in the table.	`c("no", "ifany", "always")`
dnn	Names to be given to dimensions in the result.	`list.names(...)`
deparse.level	As noted in the help file: "If the argument dnn is not supplied, the internal function list.names is called to compute the 'dim-name names'. If the arguments in ... are named, those names are used. For the remaining arguments, deparse.level = 0 gives an empty name, deparse.level = 1 uses the supplied argument if it is a symbol, and deparse.level = 2 will deparse the argument."	`1`

For example, suppose that we wanted to count the number of left-handed batters, right-handed batters, and switch hitters in 2008. We could use the data frame `batting.w.names.2008` defined above to provide the data and `table` to tabulate the results:

```
> table(batting.w.names.2008$bats)

  B   L   R
118 401 865
```

To make this a little more interesting, we could make this a two-dimensional table showing the number of players who batted and threw with each hand:

```
> table(batting.2008[,c("bats", "throws")])
    throws
bats   L   R
   B  10 108
   L 240 161
   R  25 840
```

We could extend the results to another dimension, adding league ID:

```
, , lgID = AL

    throws
bats   L   R
   B   4  47
   L 109  77
   R  11 393

, , lgID = NL

    throws
bats   L   R
   B   6  61
   L 131  84
   R  14 447
```

Study Material. Do not distribute.

Another useful function is xtabs, which creates contingency tables from factors using formulas:

```
xtabs(formula = ~., data = parent.frame(), subset, na.action,
      exclude = c(NA, NaN), drop.unused.levels = FALSE)
```

The xtabs function works the same as table, but it allows you to specify the groupings by specifying a formula and a data frame. In many cases, this can save you some typing. For example, here is how to use xtabs to tabulate batting statistics by batting arm and league:

```
> xtabs(~bats+lgID, batting.2008)
    lgID
bats  AL  NL
   B  51  67
   L 186 215
   R 404 461
```

The table function only works on factors, but sometimes you might like to calculate tables with numeric values as well. For example, suppose you wanted to count the number of players with batting averages in certain ranges. To do this, you could use the cut function and the table function:

```
> # first, add batting average to the data frame:
> batting.w.names.2008 <- transform(batting.w.names.2008, AVG = H/AB)
> # now, select a subset of players with over 100 AB (for some
> # statistical significance):
> batting.2008.over100AB <- subset(batting.2008, subset=(AB > 100))
> # finally, split the results into 10 bins:
> battingavg.2008.bins <- cut(batting.2008.over100AB$AVG,breaks=10)
> table(battingavg.2008.bins)
battingavg.2008.bins
(0.137,0.163] (0.163,0.189] (0.189,0.215]  (0.215,0.24]  (0.24,0.266]
            4             6            24            67           121
(0.266,0.292] (0.292,0.318] (0.318,0.344]  (0.344,0.37]  (0.37,0.396]
          132            70            11             5             2
```

Reshaping Data

Very often, you are presented with data that is in the wrong "shape." Sometimes, you might find that a single observation is stored across multiple lines in a data frame. This happens very often in data warehouses. In these systems, a single table might be used to represent many different "facts." Each fact might be associated with a unique identifier, a timestamp, a concept, and an observed value. To build a statistical model or to plot results, you might need to create a version of the data in which each line contains a unique identifier, a timestamp, and a column for each concept. So you might want to transform this "narrow" data set to a "wide" format.

Other times, you might be presented with a sparsely populated data frame that has a large number of columns. Although this format might make analysis straightforward, the data set might also be large and difficult to store. So you might want to transform this wide data set into a narrow one.

Study Material. Do not distribute.

Transposing matrices and data frames

A very useful function is `t`, which transposes objects. The `t` function takes one argument: an object to transpose. The object can be a matrix, vector, or data frame. Here is an example with a matrix:

```
> m <- matrix(1:10, nrow=5)
> m
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> t(m)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

When you call `t` on a vector, the vector is treated as a single column of a matrix. So the value returned by `t` will be a matrix with a single row:

```
> v <- 1:10
> v
 [1]  1  2  3  4  5  6  7  8  9 10
> t(v)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9    10
```

Reshaping data frames and matrices

R includes several functions that let you change data between narrow and wide formats. Let's use a small table of stock data to show how these functions work. First, we'll define a small portfolio of stocks. Then we'll get monthly observation for the first three months of 2009:

```
> my.tickers <- c("GE", "GOOG", "AAPL", "AXP", "GS")
> my.quotes <- get.multiple.quotes(my.tickers, from=as.Date("2009-01-01"),
+     to=as.Date("2009-03-31"), interval="m")
> my.quotes
   symbol       Date   Open   High    Low  Close    Volume Adj.Close
1      GE 2009-03-02   8.29  11.35   5.87  10.11 277426300     10.11
2      GE 2009-02-02  12.03  12.90   8.40   8.51 1949288ls00     8.51
3      GE 2009-01-02  16.51  17.24  11.87  12.13 117846700     11.78
4    GOOG 2009-03-02 333.33 359.16 289.45 348.06   5346800    348.06
5    GOOG 2009-02-02 334.29 381.00 329.55 337.99   6158100    337.99
6    GOOG 2009-01-02 308.60 352.33 282.75 338.53   5727600    338.53
7    AAPL 2009-03-02  88.12 109.98  82.33 105.12  25963400    105.12
8    AAPL 2009-02-02  89.10 103.00  86.51  89.31  27394900     89.31
9    AAPL 2009-01-02  85.88  97.17  78.20  90.13  33487900     90.13
10    AXP 2009-03-02  11.68  15.24   9.71  13.63  31136400     13.45
11    AXP 2009-02-02  16.35  18.27  11.44  12.06  24297100     11.90
12    AXP 2009-01-02  18.57  21.38  14.72  16.73  19110000     16.51
13     GS 2009-03-02  87.86 115.65  72.78 106.02  30196400    106.02
14     GS 2009-02-02  78.78  98.66  78.57  91.08  28301500     91.08
15     GS 2009-01-02  84.02  92.20  59.13  80.73  22764300     80.29
```

Study Material. Do not distribute.

Now let's keep only the Date, Symbol, and Close columns:

```
> my.quotes.narrow <- my.quotes[,c("symbol", "Date", "Close")]
> my.quotes.narrow
   symbol       Date  Close
1      GE 2009-03-02  10.11
2      GE 2009-02-02   8.51
3      GE 2009-01-02  12.13
4    GOOG 2009-03-02 348.06
5    GOOG 2009-02-02 337.99
6    GOOG 2009-01-02 338.53
7    AAPL 2009-03-02 105.12
8    AAPL 2009-02-02  89.31
9    AAPL 2009-01-02  90.13
10    AXP 2009-03-02  13.63
11    AXP 2009-02-02  12.06
12    AXP 2009-01-02  16.73
13     GS 2009-03-02 106.02
14     GS 2009-02-02  91.08
15     GS 2009-01-02  80.73
```

We can use the `unstack` function to change the format of this data from a stacked form to an unstacked form:

```
> unstack(my.quotes.narrow, form=Close~symbol)
     GE   GOOG   AAPL   AXP     GS
1 10.11 348.06 105.12 13.63 106.02
2  8.51 337.99  89.31 12.06  91.08
3 12.13 338.53  90.13 16.73  80.73
```

The first argument to `unstack` specifies the data frame. The second argument, `form`, uses a formula to specify how to unstack the data frame. The right side of the formula represents the vector to be unstacked (in this case, `symbol`). The left side indicates the groups to create (in this case `Close`).

Notice that the `unstack` operation retains the order of observations but loses the Date column. (It's probably best to use `unstack` with data in which there are only two variables that matter.) You can also transform data the other way, stacking observations to create a long list:

```
> unstacked <- unstack(my.quotes.narrow, form=Close~symbol)
> stack(unstacked)
   values  ind
1   10.11   GE
2    8.51   GE
3   12.13   GE
4  348.06 GOOG
5  337.99 GOOG
6  338.53 GOOG
7  105.12 AAPL
8   89.31 AAPL
9   90.13 AAPL
10  13.63  AXP
11  12.06  AXP
12  16.73  AXP
13 106.02   GS
```

Study Material. Do not distribute.

```
14  91.08    GS
15  80.73    GS
```

R includes a more powerful function for changing the shape of a data frame: the `reshape` function. Before explaining how to use this function (it's a bit complicated), let's use a couple of examples to show what it does.

First, suppose that we wanted each row to represent a unique date and each column to represent a different stock. We can do this with the `reshape` function:

```
> my.quotes.wide <- reshape(my.quotes.narrow, idvar="Date",
+    timevar="symbol", direction="wide")
> my.quotes.wide
        Date Close.GE Close.GOOG Close.AAPL Close.AXP Close.GS
1 2009-03-02    10.11     348.06     105.12     13.63   106.02
2 2009-02-02     8.51     337.99      89.31     12.06    91.08
3 2009-01-02    12.13     338.53      90.13     16.73    80.73
```

Parameters for `reshape` are stored as attributes of the created data frame:

```
> attributes(my.quotes.wide)
$row.names
[1] 1 2 3

$names
[1] "Date"       "Close.GE"   "Close.GOOG" "Close.AAPL" "Close.AXP"
[6] "Close.GS"

$class
[1] "data.frame"

$reshapeWide
$reshapeWide$v.names
NULL

$reshapeWide$timevar
[1] "symbol"

$reshapeWide$idvar
[1] "Date"

$reshapeWide$times
[1] GE   GOOG AAPL AXP  GS
Levels: GE GOOG AAPL AXP GS

$reshapeWide$varying
     [,1]       [,2]         [,3]         [,4]        [,5]
[1,] "Close.GE" "Close.GOOG" "Close.AAPL" "Close.AXP" "Close.GS"
```

Alternatively, we could have each row represent a stock and each column represent a different date:

```
> reshape(my.quotes.narrow, idvar="symbol", timevar="Date", direction="wide")
  symbol Close.2009-03-02 Close.2009-02-02 Close.2009-01-02
1     GE            10.11             8.51            12.13
4   GOOG           348.06           337.99           338.53
7   AAPL           105.12            89.31            90.13
```

Study Material. Do not distribute.

```
10    AXP          13.63           12.06           16.73
13     GS         106.02           91.08           80.73
```

We could even go in the opposite direction:

```
> reshape(my.quotes.wide)
                       Date symbol Close.GE
2009-03-02.GE    2009-03-02     GE    10.11
2009-02-02.GE    2009-02-02     GE     8.51
2009-01-02.GE    2009-01-02     GE    12.13
2009-03-02.GOOG 2009-03-02   GOOG   348.06
2009-02-02.GOOG 2009-02-02   GOOG   337.99
2009-01-02.GOOG 2009-01-02   GOOG   338.53
2009-03-02.AAPL 2009-03-02   AAPL   105.12
2009-02-02.AAPL 2009-02-02   AAPL    89.31
2009-01-02.AAPL 2009-01-02   AAPL    90.13
2009-03-02.AXP  2009-03-02    AXP    13.63
2009-02-02.AXP  2009-02-02    AXP    12.06
2009-01-02.AXP  2009-01-02    AXP    16.73
2009-03-02.GS   2009-03-02     GS   106.02
2009-02-02.GS   2009-02-02     GS    91.08
2009-01-02.GS   2009-01-02     GS    80.73
```

By the way, you can also use `reshape` to create columns for multiple data values at once:

```
> my.quotes.oc <- my.quotes[,c("symbol", "Date", "Close", "Open")]
> my.quotes.oc
   symbol       Date  Close   Open
1      GE 2009-03-02  10.11   8.29
2      GE 2009-02-02   8.51  12.03
3      GE 2009-01-02  12.13  16.51
4    GOOG 2009-03-02 348.06 333.33
5    GOOG 2009-02-02 337.99 334.29
6    GOOG 2009-01-02 338.53 308.60
7    AAPL 2009-03-02 105.12  88.12
8    AAPL 2009-02-02  89.31  89.10
9    AAPL 2009-01-02  90.13  85.88
10    AXP 2009-03-02  13.63  11.68
11    AXP 2009-02-02  12.06  16.35
12    AXP 2009-01-02  16.73  18.57
13     GS 2009-03-02 106.02  87.86
14     GS 2009-02-02  91.08  78.78
15     GS 2009-01-02  80.73  84.02
> # now, let's change the shape of this data frame:
> reshape(my.quotes.oc, timevar="Date", idvar="symbol", direction="wide")
   symbol Close.2009-03-02 Open.2009-03-02 Close.2009-02-02
1      GE            10.11            8.29             8.51
4    GOOG           348.06          333.33           337.99
7    AAPL           105.12           88.12            89.31
10    AXP            13.63           11.68            12.06
13     GS           106.02           87.86            91.08
   Open.2009-02-02 Close.2009-01-02 Open.2009-01-02
1            12.03            12.13           16.51
4           334.29           338.53          308.60
7            89.10            90.13           85.88
```

Study Material. Do not distribute.

```
10          16.35          16.73          18.57
13          78.78          80.73          84.02
```

The tricky thing about `reshape` is that it is actually two functions in one: a function that transforms long data to wide data and a function that transforms wide data to long data. The direction argument specifies whether you want a data frame that is "long" or "wide."

When transforming to wide data, you need to specify the `idvar` and `timevar` arguments. When transforming to long data, you need to specify the `varying` argument.

By the way, calls to `reshape` are reversible. If you have an object `d` that was created by a call to `reshape`, you can call `reshape(d)` to get back the original data frame:

```
reshape(data, varying = , v.names = , timevar = , idvar = , ids = , times = ,
        drop = , direction, new.row.names = , sep = , split = )
```

Here are the arguments to `reshape`.

Argument	Description	Default
data	A data frame to reshape.	
varying	A list of variables in the wide format that should be assigned to unique rows in the long format. Usually given as a list of variable names, but can be a matrix of names or a vector of names. (You can also use integers in this argument, which are used to index names [data].)	`NULL`
v.names	Names of variables in the long format that should be assigned to columns in the wide format.	`NULL`
timevar	The variable in the long format that identifies unique observations for the same group or individual (when going from the long to the wide format).	`"time"`
idvar	The variable in the long format that identifies unique groups or individuals (when going from the long to the wide format).	`"id"`
ids	The values to use for a new `idvar` variable.	`1:NROW(data)`
times	The values to use for a new `timevar` variable.	`seq_along( varying[[1]])`
drop	A vector of variable names to exclude from reshaping.	`NULL`
direction	A character value that specifies the reshaping direction: "wide" reshapes long data to wide data, and "long" reshapes wide data to long data.	
new.row.names	A logical value. When reshaping long data to wide data, specifies whether to create new row names from the values of the `id` and `time` variables.	`NULL`
sep	A character value. The reshape function will attempt to guess values for `v.names` and `v.times` when moving from wide to long data. This variable specifies the separator that is used in the variable names.	`"."`

Study Material. Do not distribute.

Argument	Description	Default
split	As noted in the description for `sep`, `reshape` will attempt to split variable names into `v.names` and `v.times`. If the relationship between the variables is more complicated than just concatenation with a single value, reshape can still automatically guess values for `v.names` and `v.times`. See the help file for more information.	`if (sep=="") { list(regexp= "[A-Za-z][0-9]", include=TRUE) } else { list(regexp=sep, include=FALSE, fixed=TRUE) }`

Using the Reshape Library

Many R users (like me) find the built-in functions for reshaping data (like stack, unstack, and reshape) confusing. Luckily, there's an alternative. Hadley Wickham (the author of `ggplot2`) has developed a library called `reshape` with a much more intuitive model for getting data into the right form. (Don't confuse the `reshape` library with the `reshape` function.)

Melting and Casting. Reshape uses an intuitive model to describe how to manipulate data tables. Hadley observed that if you had detailed transactional data, then you could easily manipulate that data into many different forms. Quite often, you could take an existing table of data, turn it into a list of transactions, and then shape it into a different form. He called the process of turning a table of data into a set of transactions *melting*, and the process of turning the list of transactions into a table *casting*.

Examples of reshape. Let's see how melting and casting work, using the same data that we used above to show how much easier the `reshape` library is. First, let's `melt` the quote data.

```
> # call melt using the default settings
> my.molten.quotes <- melt(my.quotes)
Using symbol, Date as id variables
> # just show the first few lines
> head(my.molten.quotes)
  symbol       Date variable  value
1     GE 2009-03-02     Open   8.29
2     GE 2009-02-02     Open  12.03
3     GE 2009-01-02     Open  16.51
4   GOOG 2009-03-02     Open 333.33
5   GOOG 2009-02-02     Open 334.29
6   GOOG 2009-01-02     Open 308.60
```

Now that we have the data into a molten form, it's very straightforward to transform it with `cast`. Here are a few examples:

```
> # prices by date for just GE
> cast(data=my.molten.quotes, variable~Date, subset=(symbol=='GE'))
   variable   2009-01-02   2009-02-02   2009-03-02
1      Open        16.51        12.03         8.29
2      High        17.24        12.90        11.35
3       Low        11.87         8.40         5.87
4     Close        12.13         8.51        10.11
5    Volume 117846700.00 194928800.00 277426300.00
6 Adj.Close        10.75         7.77         9.23
```

Study Material. Do not distribute.

```
> # Closing prices for each stock by date
> cast(data=my.molten.quotes, symbol~Date, subset=(variable=='Adj.Close'))
  symbol 2009-01-02 2009-02-02 2009-03-02
1     GE      10.75       7.77       9.23
2   GOOG     338.53     337.99     348.06
3   AAPL      90.13      89.31     105.12
4    AXP      15.70      11.32      12.79
5     GS      77.85      88.31     102.79
> # Return a list of quotes by symbol and date
> cast(data=my.molten.quotes, Date~variable|symbol)
$GE
        Date  Open  High   Low Close    Volume Adj.Close
1 2009-01-02 16.51 17.24 11.87 12.13 117846700     10.75
2 2009-02-02 12.03 12.90  8.40  8.51 194928800      7.77
3 2009-03-02  8.29 11.35  5.87 10.11 277426300      9.23

$GOOG
        Date   Open   High    Low  Close  Volume Adj.Close
1 2009-01-02 308.60 352.33 282.75 338.53 5727600    338.53
2 2009-02-02 334.29 381.00 329.55 337.99 6158100    337.99
3 2009-03-02 333.33 359.16 289.45 348.06 5346800    348.06

$AAPL
        Date  Open   High   Low  Close   Volume Adj.Close
1 2009-01-02 85.88  97.17 78.20  90.13 33487900     90.13
2 2009-02-02 89.10 103.00 86.51  89.31 27394900     89.31
3 2009-03-02 88.12 109.98 82.33 105.12 25963400    105.12

$AXP
        Date  Open  High   Low Close   Volume Adj.Close
1 2009-01-02 18.57 21.38 14.72 16.73 19110000     15.70
2 2009-02-02 16.35 18.27 11.44 12.06 24297100     11.32
3 2009-03-02 11.68 15.24  9.71 13.63 31136400     12.79

$GS
        Date  Open   High   Low  Close   Volume Adj.Close
1 2009-01-02 84.02  92.20 59.13  80.73 22764300     77.85
2 2009-02-02 78.78  98.66 78.57  91.08 28301500     88.31
3 2009-03-02 87.86 115.65 72.78 106.02 30196400    102.79
```

Cool, huh? I find reshape much easier to use than other functions for reshaping data. Now that we've seen how `melt` and `cast` work, let's dive into the two functions in more detail.

melt. `melt` is a generic function; the reshape package includes methods for data frames, arrays, and lists. Here's an overview of the arguments for each form.

```
melt.data.frame(data, id.vars, measure.vars, variable_name, na.rm,
  preserve.na, ...)
```

Here is a description of the arguments to `melt.data.frame`:

Study Material. Do not distribute.

Argument	Description	Default
data	The data frame to melt.	
id.vars	ID variables (variables used to identify each unique observation).	All non-measure variables. If neither id.vars nor measure.vars is specified, assumes all factor and character variables are measured.
measure.vars	Measured variables (variables that describe the thing being measured).	All non-ID variables. If neither id.vars nor measure.vars is specified, assumes all variables that are neither factor nor character variables are measured.
variable_name	The name of the variable that stores the names of the original variables.	`"variable"`
na.rm	Tells melt what to do with NA values.	`!preserve.na`
preserve.na	Deprecated; opposite of `na.rm`.	TRUE
...	Other arguments are ignored.	

For multi-dimensional arrays, `melt` is conceptually more simple. You simply need to specify the dimensions to keep, and `melt` will melt the array.

```
melt.array(data, varnames, ...)
```

Here is a description of the arguments to the array form:

Argument	Description	Default
data	The array to melt	
varnames	A vector	All dimensions (`dimnames(data)`)
...	Other arguments are ignored	

Finally, the list form of melt will recursively melt each element in the list, join the results, and return the joined form:

```
melt.list(data, ..., level)
```

Argument	Description	Default
data	The list of items to melt	
level		1
...	Other arguments are passed to recursive calls to melt	

Cast. After you have melted your data, you use `cast` to reshape the results. Here is a description of the arguments to `cast`:

```
cast(data, formula, fun.aggregate=NULL, ..., margins, subset, df, fill,
  add.missing, value = guess_value(data))
```

Study Material. Do not distribute.

Argument	Description	Default
data	A molten data frame (typically created by `melt`).	
formula	A description of the output data frame as a formula, in the form `x_variable + x_2 ~ y_variable + y_2 ~ z_variable ~ ... | list_variable + ...` Use an ellipsis to mean "all variables not otherwise mentioned in the formula" and "." to represent "no variables."	`...~variable`
fun.aggregate	An aggregation function. If you want to aggregate the molten data in the output, specify an aggregation function to describe how to aggregate the data.	`NULL`
...	Arguments passed to `fun.aggregate`.	
margins	Variables on which to compute margins, specified as a vector of variable names, or `TRUE` to use all variables or `FALSE` to use none.	`FALSE`
subset	A logical vector that describes which observations in the molten data to include in the cast form.	`TRUE`
df	"An argument used internally," according the to the documentation.	`FALSE`
fill	Value with which to fill in missing combinations when `add.missing=TRUE`	`NULL`
add.missing	Fill in missing combinations.	`FALSE`
value	Name of value column.	`guess_value(data)`

Data Cleaning

Even when data is in the right form, there are often surprises in the data. For example, I used to work with credit data in a financial services company. Valid credit scores (specifically, FICO credit scores) always fall between 340 and 840. However, our data often contained values like 997, 998, and 999. These values did not mean that the customer had really super credit; instead, they had special meanings like "insufficient data" or there might be duplicate records in the data. Again, suppose that you were analyzing data on patients at a hospital. Often, the same doctor might see multiple patients with the same first *and* last names, so multiple patients may be rolled up into a single record incorrectly. However, sometimes the same patient might see multiple doctors, creating multiple records in the database for the same patient.

Data cleaning doesn't mean changing the meaning of data. It means identifying problems caused by data collection, processing, and storage processes and modifying the data so that these problems don't interfere with analysis.

Finding and Removing Duplicates

Data sources often contain duplicate values. Depending on how you plan to use the data, the duplicates might cause problems. It's a good idea to check for duplicates in your data (if they aren't supposed to be there).

R provides some useful functions for detecting duplicate values.

Study Material. Do not distribute.

Suppose that you accidentally included one stock ticker twice (say, GE) when you fetched stock quotes:

```
> my.tickers.2 <- c("GE", "GOOG", "AAPL", "AXP", "GS", "GE")
> my.quotes.2 <- get.multiple.quotes(my.tickers.2, from=as.Date("2009-01-01"),
+   to=as.Date("2009-03-31"), interval="m")
```

R provides some useful functions for detecting duplicate values such as the `duplicated` function. This function returns a logical vector showing which elements are duplicates of values with lower indices. Let's apply `duplicated` to the data frame `my.quotes.2`:

```
> duplicated(my.quotes.2)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

As expected, `duplicated` shows that the last three rows are duplicates of earlier rows. You can use the resulting vector to remove duplicates:

```
> my.quotes.unique <- my.quotes.2[!duplicated(my.quotes.2),]
```

Alternatively, you could use the `unique` function to remove the duplicate values:

```
> my.quotes.unique <- unique(my.quotes.2)
```

Sorting

Two final operations that you might find useful for analysis are sorting and ranking functions.

To sort the elements of an object, use the `sort` function:

```
> w <- c(5, 4, 7, 2, 7, 1)
> sort(w)
[1] 1 2 4 5 7 7
```

Add the `decreasing=TRUE` option to sort in reverse order:

```
> sort(w, decreasing=TRUE)
[1] 7 7 5 4 2 1
```

You can control the treatment of `NA` values by setting the `na.last` argument:

```
> length(w)
[1] 6
> length(w) <- 7
> # note that by default, NA.last=NA and NA values are not shown
> sort(w)
[1] 1 2 4 5 7 7
> # set NA.last=TRUE to put NA values last
> sort(w, na.last=TRUE)
[1]  1  2  4  5  7  7 NA
> # set NA.last=FALSE to put NA values first
> sort(w, na.last=FALSE)
[1] NA  1  2  4  5  7  7
```

Study Material. Do not distribute.

Sorting data frames is somewhat nonintuitive. To sort a data frame, you need to create a permutation of the indices from the data frame and use these to fetch the rows of the data frame in the correct order. You can generate an appropriate permutation of the indices using the `order` function:

```
order(..., na.last = , decreasing = )
```

The `order` function takes a set of vectors as arguments. It sorts recursively by each vector, breaking ties by looking at successive vectors in the argument list. At the end, it returns a permutation of the indices of the vector corresponding to the sorted order. (The arguments `na.last` and `decreasing` work the same way as they do for `sort`.) To see what this means, let's use a simple example. First, we'll define a vector with two elements out of order:

```
> v <- c(11, 12, 13, 15, 14)
```

You can see that the first three elements (11, 12, 13) are in order, and the last two (15, 14) are reversed. Let's call `order` to see what it does:

```
> order(v)
[1] 1 2 3 5 4
```

This means "move row 1 to row 1, move row 2 to row 2, move row 3 to row 3, move row 4 to row 5, move row 5 to row 4." We can return a sorted version of `v` using an indexing operator:

```
> v[order(v)]
[1] 11 12 13 14 15
```

Suppose that we created the following data frame from the vector `v` and a second vector `u`:

```
> u <- c("pig", "cow", "duck", "horse", "rat")
> w <- data.frame(v, u)
> w
   v     u
1 11   pig
2 12   cow
3 13  duck
4 15 horse
5 14   rat
```

We could sort the data frame `w` by `v` using the following expression:

```
> w[order(w$v),]
   v     u
1 11   pig
2 12   cow
3 13  duck
5 14   rat
4 15 horse
```

As another example, let's sort the `my.quotes` data frame (that we created earlier) by closing price:

Study Material. Do not distribute.

```
> my.quotes[order(my.quotes$Close),]
   symbol       Date   Open   High    Low  Close    Volume Adj.Close
2      GE 2009-02-02  12.03  12.90   8.40   8.51 194928800      8.51
1      GE 2009-03-02   8.29  11.35   5.87  10.11 277426300     10.11
11    AXP 2009-02-02  16.35  18.27  11.44  12.06  24297100     11.90
3      GE 2009-01-02  16.51  17.24  11.87  12.13 117846700     11.78
10    AXP 2009-03-02  11.68  15.24   9.71  13.63  31136400     13.45
12    AXP 2009-01-02  18.57  21.38  14.72  16.73  19110000     16.51
15     GS 2009-01-02  84.02  92.20  59.13  80.73  22764300     80.29
8    AAPL 2009-02-02  89.10 103.00  86.51  89.31  27394900     89.31
9    AAPL 2009-01-02  85.88  97.17  78.20  90.13  33487900     90.13
14     GS 2009-02-02  78.78  98.66  78.57  91.08  28301500     91.08
7    AAPL 2009-03-02  88.12 109.98  82.33 105.12  25963400    105.12
13     GS 2009-03-02  87.86 115.65  72.78 106.02  30196400    106.02
5    GOOG 2009-02-02 334.29 381.00 329.55 337.99   6158100    337.99
6    GOOG 2009-01-02 308.60 352.33 282.75 338.53   5727600    338.53
4    GOOG 2009-03-02 333.33 359.16 289.45 348.06   5346800    348.06
```

You could sort by symbol and then by closing price using the following expression:

```
> my.quotes[order(my.quotes$symbol, my.quotes$Close),]
   symbol       Date   Open   High    Low  Close    Volume Adj.Close
2      GE 2009-02-02  12.03  12.90   8.40   8.51 194928800      8.51
1      GE 2009-03-02   8.29  11.35   5.87  10.11 277426300     10.11
3      GE 2009-01-02  16.51  17.24  11.87  12.13 117846700     11.78
5    GOOG 2009-02-02 334.29 381.00 329.55 337.99   6158100    337.99
6    GOOG 2009-01-02 308.60 352.33 282.75 338.53   5727600    338.53
4    GOOG 2009-03-02 333.33 359.16 289.45 348.06   5346800    348.06
8    AAPL 2009-02-02  89.10 103.00  86.51  89.31  27394900     89.31
9    AAPL 2009-01-02  85.88  97.17  78.20  90.13  33487900     90.13
7    AAPL 2009-03-02  88.12 109.98  82.33 105.12  25963400    105.12
11    AXP 2009-02-02  16.35  18.27  11.44  12.06  24297100     11.90
10    AXP 2009-03-02  11.68  15.24   9.71  13.63  31136400     13.45
12    AXP 2009-01-02  18.57  21.38  14.72  16.73  19110000     16.51
15     GS 2009-01-02  84.02  92.20  59.13  80.73  22764300     80.29
14     GS 2009-02-02  78.78  98.66  78.57  91.08  28301500     91.08
13     GS 2009-03-02  87.86 115.65  72.78 106.02  30196400    106.02
```

Sorting a whole data frame is a little strange. You can create a suitable permutation using the `order` function, but you need to call `order` using `do.call` for it to work properly. (The reason for this is that `order` expects a list of vectors and interprets the data frame as a single vector, not as a list of vectors.) Let's try sorting the `my.quotes` table we just created:

```
> # what happens when you call order on my.quotes directly: the data
> # frame is interpreted as a vector
> order(my.quotes)
  [1]  61  94  96  95  31  62  77 107  70  76 106  46  71  40 108  63
 [17] 116  32  86  78  47 115  85  72  55  41  33 117  87  48  56  42
 [33] 102  57 105 101  97  98 104 103 100  99  75  73  69  74  44 120
 [49]  90  67  45  39  68  43  37  38  83 113  84 114  89 119  60  54
 [65]  59  53  82 112  88 118  52  58  93  92  18  21  24  27  30  17
 [81]  20  23  26  29  16  19  22  25  28  91  66  64  36  65  34  35
 [97]  80 110  81 111  79 109  51  49  50   7   8   9  10  11  12   1
[113]   2   3   4   5   6  13  14  15
> # what you get when you use do.call:
```

Study Material. Do not distribute.

```
> do.call(order,my.quotes)
 [1]  3  2  1  6  5  4  9  8  7 12 11 10 15 14 13
> # now, return the sorted data frame using the permutation:
> my.quotes[do.call(order, my.quotes),]
   symbol       Date   Open   High    Low  Close    Volume Adj.Close
3      GE 2009-01-02  16.51  17.24  11.87  12.13 117846700     11.78
2      GE 2009-02-02  12.03  12.90   8.40   8.51 194928800      8.51
1      GE 2009-03-02   8.29  11.35   5.87  10.11 277426300     10.11
6    GOOG 2009-01-02 308.60 352.33 282.75 338.53   5727600    338.53
5    GOOG 2009-02-02 334.29 381.00 329.55 337.99   6158100    337.99
4    GOOG 2009-03-02 333.33 359.16 289.45 348.06   5346800    348.06
9    AAPL 2009-01-02  85.88  97.17  78.20  90.13  33487900     90.13
8    AAPL 2009-02-02  89.10 103.00  86.51  89.31  27394900     89.31
7    AAPL 2009-03-02  88.12 109.98  82.33 105.12  25963400    105.12
12    AXP 2009-01-02  18.57  21.38  14.72  16.73  19110000     16.51
11    AXP 2009-02-02  16.35  18.27  11.44  12.06  24297100     11.90
10    AXP 2009-03-02  11.68  15.24   9.71  13.63  31136400     13.45
15     GS 2009-01-02  84.02  92.20  59.13  80.73  22764300     80.29
14     GS 2009-02-02  78.78  98.66  78.57  91.08  28301500     91.08
13     GS 2009-03-02  87.86 115.65  72.78 106.02  30196400    106.02
```