2

Reading and Writing Data

2.1 Reading Vectors and Matrices

The `c` function has already been introduced as a way to input small amounts of data into R. When the amount of data is large, and especially when typing the data into the console is inappropriate, the `scan` function can be used. `scan` is most appropriate when all the data to be read is of the same mode, so that it can be accommodated by a vector or matrix. For reading data with variables of mixed modes, see Section 2.2.

The first argument to `scan` can be a quoted string or character variable containing the name of a file or a URL, or it can be any of a number of connections (Section 2.1) to allow other input sources. If no first argument is given, `scan` will read from the console, stopping when a completely blank line is entered.

By default, `scan` expects all of its input to be numeric data; this can be overridden with the `what=` argument, which specifies the type of data that `scan` will see. For example, to read a vector of character values with `scan`, you can specify `what=""`:

```
> names = scan(what="")
1: joe fred bob john
5: sam sue robin
8:
Read 7 items
> names
[1] "joe"   "fred"  "bob"   "john"  "sam"   "sue"   "robin"
```

When reading from the console, R will prompt you with the index of the next item to be entered, and report on the number of elements read when it's done.

If the `what=` argument to `scan` is a list containing examples of the expected data types, `scan` will output a list with as many elements as there are data types provided. To specify numeric values, you can pass a value of 0:

```
> names = scan(what=list(a=0,b="",c=0))
1: 1 dog 3
2: 2 cat 5
3: 3 duck 7
4:
Read 3 records
> names
$a
[1] 1 2 3

$b
[1] "dog"  "cat"  "duck"

$c
[1] 3 5 7
```

Note that, by naming the elements in the list passed through the `what=` argument, the output list elements are appropriately named. When the argument to `what=` is a list, the `multi.line=` option can be set to `FALSE` to prevent `scan` from trying to use multiple lines to read the records for an observation.

One of the most common uses for `scan` is to read in data matrices. Since `scan` returns a vector, a call to `scan` can be embedded inside a call to the `matrix` function:

```
> mymat = matrix(scan(),ncol=3,byrow=TRUE)
1: 19 17 12
4: 15 18 9
7: 9 10 14
10: 7 12 15
13:
Read 12 items
> mymat
     [,1] [,2] [,3]
[1,]   19   17   12
[2,]   15   18    9
[3,]    9   10   14
[4,]    7   12   15
```

Notice the use of the `byrow=TRUE` argument. This allows the vector to be converted to a matrix in the way that such data is usually presented.

In order to skip fields while reading in data with `scan` , a type of `NULL` can be used in the list passed to the `what=` argument. Suppose we have a large data file with ten numeric fields on each line, but we only need to read the contents of the first, third, and tenth fields. We could use a call to `scan` as follows:

```
> values = scan(filename,
+             what=c(f1=0,NULL,f3=0,rep(list(NULL),6),f10=0))
```

Since a value of NULL will not be replicated by rep, multiple NULL values are added as lists, and the c function properly integrates them into the list passed to scan. Once the file is read in this way, a matrix with the extracted fields could be constructed with the cbind function:

```
result = cbind(values$f1,values$f3,values$f10)
```

2.2 Data Frames: read.table

The read.table function is used to read data into R in the form of a data frame. read.table always returns a data frame, which means that it is ideally suited to read data with mixed modes. (For data of a single mode, like numeric matrices, it is more efficient to use scan.) read.table expects each field (variable) in the input source to be separated by one or more separators, by default any of spaces, tabs, newlines or carriage returns. The sep= argument can be used to specify an alternative separator. (See Section 2.3 for convenience functions designed for comma- or tab-separated data.) If there are no consistent separators in the input data, but each variable occupies the same columns for every observation, the read.fwf function, described in Section 2.4, can be used.

If the first line in your input data consists of variable names separated by the same separator as the data, the header=TRUE argument can be passed to read.table to use these names to identify the columns of the output data frame. Alternatively, the col.names= argument to read.table can specify a character vector containing the variable names. Without other guidance, read.table will name the variables using a V followed by the column number.

The only required argument to read.table is a file name, URL, or connection object (See Section 2.1). Under Windows, make sure that double backslashes are used in pathnames, since a single backslash in a character string in R indicates that the next character should be treated specially. If your data is in the standard format as described above, that should be all that read.table needs, with the possible addition of header=TRUE. However, read.table is very flexible, and you may sometimes need to make adjustments using the features described below.

Because it offers increased efficiency in storage, read.table automatically converts character variables to factors. This may cause some problems when trying to use the variables as simple character strings. While this can usually be resolved using the methods discussed in Chapter 5, you can prevent conversion to factors by using the stringsAsFactors= argument. Passing the value FALSE through this argument will prevent any factor conversion. To insure that character variables are never converted to factors, the system option stringsAsFactors can be set to FALSE using

```
> options(stringsAsFactors=FALSE)
```

The `as.is=` argument can be used to suppress factor conversion for a subset of the variables in your data, by supplying a vector of indices specifying the columns not to be converted, or a logical vector with length equal to the number of columns to be read and `TRUE` wherever factor conversion is to be suppressed. You may notice a speedup in reading your data if you suppress some or all of the factor conversion, at the cost of increased storage space.

The `row.names=` argument can be used to pass a vector of character values to be used as row names to identify the output and which can be used instead of numeric subscripts when indexing the data frame. (See Section 6.1.) An argument of `row.names=NULL` will use a character representation of the observation number for the row names.

`read.table` will automatically treat the symbol `NA` as representing a missing value for any data type, and `NaN`, `Inf` and `-Inf` as missing for numeric data. To modify this behavior, the `na.strings` argument can be passed a vector of character values that should be interpreted as representing missing values.

By default, `read.table` will treat any text after a pound sign (`#`) as a comment. You can change the character used as a comment character through the `comment.char=` argument. If your input source doesn't contain any comments, setting `comment.char=''` may speed up reading your data.

For locales which use a character other than the period (`.`) as a decimal point, the `dec=` argument can be used to specify an alternative. The `encoding=` argument can be used to interpret non-ASCII characters in your input data.

You can control which lines are read from your input source using the `skip=` argument that specifies a number of lines to skip at the beginning of your file, and the `nrows=` argument which specifies the maximum number of rows to read. For very large inputs, specifying a value for `nrows=` which is close to but greater than the number of rows to be read may provide an increase in speed.

`read.table` expects the same number of fields on each line, and will report an error if it detects otherwise. If the unequal numbers of fields are due to the fact that some observations naturally have more variables than others, the `fill=TRUE` argument can be used to fill in observations with fewer variables using blanks or `NA`s. If `read.table` reports that there are unequal numbers of fields on some of the lines, the `count.fields` function can often help determine where the problem is.

`read.table` accepts a `colClasses=` argument, similar to the `what=` argument of `scan`, to specify the modes of the columns to be read. Since `read.table` will automatically recognize character and numeric data, this argument is most useful when you want to perform more complex conversions as the data is being read, or if you need to skip some of the fields in your input connection. Explicitly declaring the types of the columns may also improve the efficiency of reading data. To specify the column classes, provide a vector of character values representing the data types; any type for which

there is an "`as.`" method (See Section 1.3) can be used. A value of `"NULL"` instructs `read.table` to skip that column, and a value of `NA` (unquoted) lets `read.table` decide the format to use when reading that column.

2.3 Comma- and Tab-Delimited Input Files

For the common cases of reading in data whose fields are separated by commas or tabs, R provides three convenience functions, `read.csv`, `read.csv2`, and `read.delim`. These functions are wrappers for `read.table`, with appropriate arguments set for comma-, semicolon-, or tab-delimited data, respectively. Since these functions will accept any of the optional arguments to `read.table`, they are often more convenient than using `read.table` and setting the appropriate arguments manually.

2.4 Fixed-Width Input Files

Although not as common as white-space-, tab-, or comma-separated data, sometimes input data is stored with no delimiters between the values, but with each variable occupying the same columns on each line of input. In cases like this, the `read.fwf` function can be used. The `widths=` argument can be a vector containing the widths of the fields to be read, using negative numbers to indicate columns to be skipped. If the data for each observation occupies more than one line, `widths=` can be a list of as many vectors as there are lines per observation. The `header=`, `row.names=`, and `col.names=` arguments behave similarly to those in `read.table`.

To illustrate the use of `read.fwf`, consider the following lines, showing the 10 counties of the United States with the highest population density (measured in population per square mile):

```
New York, NY                    66,834.6
Kings, NY                       34,722.9
Bronx, NY                       31,729.8
Queens, NY                      20,453.0
San Francisco, CA               16,526.2
Hudson, NJ                      12,956.9
Suffolk, MA                     11,691.6
Philadelphia, PA                11,241.1
Washington, DC                   9,378.0
Alexandria IC, VA                8,552.2
```

Since the county names contain blanks and are not surrounded by quotes, `read.table` will have difficulty reading the data. However, since the names are always in the same columns, we can use `read.fwf`. The commas in the population values will force `read.fwf` to treat them as character values, and,

like `read.table`, it will convert them to factors, which may prove inconvenient later. If we wanted to extract the state values from the county names, we might want to suppress factor conversion for these values as well, and `as.is=TRUE` will be used. Assuming that the data is stored in a file named `city.txt`, the values could be read as follows:

```
> city = read.fwf("city.txt",widths=c(18,-19,8),as.is=TRUE)
> city
                   V1       V2
1  New York, NY       66,834.6
2  Kings, NY          34,722.9
3  Bronx, NY          31,729.8
4  Queens, NY         20,453.0
5  San Francisco, CA  16,526.2
6  Hudson, NJ         12,956.9
7  Suffolk, MA        11,691.6
8  Philadelphia, PA   11,241.1
9  Washington, DC      9,378.0
10 Alexandria IC, VA   8,552.2
```

Before using `V2` as a numeric variable, the commas would need to be removed using `gsub` (see Section 7.8):

```
> city$V2 = as.numeric(gsub(',','',city$V2))
```

2.5 Extracting Data from R Objects

While previous sections have discussed working with data stored in built-in classes, R provides two mechanisms for developers to define their own classes, so it's important to understand how data is stored in such objects. The class mechanisms in R provide some of the features of object-oriented programming, namely, method dispatch and inheritance. Method dispatch allows R to examine the class of the arguments to a function, and to invoke a special version of the function designed for that class of object. Not all functions in R provide method dispatch; the ones that do are known as generic functions. Inheritance allows developers to create new classes that are similar to other classes; only methods that differ from the original class need to be provided. When an object in R inherits the properties of an already defined object, its class attribute will be a vector containing the object's class (in the first position), along with the classes from which it inherits.

In the first mechanism for object orientation in R, known as S3 or "old-style" classes, method dispatch is implemented for generic functions by the existence of a function whose name is of the form `function.class`. If no such function is found in the search path, a function whose name is of the form `function.default` will be invoked, and default functions exist for all the S3

generics. S3 generic functions can be recognized because their body consists of a call to the `UseMethod` function, which actually performs the dispatch. It's important to recognize when generic functions are being called, because help pages for specific method/object combinations may be available through their "full" names. For example, the help page for the `summary` function doesn't discuss any properties of the method that will be invoked when `summary` is passed an `lm` object; the help page for `summary.lm` would have to be accessed directly. Even though you may refer to these specific methods to view their help pages, it's rarely if ever necessary to call them directly—they should always be called through the generic function.

As an example, consider the `lm` function, which performs linear model calculations. A object returned by this function will have class of `lm`; when the object is printed or displayed, R will look for a function called `print.lm`, which will display appropriate information about the linear model which was fit. For example, the following code produces an `lm` object, and then displays it through the `print` function:

```
> slm = lm(stack.loss ~ Air.Flow + Water.Temp,data=stackloss)
> class(slm)
[1] "lm"
> slm

Call:
lm(formula = stack.loss ~ Air.Flow + Water.Temp,
    data = stackloss)

Coefficients:
(Intercept)     Air.Flow   Water.Temp
   -50.3588       0.6712       1.2954
```

When a class is created, a set of functions to extract data from objects of that class is usually also provided. These so-called accessor functions are the recommended way to extract information from objects in R, since they will provide a stable interface even if the internal structure of the object changes. Because of the naming convention used in S3 method dispatch, the `apropos` function can be used to find all the available methods for a given class:

```
> apropos('.*\\.lm$')
 [1] "anovalist.lm"    "anova.lm"        "hatvalues.lm"
 [4] "model.frame.lm"  "model.matrix.lm" "plot.lm"
 [7] "predict.lm"      "print.lm"        "residuals.lm"
[10] "rstandard.lm"    "rstudent.lm"     "summary.lm"
[13] "kappa.lm"
```

(If any functions are marked nonvisible, the `getAnywhere` function can be used to see them.) Thus, to get the predicted values from an `lm` object, the `predict` function will dispatch to `predict.lm`. It's worth repeating that you should avoid calling a function like `predict.lm` directly in favor of relying on

the generic function (in this case, `predict`). Also note that if an object has multiple classes, you should look for relevant functions designed for any of the classes from which the object inherits.

Most S3 objects are stored as lists, so if an appropriate accessor function is not available, data can be extracted directly from the object by treating it as a list. The first step in these cases is to use the `names` function to find the available elements:

```
> names(slm)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

Now, for example, we could find the residual degrees of freedom for the model by extracting it directly:

```
> slm$df.residual
[1] 18
```

or

```
> slm['df.residual']
[1] 18
```

Since the method dispatch provided by the old-style classes is limited to using only the first argument to the function, and since the naming conventions can sometimes lead to confusion, a more formal method of defining classes (known as "new-style" or S4 classes) has also been developed. This is the preferred way to implement new classes in R, and will become much more prevalent over time. Some of the functions required to work with new-style classes are found in the `methods` package, so if they are not available, they can be loaded using

```
> library(methods)
```

With S4 classes, generic functions can be identified by the presence of a call to the `standardGeneric` function inside the generic function's definition.

As an example of an S4 class, consider the `mle` function, used for maximum likelihood estimation, and found in the `stats4` package. We'll simulate data from a gamma distribution, and then use `mle` to estimate the parameters for that distribution:

```
> library(stats4)
> set.seed(19)
> gamdata = rgamma(100,shape=1.5,rate=5)
> loglik = function(shape=1.5,rate=5)
+          -sum(dgamma(gamdata,shape=shape,rate=rate,log=TRUE))
> mgam = mle(loglik)
```

The `isS4` function can be used to determine whether or not an object is using the old-style or new-style classes:

```
> class(mgam)
[1] "mle"
attr(,"package")
[1] "stats4"
> isS4(mgam)
[1] TRUE
```

As with old-style methods the first choice for accessing information from an S4 class should always be using the accessor functions provided along with the function that created the object. For S4 classes, it's easy to find the available methods using the `showMethods` function (from the `methods` package):

```
> showMethods(class='mle')
Function: coef (package stats)
object="mle"

Function: confint (package stats)
object="mle"

Function: initialize (package methods)
.Object="mle"
    (inherited from: .Object="ANY")

Function: logLik (package stats)
object="mle"

Function: profile (package stats)
fitted="mle"

Function: show (package methods)
object="mle"

Function: summary (package base)
object="mle"

Function: update (package stats)
object="mle"

Function: vcov (package stats)
object="mle"
```

Thus, for example, the variance–covariance matrix of the estimators is available by using the `vcov` function:

```
> vcov(mgam)
            shape      rate
shape 0.05464054 0.1724472
rate  0.17244719 0.7228044
```

Naturally, the help page for the function that produced the object, as well as additional help pages describing the class (if available), can be consulted for additional information.

Although there is no generic `print` function for S4 classes, the generic `show` function takes its place for printing or displaying S4 classes.

The actual entities that compose an S4 object are stored in so-called slots. To see the available slots in an object, the `showClass` function can be used. If it becomes necessary to access the slots directly, the `@` operator can be used in a fashion similar to the `$` operator. Following the `mle` example, suppose we wanted to retrieve the function that was used to calculate the likelihood which was stored in `mgam`:

```
> getClass(class(mgam))
Slots:

Name:       call      coef  fullcoef      vcov       min
Class:  language   numeric   numeric    matrix   numeric

Name:    details minuslogl    method
Class:      list  function character
> mgam@minuslogl
function(shape=1.5,rate=5)
    -sum(dgamma(gamdata,shape=shape,rate=rate,log=TRUE))
```

The `slot` function can be used if the name of the desired slot is stored in a character variable:

```
> want = 'minuslogl'
> slot(mgam,want)
function(shape=1.5,rate=5)
    -sum(dgamma(gamdata,shape=shape,rate=rate,log=TRUE))
```

For both styles of classes, some of the methods provided may create objects containing additional information about the objects they operate on. This is especially true for the `summary` method for many objects. Returning to the `lm` example, we can examine what's available through the `summary` method by creating a `summary` object from the `lm` object, and examining its names:

```
> sslm = summary(slm)
> class(sslm)
[1] "summary.lm"
```

```
> names(sslm)
 [1] "call"          "terms"         "residuals"
 [4] "coefficients"  "aliased"       "sigma"
 [7] "df"            "r.squared"     "adj.r.squared"
[10] "fstatistic"    "cov.unscaled"
```

As can be seen, a number of useful quantities have been calculated through the summary method, and are available through their named components in the resulting `summary.lm` object.

2.6 Connections

Connections provide a flexible way for R to read data from a variety of sources, providing more complete control over the nature of the connection than simply specifying a file name as input to functions like `read.table` and `scan`. Table 2.1 lists some of the functions in R which can create a connection.

Function	Data source
`file`	Files on the local file system
`pipe`	Output from a command
`textConnection`	Treats text as a file
`gzfile`	Local gzipped file
`unz`	Local zip archive (with single file;read-only)
`bzfile`	Local bzipped file
`url`	Remote file read via http
`socketConnection`	socket for client/server programs

Table 2.1. Connections

When you create a connection object, it simply defines the object; it does not automatically open the object. If a function which accepts connections receives a connection which has not been opened, it will always open it, then close it at the end of the function invocation. Thus, in the usual case, you can simply pass the connection to the function that will operate on it, without worrying about when the connection will be opened or closed. If a connection doesn't behave the way you expect, or if you're not sure if you've already closed it, the `isOpen` function can be used to test if a connection is open; an optional second argument, set equal to `"read"` or `"write"` can test the mode with which it was opened.

One exception to this scheme is the case where a file is read in pieces, for example through the `readLines` function. If an (unopened) connection is passed to this function inside a loop, it will repeatedly open and close the file each time it's called, reading the same data over and over. To take more control over the connection, it can either be passed to the `open` function, or

an optional mode can be provided as the second argument to the function that created the connection. Note that in this case, the connection will not be automatically closed; you must explicitly pass the connection to the `close` function.

As an illustration of this technique, consider the R project homepage, `http://www.r-project.org/main.shtml`. The latest version number of R is displayed on that page, followed by the phrase "has been released". The following program opens the connection to this URL by passing a mode of "`r`" for read, then reads each line until it finds the one with the latest version number:

```
> rpage = url('http://www.r-project.org/main.shtml','r')
> while(1){
+     l = readLines(rpage,1)
+     if(length(l) == 0)break;
+     if(regexpr('has been released',l) > -1){
+             ver = sub('</a.*$','',l)
+             print(gsub('^ *','',ver))
+             break
+     }
+ }
[1] "R version 2.2.1"
> close(rpage)
```

The second argument to `readLines` specifies the number of lines to read, where a value of `-1` means to read everything that the connection provides. Although it may seem inefficient to read only one line at a time, the actual reads are being performed by the operating system and are buffered in memory, so that you can choose whatever number of lines is most convenient. By using this technique, we only need to process as much of the connection as necessary.

Note that connections can be used anywhere a file name could be passed to functions like `scan`, `read.table`, `write.table`, and `cat`. So to write a gzipped, comma-separated version of a data frame, we could use:

```
gfile = gzfile('mydata.gz')
write.table(mydata,sep=',',file=gfile)
```

The `write.table` function takes care of opening and closing the gzipped file, since it was not explicitly opened.

A `textConnection` can often be useful when you need to test a function that only operates on files. For example, in Section 2.2, the `colClasses` argument was introduced as a way to automatically convert data into appropriate R objects. Suppose we want to test conversion to `Date` objects (Section 4.1) using this argument. First, we create a `textConnection` with the kind of data we'll be using:

```
> sample = textConnection('2000-2-29 1 0
+ 2002-4-29 1 5
+ 2004-10-4 2 0')
```

Now we can use `sample` in place of a file name in any function that's expecting a file name:

```
> read.table(sample,colClasses=c('Date',NA,NA))
         V1 V2 V3
1 2000-02-29  1  0
2 2002-04-29  1  5
3 2004-10-04  2  0
```

The `unz` function allows read-only access to zipfiles. Since a zipfile is an archive, potentially containing many files, an additional argument to `unz` is required to specify which file you wish to extract. For example, suppose we have a zip file called `data.zip` containing several files, and we wish to create a connection to read the file called `mydata.txt` into a vector with the `scan` function. The following code could be used:

```
mydata = scan(unz('data.zip','mydata.txt'))
```

Only one file can be extracted at a time with `unz`.

When you need to explicitly open a connection (either by passing a mode to the function that creates the connection, or calling `open` directly), you can specify any of the following modes: `"w"` for write, `"r"` for read, or `"a"` for append. You can append a `t` to the end of any of these modes to specify a text connection, or a `b` to specify a binary connection. While there is no distinction between text and binary on UNIX systems, on Windows it is required to include a `b` anytime you want to operate on a file that has any nonprintable characters. In addition, specifying a binary file for writing on Windows will cause R to use UNIX-style line endings (a single newline) instead of Windows-style line endings (newline plus carriage return, sometimes displayed as control-M on UNIX systems). For more complex situations, such as opening a file for both reading and writing, consult the help file for `open`.

2.7 Reading Large Data Files

Since `readLines` and `scan` don't need to read an entire file into memory, there are situations where very large files can be processed by R in pieces. For example, suppose we have a large file containing numeric variables, and we wish to read a random sample of that file into R. Of course, if we could accommodate the entire dataset in memory, a call to `sample` (Section 2.9.1) could extract such a sample, but we'll assume that the file in question is too large to read into R in its entirety. The strategy is to select a random sample of rows before reading the data, and then extracting the selected rows

as the dataset is being read in pieces. To avoid memory allocation problems, the entire matrix is preallocated before the reading begins. These ideas are implemented in the following function:

```
readbig = function(file,samplesz,chunksz,nrec=0){
   if(nrec <= 0)nrec = length(count.fields(file))
   f = file(file,'r')
   on.exit(close(f))
   use = sort(sample(nrec,samplesz))
   now = readLines(f,1)
   k = length(strsplit(now,' +')[[1]])
   seek(f,0)

   result = matrix(0,samplesz,k)

   read = 0
   left = nrec
   got = 1
   while(left > 0){
        now = matrix(scan(f,n=chunksz*k),ncol=k,byrow=TRUE)
        begin = read + 1
        end = read + chunksz
        want = (begin:end)[begin:end %in% use] - read
        if(length(want) > 0){
           nowdat = now[want,]
           newgot =  got + length(want) - 1
           result[got:newgot,] = nowdat
           got = newgot + 1
        }
        read = read + chunksz
        left = left - chunksz
   }
   return(result)
}
```

If the number of records in the file, `nrec`, is specified as zero or a negative number, the function calculates the number of lines in the file through a call to `count.fields`; your operating system may provide a more efficient way of achieving this, such as the `wc -l` command on Linux or Mac OS X, or the `find /c` command on Windows, searching for the separator character which will be present on each line. Suppose we have a comma-separated file called `comma.txt` in the current directory. Under Windows, we could calculate the number of lines in the file using

```
> nrec = as.numeric(shell('type "comma.txt" | find /c ","',
+                         intern=TRUE))
```

while under UNIX-like systems, the command would be

```
> nrec = as.numeric(system('cat comma.txt | wc -l',
+                          intern=TRUE))
```

To calculate the number of columns in the file, a single line is read using `readLines`, and the `strsplit` function is called with an appropriate separator, in this case one or more blanks. Then the file is repositioned to its origin with the `seek` command, to prepare to actually read the data.

For optimum results, the `chunksz` argument can be adjusted for a given situation, but most reasonable values will result in acceptable performance.

2.8 Generating Data

Even though one of the main motivations in learning or working with R is to analyze existing data, sometimes it may be advantageous to create a data set from within R. This would be necessary, for example, to carry out a simulation, but may also be useful if you want to test a new technique or determine if a program may be appropriate for a very large dataset. In the following subsections, we'll look at a number of ways in R to generate vectors of data which can be used for simulations or testing programs when "real" data isn't available.

2.8.1 Sequences

To generate a sequence of integers between two values, the colon operator (`:`) can be used. For example, to create a vector of the integers from 1 to 10, we can use

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```

To get more control over the sequence, the `seq` function can be used. This function allows an optional increment through the `by=` argument, as well as more options for determining the length of the output sequence. In its simplest form, it behaves like the colon operator:

```
> seq(1,10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

To create a vector of values from 10 to 100, each element separated by 5, we could use:

```
> seq(10,100,5)
 [1]  10  15  20  25  30  35  40  45  50  55  60  65  70  75
[15]  80  85  90  95 100
```

Alternatively, we can specify the length of the generated sequence instead of providing the end point:

```
> seq(10,by=5,length=10)
 [1] 10 15 20 25 30 35 40 45 50 55
```

One common use of sequences is to generate factors corresponding to the levels of a designed experiment. Suppose we wish to simulate the levels for an experiment with three groups and five subgroups, with two observations in each subgroup, for a total of 30 observations. The `gl` function (mnemonic for "generate levels") has two required arguments. The first is the number of different levels desired, and the second is the number of times each level needs to be repeated. An optional third argument specifies the length of the output vector. To generate a data frame composed of vectors representing the three groups, five subgroups, and two observations, we could use `gl` as follows:

```
> thelevels = data.frame(group=gl(3,10,length=30),
+                        subgroup=gl(5,2,length=30),
+                        obs=gl(2,1,length=30))
> head(thelevels)
  group subgroup obs
1     1        1   1
2     1        1   2
3     1        2   1
4     1        2   2
5     1        3   1
6     1        3   2
```

Further control over the output of `gl` can be obtained with the optional `labels=` argument. `gl` also accepts an `ordered=TRUE` argument to produce ordered factors.

To create a data frame with the unique combinations defined by a collection of sequences, the `expand.grid` function can be used. This function accepts any number of sequences, and returns a data frame with one row for each unique combination of the values passed as input. Alternatively, all of the vectors can be passed to `expand.grid` as a single list. Suppose we wanted to create a data frame with one observation for each combination of odd and even integers between 1 and 5. We could use `expand.grid` as follows:

```
> oe = expand.grid(odd=seq(1,5,by=2),even=seq(2,5,by=2))
> oe
  odd even
1   1    2
2   3    2
3   5    2
4   1    4
5   3    4
6   5    4
```

Note that the column generated from the first argument varies the most quickly, and that the sequences passed to `expand.grid` need not be of the same length. The number of rows in the output data frame will always be equal to the product of the length of all the sequences passed to `expand.grid`.

One important use of data frames produced by `expand.grid` is to evaluate a function over a range of parameter values. The `apply` function, discussed in Section 8.4, is the most effective tool for this purpose. For example, suppose we wanted to evaluate the function $x^2 + y^2$ for various values of `x` and `y` in the range of 0 to 10. First, we can generate a matrix of input values using `expand.grid`:

```
> input = expand.grid(x=0:10,y=0:10)
```

Now we can use `apply` to calculate the function for each row of the data frame returned by `expand.grid`, and use `cbind` (Section 9.6) to combine it to the input data:

```
> res = apply(input,1,function(row)row[1]^2 + row[2]^2)
> head(cbind(input,res))
  x y res
1 0 0   0
2 1 0   1
3 2 0   4
4 3 0   9
5 4 0  16
6 5 0  25
```

2.8.2 Random Numbers

Function	Distribution	Function	Distribution
`rbeta`	Beta	`rlogis`	Logistic
`rbinom`	Binomial	`rmultinom`	Multinomial
`rcauchy`	Cauchy	`rnbinom`	Negative Binomial
`rchisq`	Chi-square	`rnorm`	Normal
`rexp`	Exponential	`rpois`	Poisson
`rf`	F	`rsignrank`	Signed Rank
`rgamma`	Gamma	`rt`	Student's t
`rgeom`	Geometric	`runif`	Uniform
`rhyper`	Hypergeometric	`rweibull`	Weibull
`rlnorm`	Log Normal	`rwilcox`	Wilcoxon Rank Sum

Table 2.2. Random number generators

If you are creating a dataset for a simulation, or to test a function in R for which there is no real data available, R provides a large number of

random number generators, listed in Table 2.2. The first argument to all of the random number generation functions is the number of random numbers desired; additional arguments allow specification of the parameters of the underlying distribution, and will vary depending on which distribution you are working with. Consult the help file for the function for further details.

The state of the random number generator that underlies all of the functions in Table 2.2 is stored in an object called `.Random.seed`. To create a reproducible sequence, an integer can be passed to the `set.seed` function insuring that an identical stream of random numbers will be generated whenever `set.seed` is set to the same value.

2.9 Permutations

2.9.1 Random Permutations

The `sample` function conveniently provides random permutations of either a vector of values (if the first argument is a vector), or of indices starting from one (if the first argument is a single number). With only one argument, `sample` returns a vector of length equal to the number of elements (if the argument is a vector) or the value of the argument if it is a number, sampling without replacement, so that each element of the input will appear exactly once in the output. Two optional arguments can override these defaults; the `size=` argument will return a vector of the designated size, and the `replace=` argument, if set to `TRUE`, will allow the possibility of elements of the input to appear more than once in a given sample. If the size of the desired sample is greater than the number of elements implied by the first argument, the `replace=` must be set to `TRUE`. Finally, if some elements of the input should be sampled at a higher probability than others, the optional `prob=` argument can provide a vector of sampling probabilities.

2.9.2 Enumerating All Permutations

Since the `sample` function provides a random permutation of its input, it will not be very effective in generating all possible permutations for a sequence, since it is possible that some permutations will appear more often than others. In cases like this, the `permn` function of the `combinat` package (available from CRAN) can be used. Like `sample`, the first argument to `permn` is either a vector or a single number. Called with a single argument, it returns a list containing each possible permutation of the input sequence. The optional `fun=` argument can be used to specify a function which will be applied to each permutation in the output list. Since the number of permutations of n elements is $n!$ (n factorial), the size of the output from `permn` can be very large, even for moderate values of n. The `factorial` function (or the `fact` function

of the `combinat` package) can be used to calculate how many permutations exist.

If the number of permutations generated is too large to be accommodated in memory, the `numperm` function of the `sna` package(available from CRAN) can be used. This function accepts two arguments: the first represents the length of the sequence, and the second represents the specific permutation desired. Thus, if the entire set of permutations is too large to be held in memory, the `numperm` function can be used inside a loop which successively increments an index (the second argument to `numperm`) from 1 to `factorial(n)`, where `n` is the length of the sequence.

2.10 Working with Sequences

R provides several functions which are useful when working with sequences of numbers. The `table` function, introduced in Section 8.1, can tabulate the number of occurrences of each value in a sequence. To get just the unique values in a sequence, the `unique` function can be used. Alternatively, the `duplicated` function can be used to return a vector of logical values indicating whether each value in the sequence is duplicated; `!duplicated(x)` will return a logical vector which will be true for the unique values. In both cases, the results will be in the order that the values are encountered in the vector being studied.

The `rle` (run-length encoding) function can be used to solve a variety of problems regarding consecutive identical values in a sequence. The returned value from `rle` is a list with two components: `values`, a vector which contains the repeated values that were found, and `lengths`, a vector of the same length as `values` which tells how many consecutive values were observed. If there are no repeated values, all of the elements of `lengths` will be 1:

```
> sequence = sample(1:10)
> rle(sequence)
Run Length Encoding
  lengths: int [1:10] 1 1 1 1 1 1 1 1 1 1
  values : int [1:10] 10 5 2 8 3 1 7 4 6 9
```

As an example of the use of `rle`, suppose we have a sequence of integers, and we wish to know if there are 3 or more consecutive appearances of the value 2. Considering the return value of `rle`, this will mean that 2 will appear in the `values` component returned by `rle` with a corresponding entry in the `lengths` component with a value of three or more:

```
> seq1 = c(1,3,5,2,4,2,2,2,7,6)
> rle.seq1 = rle(seq1)
> any(rle.seq1$values == 2 & rle.seq1$lengths >= 3)
[1] TRUE
```

```
> seq2 = c(7,5,3,2,1,2,2,3,5,8)
> rle.seq2 = rle(seq2)
> any(rle.seq2$values == 2 & rle.seq2$lengths >= 3)
[1] FALSE
```

To find the location within a sequence where a particular combination of `values` and `lengths` occurs, the `cumsum` function can be applied to the `lengths` component of the returned value of `rle`. The returned value from `cumsum` will provide the index where each set of repeated values terminates; thus by using the `which` function as an index into this vector, we can find the end points of any desired run of values.

Continuing with the previous example, we can find the index into `seq1` where a sequence of more than three values of `2` terminates:

```
> seq1 = c(1,3,5,2,4,2,2,2,7,6)
> rle.seq1 = rle(seq1)
> index = which(rle.seq1$values == 2 & rle.seq1$lengths >= 3)
> cumsum(rle.seq1$lengths)[index]
[1] 8
```

This indicates that the run of three or more values of `2` ended at position 8. To find the index where a run began, we can adjust the subscript used to cumsum, taking special care to properly handle runs at the beginning of a sequence:

```
> index = which(rle.seq1$values == 2 & rle.seq1$lengths >= 3)
> newindex = ifelse(index > 1,index - 1,0)
> starts = cumsum(rle.seq1$lengths)[newindex] + 1
> if(0 %in% newindex)starts = c(1,starts)
> starts
```

Due to the vectorization of these operators, multiple runs can be accommodated by the same strategy:

```
> seq3 = c(2,2,2,2,3,5,2,7,8,2,2,2,4,5,9,2,2,2)
> rle.seq3 = rle(seq3)
> cumsum.seq3 = cumsum(rle.seq3$lengths)
> myruns = which(rle.seq3$values == 2 &
+                rle.seq3$lengths >= 3)
> ends = cumsum.seq3[myruns]
> newindex = ifelse(myruns > 1,myruns - 1,0)
> starts = cumsum.seq3[newindex] + 1
> if(0 %in% newindex)starts = c(1,starts)
> starts
[1]  1 10 16
> ends
[1]  4 12 18
```

For more complex situations, a logical expression can often be used as an argument to `rle`. For example, to find the location of five or more successive values greater than zero in a sequence of random numbers, we can use the following approach:

```
> set.seed(19)
> randvals = rnorm(100)
> rle.randvals = rle(randvals > 0)
> myruns = which(rle.randvals$values == TRUE &
+                rle.randvals$lengths >= 5)
> any(myruns)
[1] TRUE
> cumsum.randvals = cumsum(rle.randvals$lengths)
> ends = cumsum.randvals[myruns]
> newindex = ifelse(myruns > 1,myruns - 1,0)
> starts = cumsum.randvals[newindex] + 1
> if(0 %in% newindex)starts = c(1,starts)
> starts
[1] 47
> ends
[1] 51
> randvals[starts:ends]
[1] 0.5783932 0.8276480 1.3111752 0.1783597 1.7036697
```

2.11 Spreadsheets

Spreadsheets, especially Microsoft Excel spreadsheets, are one of the most common methods of distributing data, and R provides several ways to access them. The simplest method, which may also be the most flexible, is to use a spreadsheet program to write the data to a comma- or tab-separated file, and then use the methods described in Section 2.2. This method is especially useful when there is additional nondata material (like headings and notes) in the spreadsheet, since such material can be removed by editing the file derived from the spreadsheet.

There are some situations, however, where this may not be feasible. An updated spreadsheet may need to be accessed every day, or an analysis may require data that comes from several spreadsheets, or from several sheets within a single spreadsheet. The next sections will look at ways of accessing spreadsheets through R functions, without the need to dump them to files.

2.11.1 The `RODBC` Package on Windows

On the Windows platform, spreadsheets can be read directly using the `ODBConnectExcel` function from the `RODBC` package, available from CRAN.

(For information about using RODBC to read databases, see Chapter 3.) This function provides an interface to Excel spreadsheets using the SQL language familiar to databases, and does not require Excel itself to be installed on your computer.

The `RODBC` interface treats the various sheets stored in a spreadsheet file as database tables. To use the interface, a connection object is obtained by providing the pathname of the Excel spreadsheet file to `odbcConnectExcel`. For example, suppose a spreadsheet is stored in the file `c:\Documents and Settings\user\My Documents\sheet.xls`. To get a connection object, we could use the following call:

```
> library(RODBC)
> sheet = 'c:\\Documents and Settings\\user\\My Documents
              \\sheet.xls'
> con = odbcConnectExcel(sheet)
```

Note the use of double slashes in the file name; this is used because the backslash has special meaning in R character strings, namely to inform R that certain characters need to be treated specially. Often the first step in working with spreadsheets in this way is to look at the names of the available sheets. This can be done with the `sqlTables` command. Continuing with the current example, we could find the names of the sheets in the `sheet.xls` spreadsheet by issuing the command

```
> tbls = sqlTables(con)
```

and then examining `tbls$TABLE_NAME`, the column of the returned data frame that contains the sheet names. From this point on, each of the sheets can be treated like a separate database table. Thus, to extract the contents of the first sheet of the database to a data frame called `data1`, we could use the following commands:

```
> qry = paste("SELECT * FROM",tbls$TABLE_NAME[1],sep=' ')
> result = sqlQuery(con,qry)
```

If the table name contains special characters, like spaces, brackets, or dollar signs, then it needs to be surrounded by backquotes (‘). As a precautionary measure, it may be advisable to include the backquotes in all queries:

```
> qry = paste("SELECT * FROM `",tbls$TABLE_NAME[1],"`",sep="")
> result = sqlQuery(con,qry)
```

Most SQL queries will work using this method.

2.11.2 The `gdata` Package (All Platforms)

An alternative to using the `RODBC` package is the `read.xls` function of the `gdata` package, available from CRAN. This function uses a module developed for the scripting language perl (`http://perl.org`), and thus requires perl to be installed on your computer. This will be the case for virtually all Mac OS

X, Unix, and Linux computers, but on Windows an installation of perl will be necessary. (The perl installer and instructions can be found on the above-referenced web site.) `read.xls` translates a specified sheet of a spreadsheet to a comma-separated file, and then calls `read.csv` (see Section 2.3). Thus, any option accepted by `read.csv` can be used with `read.xls`. The `skip=` and `header=` arguments are especially useful to avoid misinterpreting headers and notes as data, and the `as.is=TRUE` argument can be used to suppress factor conversion.

2.12 Saving and Loading R Data Objects

In situations where a good deal of processing must be used on a raw dataset in order to prepare it for analysis, it may be prudent to save the R objects you create in their internal binary form. One attractive feature of this scheme is that the objects created can be read by R programs running on different computer architectures than the one on which they were created, making it very easy to move your data between different computers. Each time an R session is completed, you are prompted to save the workspace image, which is a binary file called `.RData` in the working directory. Whenever R encounters such a file in the working directory at the beginning of a session, it automatically loads it making all your saved objects available again. So one method for saving your work is to always save your workspace image at the end of an R session. If you'd like to save your workspace image at some other time during your R session, you can use the `save.image` function, which, when called with no arguments, will also save the current workspace to a file called `.RData` in the working directory.

Sometimes it is desirable to save a subset of your workspace instead of the entire workspace. One option is to use the `rm` function to remove unwanted objects right before exiting your R session; another possibility is to use the `save` function. The `save` function accepts multiple arguments to specify the objects you wish to save, or, alternatively, a character vector with the names of the objects can be passed to `save` through the `list=` argument. Once the objects to be saved are specified, the only other required option is the `file=` option, specifying the destination of the saved R object. Although there is no requirement to do so, it is common to use a suffix of `.rda` or `.RData` for saved R workspace files.

For example, to save the R objects `x`, `y`, and `z` to a file called `mydata.rda`, the following statements could be used:

```
> save(x,y,z,file='mydata.rda')
```

If the names of the objects to be saved are stored as character vectors (for example, from the output of the `objects` function), the `list=` argument can be used:

```
> save(list=c('x','y','z'),file='mydata.rda')
```

Once the data is saved, it can be reloaded into a running R session with the `load` command, whose only required argument is the name of the file to be loaded. For example, to load the objects contained in the `mydata.rda` file, we can use the following command:

```
> load('mydata.rda')
```

The `load` command can also be used to load workspaces stored in `.RData` files in other directories by specifying their complete file path.

2.13 Working with Binary Files

While the natural way to store R objects is through the `save` command, other programs may also produce binary files which, not surprisingly, use their own format and are not readable by `load`. The `readBin` and `writeBin` functions provide a flexible way to read and write such files. It should be noted that a fairly complete knowledge of the format of a non-R binary file is required before `readBin` will be able to read it. However, for well-documented file formats, `readBin` should be able to access the full information contained in the file. Each call to `readBin` will read as many values as required, but a single call can only read one type of data. The types of data that `readBin` can understand include double precision numeric data, integers, character strings (although the `readChar` and `writeChar` functions provide additional flexibility), complex numbers, and raw data. Since multiple calls to `readBin` will have to be used if there is a mixture of data types in the file, it is usually necessary to pass a connection object to `readBin`, so that it will not automatically re-open the file each time it is called.

As an example of the use of `readBin`, consider a binary file called `data.bin`, which consists of 20 records, each containing one integer followed by five double-precision values. Such a file could be produced, for example, using the low-level `write` function in a C program. The first step is opening a file connection:

```
> bincon = file('data.bin','rb')
```

Note that R will not allow access to files through `readBin` or `writeBin` unless the file is opened in binary mode; thus the value of `'rb'` is passed to `file` to specify the mode. (The "`r`" stands for read, and the "`b`" stands for binary.)

For efficiency's sake, it's a good idea to preallocate memory for the vector or matrix which will hold the output from `readBin`. In this case, we can use a 20×6 matrix, and store the integer and five doubles in the rows of the matrix:

```
> result = matrix(0,20,6)
> for(i in 1:20){
+    theint = readBin(bincon,integer(),1)
+    thedoubles = readBin(bincon,double(),5)
+    result[i,] = c(theint,thedoubles)
+ }
> close(bincon)
```

As always when opening a connection inside of R, it's a good idea to close the file when you're done.

While no problems should be encountered if the data to be read was written on a computer with the same architecture as the one on which it is to be read, problems will sometimes occur if binary data from other architectures is used. There are two ways of storing data on a computer, depending on the order in which the bits of a binary value are stored; these two types are known as "little-endian" and "big-endian". Among some of the common architectures, x86 and its derivatives are little-endian, while the PowerPC and SPARC platforms are big-endian. `readBin` and `writeBin` each accept an `endian=` argument, which take values of `"big"`, `"little"`, or `"swap"`. (Note that endianness is not an issue when using the `save` and `load` commands since R uses the same format on all architectures for its saved objects.)

Writing binary files is essentially the reverse of reading them. `writeBin` can only write vectors of character, numeric, logical, or complex values; in particular lists or factors will need to be converted before writing.

As an example of using `writeBin`, consider a data frame constructed from the `state.x77` matrix:

```
> mystates = data.frame(name=row.names(state.x77),state.x77,
+                       row.names=NULL,stringsAsFactors=FALSE)
```

Note that the `stringsAsFactors=FALSE` argument was used to avoid factor conversion which could cause `writeBin` to fail. For existing datasets, the `as.character` function could be used to convert factors to character variables.

Suppose we wish to write a binary version of each row of the `mystate` data frame to a file. When `writeBin` converts a character variable, it uses the C programming language convention of terminating the string with a binary zero. If the program to be reading the data requires fixed-width fields, the `sprintf` function can be used to convert variable-length character values to fixed length. For example, to make all the elements in `mystate$name` the same length we can use the `sprintf` function as follows:

```
> maxl = max(nchar(mystates$name))
> mystates$newname = sprintf(paste('%-',maxl,'s',sep=''),
+                            mystates$name)
```

Omitting the minus sign (`-`) would pad the strings at the beginning, instead of the end.

Since R knows the size and types of its own objects, there is no need to explicitly provide this information to `writeBin`, but if you want `writeBin` to use a nonnative size for any of its output, the `size=` argument is available. Note that using nonnative sizes may make it difficult or impossible to read a binary file on other architectures.

We can now loop over the rows of `mystates`, first writing the character value, then the numeric ones:

```
f = file('states.bin','wb')
for(i in 1:nrow(mystates)){
   writeBin(mystates$newname[i],f)
   writeBin(unlist(mystates[i,2:9]),f)
}
```

Note the use of `unlist` to convert a row of the data frame to a vector suitable for `writeBin`.

2.14 Writing R Objects to Files in ASCII Format

While the binary format that R uses to store data (Section 2.12) is the natural choice for saving data that will be used by R, there are several other ways that the contents of R objects can be written to files. The idea of a human-readable (nonbinary) file with data is very attractive, since most programs can read files of this type, and, in the worst possible case, you can see what the file contains by using an ordinary editor. R provides two functions for writing objects to files in ASCII format; `write`, which is suitable for the same kinds of data as `scan` (Section 2.1), and `write.table`, which is suitable for the types of data which would normally be read using `read.table` (Section 2.2).

2.14.1 The `write` Function

The `write` function accepts an R object and the name of a file or connection object, and writes an ASCII representation of the object to the appropriate destination. The `ncolumns=` argument can be used to specify the number of values to write on each line; it defaults to five for numeric variables, and one for character variables. To build up an output file incrementally, the `append=TRUE` argument can be used.

Note that matrices are internally stored by columns, and will be written to any output connection in that order. To write a matrix in row-wise order, use its transpose and adjust the `ncolumn=` argument appropriately. For example, to write the values in the `state.x77` matrix to file in row-wise order, the following statement could be used:

```
> write(t(state.x77),file='state.txt',ncolumns=ncol(state.x77))
```

2.14.2 The `write.table` function

For mixed-mode data, like data frames, the basic tool to produce ASCII files is `write.table`. The only required argument to `write.table` is the name of a dataset or matrix; with just a single argument, the output will be printed on the console, making it easy to test that the file you'll be creating is in the correct format. Usually, the second argument, `file=` will be used to specify the destination as either a character string to represent a file, or a connection (Section 2.1).

By default, character strings are surrounded by quotes by `write.table`; use the `quote=FALSE` argument to suppress this feature. To suppress row names or column names from being written to the file, use the `row.names=FALSE` or `col.names=FALSE` arguments, respectively. Note that `col.names=TRUE` (the default) produces the same sort of headers that are read using the `header=TRUE` argument of `read.table`. Finally, the `sep=` argument can be used to specify a separator other than a blank space. Using `sep=','` (comma-separated) or `sep='\t'` (tab-separated) are two common choices.

For example, to write the `CO2` data frame as a comma-separated file without row names, but with column headers and quotes surrounding character strings, we could use

```
> write.table(CO2,file='co2.txt',row.names=FALSE,sep=',')
```

Similarly to `read.csv` and `read.csv2`, the functions `write.csv` and `write.csv2` are provided as wrappers to `read.table`, with appropriate options set to produce comma- or semicolon-separated files. In addition, the `write.fwf` function in the `gdata` package, available from CRAN, provides a similar functionality for writing R objects to a file using fixed-width fields.

2.15 Reading Data from Other Programs

It sometimes becomes necessary to access data which was created by a program other than R, or to create data in a form that will be easily accessible by some other program. When collaborating with others, they may have already created a saved object using some other program, or may want a dataset you're working with in a format that their favorite program can understand. You may also encounter situations where some other program is more suitable for a particular task, and, once you've created a saved object with that program, you'll want to bring the results into R. In many cases, the most expedient solution is to rely on human-readable comma-separated files to provide access to data, since almost every program can read such files. If this is not an option, or if there are a large number of datasets that need to be processed or created, it may make sense to try to read data directly into R from the file created by the other program, or to write data from R into a format more suitable to some other program.

Function(s)	Purpose
`data.restore` `read.S`	read `data.dump` output or saved objects from S version 3 may work with older Splus objects
`read.dbf`	read or write saved objects from DBF files (FoxPro, dBase, etc.)
`read.dta` `write.dta`	read saved objects from Stata (versions 5-9) create a Stata saved object
`read.epinfo`	read saved objects from epinfo
`read.spss`	read saved objects from SPSS written using the `save` or `export` command
`read.mtp`	read Minitab Portable Worksheet files
`read.octave`	read saved objects from GNU octave
`read.xport`	read saved objects in SAS export format
`read.systat`	read saved objects from systat rectangular (mtype=1) data only

Table 2.3. Functions in the `foreign` package

The `foreign` package, available from CRAN, provides programs to read and write data in formats supported by a variety of different programs, summarized in Table 2.3. None of the programs in the table require that the foreign program be available on your computer; for example, you can read and write Stata files (using `read.dta` and `write.dta`) even though you may not have a copy of Stata on your computer. The functions that read files all require a filename argument; those that write files require the data frame to be written as the first argument and a destination filename as the second. Some of the functions listed have additional options to control factor conversion and variable name conventions; full details can be found in their respective help files.

For getting data into other programs, the package also provides the `write.foreign` program, which will generate two files: one containing the data in a form that the foreign program can read, and the second containing instructions that will allow the foreign program to read the data. This provides an alternative means of making data available to someone who wishes to use a program other than R. Currently, `write.foreign` supports SPSS, Stata, and SAS. The help page for `write.foreign` explains how to extend it to support other programs.

To use `write.foreign`, provide the name of a data frame, along with a filename where the data will be written (`datafile=`), and a second filename where the foreign program will be written (`codefile=`), along with the `package=` argument indicating the target program. For example, to create data and programs to read an R data frame called `mydata` into Stata, the following call to `write.foreign` could be used:

```
> write.foreign(mydata,'mydata.txt','mydata.stata',
+               package='Stata')
```

If `mydata.stata` is provided as input to Stata, with `mydata.txt` in the current directory, it will load the data from `mydata` into Stata.

In the case of SAS, the `read.ssd` function in the `foreign` package will create an R data frame from any SAS dataset (not just those in export format), by writing and executing a SAS program to write the data in export format and then calling `read.xport`. Thus, to use the program, SAS must be available on your computer. If this is the case, the `Hmisc` package, available from CRAN, also provides a number of programs useful for working with SAS datasets by using SAS to process the data.