Use the `sink` function to redirect *all* output from both `print` and `cat`. Call `sink` with a filename argument to begin redirecting console output to that file. When you are done, use `sink` with no argument to close the file and resume output to the console:

```
> sink("filename")                    # Begin writing output to file

. . . other session work . . .

> sink()                              # Resume writing output to console
```

Discussion

The `print` and `cat` functions normally write their output to your console. The `cat` function writes to a file if you supply a `file` argument, which can be either a filename or a connection. The `print` function cannot redirect its output, but the sink function can force all output to a file. A common use for sink is to capture the output of an R script:

```
> sink("script_output.txt")           # Redirect output to file
> source("script.R")                  # Run the script, capturing its output
> sink()                              # Resume writing output to console
```

If you are repeatedly `cat`ing items to one file, be sure to set `append=TRUE`. Otherwise, each call to `cat` will simply overwrite the file's contents:

```
cat(data, file="analysisReport.out")
cat(results, file="analysisRepart.out", append=TRUE)
cat(conclusion, file="analysisReport.out", append=TRUE)
```

Hard-coding file names like this is a tedious and error-prone process. Did you notice that the filename is misspelled in the second line? Instead of hard-coding the filename repeatedly, I suggest opening a connection to the file and writing your output to the connection:

```
con <- file("analysisReport.out", "w")
cat(data, file=con)
cat(results, file=con)
cat(conclusion, file=con)
close(con)
```

(You don't need `append=TRUE` when writing to a connection because it's implied.) This technique is especially valuable inside R scripts because it makes your code more reliable and more maintainable.

4.4 Listing Files

Problem

You want to see a listing of your files without the hassle of switching to your file browser.

Solution

The `list.files` function shows the contents of your working directory:

```
> list.files()
```

Discussion

This function is just plain handy. If I can't remember the name of my data file (was it *sample_data.csv* or *sample-data.csv*?), I do a quick `list.files` to refresh my memory:

```
> list.files()
[1] "sample-data.csv" "script.R"
```

To see all the files in your subdirectories, too, use `list.files(recursive=T)`.

A possible "gotcha" of `list.files` is that it ignores hidden files—typically, any file whose name begins with a period. If you don't see the file you expected to see, try setting `all.files=TRUE`:

```
> list.files(all.files=TRUE)
```

See Also

R has other handy functions for working with files; see `help(files)`.

4.5 Dealing with "Cannot Open File" in Windows

Problem

You are running R on Windows, and you are using file names such as *C:\data\sample.txt*. R says it cannot open the file, but you know the file does exist.

Solution

The backslashes in the file path are causing trouble. You can solve this problem in one of two ways:

- Change the backslashes to forward slashes: `"C:/data/sample.txt"`.
- Double the backslashes: `"C:\\data\\sample.txt"`.

Discussion

When you open a file in R, you give the file name as a character string. Problems arise when the name contains backslashes (\) because backslashes have a special meaning inside strings. You'll probably get something like this:

```
> samp <- read.csv("C:\Data\sample-data.csv")
Error in file(file, "rt") : cannot open the connection
In addition: Warning messages:
1: '\D' is an unrecognized escape in a character string
```

```
2: '\s' is an unrecognized escape in a character string
3: unrecognized escapes removed from "C:\Data\sample-data.csv"
4: In file(file, "rt") :
  cannot open file 'C:Datasample-data.csv': No such file or directory
```

R escapes every character that follows a backslash and then removes the backslashes. That leaves a meaningless file path, such as *C:Datasample-data.csv* in this example.

The simple solution is to use forward slashes instead of backslashes. R leaves the forward slashes alone, and Windows treats them just like backslashes. Problem solved:

```
> samp <- read.csv("C:/Data/sample-data.csv")
>
```

An alternative solution is to double the backslashes, since R replaces two consecutive backslashes with a single backslash:

```
> samp <- read.csv("C:\\Data\\sample-data.csv")
>
```

4.6 Reading Fixed-Width Records

Problem

You are reading data from a file of fixed-width records: records whose data items occur at fixed boundaries.

Solution

Read the file using the `read.fwf` function. The main arguments are the file name and the widths of the fields:

```
> records <- read.fwf("filename", widths=c(w1, w2, ..., wn))
```

Discussion

Suppose we want to read an entire file of fixed-width records, such as *fixed-width.txt*, shown here:

```
Fisher    R.A.      1890 1962
Pearson   Karl      1857 1936
Cox       Gertrude  1900 1978
Yates     Frank     1902 1994
Smith     Kirstine  1878 1939
```

We need to know the column widths. In this case the columns are: last name, 10 characters; first name, 10 characters; year of birth, 4 characters; and year of death, 4 characters. In between the two last columns is a 1-character space. We can read the file this way:

```
> records <- read.fwf("fixed-width.txt", widths=c(10,10,4,-1,4))
```

The `-1` in the `widths` argument says there is a one-character column that should be ignored. The result of `read.fwf` is a data frame:

```
> records
          V1         V2   V3   V4
1 Fisher     R.A.       1890 1962
2 Pearson    Karl       1857 1936
3 Cox        Gertrude   1900 1978
4 Yates      Frank      1902 1994
5 Smith      Kirstine   1878 1939
```

Note that R supplied some funky, synthetic column names. We can override that default by using a `col.names` argument:

```
> records <- read.fwf("fixed-width.txt", widths=c(10,10,4,-1,4),
+                     col.names=c("Last","First","Born","Died"))
> records
        Last      First Born Died
1 Fisher     R.A.       1890 1962
2 Pearson    Karl       1857 1936
3 Cox        Gertrude   1900 1978
4 Yates      Frank      1902 1994
5 Smith      Kirstine   1878 1939
```

`read.fwf` interprets nonnumeric data as a factor (categorical variable) by default. For instance, the Last and First columns just displayed were interpreted as factors. Set `stringsAsFactors=FALSE` to have the function read them as character strings.

The `read.fwf` function has many bells and whistles that may be useful for reading your data files. It also shares many bells and whistles with the `read.table` function. I suggest reading the help pages for both functions.

See Also

See Recipe 4.7 for more discussion of reading text files.

4.7 Reading Tabular Data Files

Problem

You want to read a text file that contains a table of data.

Solution

Use the `read.table` function, which returns a data frame:

```
> dfrm <- read.table("filename")
```

Discussion

Tabular data files are quite common. They are text files with a simple format:

- Each line contains one record.
- Within each record, fields (items) are separated by a one-character delimiter, such as a space, tab, colon, or comma.
- Each record contains the same number of fields.

This format is more free-form than the fixed-width format because fields needn't be aligned by position. Here is the data file of Recipe 4.6 in tabular format, using a space character between fields:

```
Fisher R.A. 1890 1962
Pearson Karl 1857 1936
Cox Gertrude 1900 1978
Yates Frank 1902 1994
Smith Kirstine 1878 1939
```

The `read.table` function is built to read this file. By default, it assumes the data fields are separated by white space (blanks or tabs):

```
> dfrm <- read.table("statisticians.txt")
> print(dfrm)
       V1       V2   V3   V4
1  Fisher     R.A. 1890 1962
2 Pearson     Karl 1857 1936
3     Cox Gertrude 1900 1978
4   Yates    Frank 1902 1994
5   Smith Kirstine 1878 1939
```

If your file uses a separator other than white space, specify it using the `sep` parameter. If our file used colon (`:`) as the field separator, we would read it this way:

```
> dfrm <- read.table("statisticians.txt", sep=":")
```

You cannot tell from the printed output, but `read.table` interpreted the first and last names as factors, not strings. We see that by checking the class of the resulting column:

```
> class(dfrm$V1)
[1] "factor"
```

To prevent `read.table` from interpreting character strings as factors, set the `stringsAsFactors` parameter to `FALSE`:

```
> dfrm <- read.table("statisticians.txt", stringsAsFactor=FALSE)
> class(dfrm$V1)
[1] "character"
```

Now the class of the first column is character, not factor.

If any field contains the string "NA", then `read.table` assumes that the value is missing and converts it to NA. Your data file might employ a different string to signal missing values, in which case use the `na.strings` parameter. The SAS convention, for example, is that missing values are signaled by a single period (`.`). We can read such data files in this way:

```
> dfrm <- read.table("filename.txt", na.strings=".")
```

I am a huge fan of *self-describing data*: data files which describe their own contents. (A computer scientist would say the file contains its own *metadata*.) The `read.table` function has two features that support this characteristic. First, you can include a *header line* at the top of your file that gives names to the columns. The line contains one name for every column, and it uses the same field separator as the data. Here is our data file with a header line that names the columns:

```
lastname firstname born died
Fisher R.A. 1890 1962
Pearson Karl 1857 1936
Cox Gertrude 1900 1978
Yates Frank 1902 1994
Smith Kirstine 1878 1939
```

Now we can tell `read.table` that our file contains a header line, and it will use the column names when it builds the data frame:

```
> dfrm <- read.table("statisticians.txt", header=TRUE, stringsAsFactor=FALSE)
> print(dfrm)
  lastname firstname born died
1   Fisher      R.A. 1890 1962
2  Pearson      Karl 1857 1936
3      Cox  Gertrude 1900 1978
4    Yates     Frank 1902 1994
5    Smith  Kirstine 1878 1939
```

The second feature of `read.table` is *comment lines*. Any line that begins with a pound sign (`#`) is ignored, so you can put comments on those lines:

```
# This is a data file of famous statisticians.
# Last edited on 1994-06-18
lastname firstname born died
Fisher R.A. 1890 1962
Pearson Karl 1857 1936
Cox Gertrude 1900 1978
Yates Frank 1902 1994
Smith Kirstine 1878 1939
```

`read.table` has many parameters for controlling how it reads and interprets the input file. See the help page for details.

See Also

If your data items are separated by commas, see Recipe 4.8 for reading a CSV file.

4.8 Reading from CSV Files

Problem

You want to read data from a comma-separated values (CSV) file.

Solution

The `read.csv` function can read CSV files. If your CSV file has a header line, use this:

```
> tbl <- read.csv("filename")
```

If your CSV file does not contain a header line, set the `header` option to `FALSE`:

```
> tbl <- read.csv("filename", header=FALSE)
```

Discussion

The CSV file format is popular because many programs can import and export data in that format. Such programs include R, Excel, other spreadsheet programs, many database managers, and most statistical packages. It is a flat file of tabular data, where each line in the file is a row of data, and each row contains data items separated by commas. Here is a very simple CSV file with three rows and three columns (the first line is a *header line* that contains the column names, also separated by commas):

```
label,lbound,ubound
low,0,0.674
mid,0.674,1.64
high,1.64,2.33
```

The `read.csv` file reads the data and creates a data frame, which is the usual R representation for tabular data. The function assumes that your file has a header line unless told otherwise:

```
> tbl <- read.csv("table-data.csv")
> tbl
  label lbound ubound
1   low  0.000  0.674
2   mid  0.674  1.640
3  high  1.640  2.330
```

Observe that `read.csv` took the column names from the header line for the data frame. If the file did not contain a header, then we would specify `header=FALSE` and R would synthesize column names for us (`V1`, `V2`, and `V3` in this case):

```
> tbl <- read.csv("table-data-with-no-header.csv", header=FALSE)
> tbl
    V1    V2    V3
1  low 0.000 0.674
2  mid 0.674 1.640
3 high 1.640 2.330
```

A good feature of `read.csv` is that is automatically interprets nonnumeric data as a factor (categorical variable), which is often what you want since after all this is a statistical package, not Perl. The `label` variable in the `tbl` data frame just shown is actually a factor, not a character variable. You see that by inspecting the structure of `tbl`:

```
> str(tbl)
'data.frame':   3 obs. of  3 variables:
 $ label : Factor w/ 3 levels "high","low","mid": 2 3 1
 $ lbound: num  0 0.674 1.64
 $ ubound: num  0.674 1.64 2.33
```

Sometimes you really want your data interpreted as strings, not as a factor. In that case, set the `as.is` parameter to `TRUE`; this indicates that R should not interpret nonnumeric data as a factor:

```
> tbl <- read.csv("table-data.csv", as.is=TRUE)
> str(tbl)
'data.frame':   3 obs. of  3 variables:
 $ label : chr  "low" "mid" "high"
 $ lbound: num  0 0.674 1.64
 $ ubound: num  0.674 1.64 2.33
```

Notice that the `label` variable now has character-string values and is no longer a factor.

Another useful feature is that input lines starting with a pound sign (`#`) are ignored, which lets you embed comments in your data file. Disable this feature by specifying `comment.char=""`.

The `read.csv` function has many useful bells and whistles. These include the ability to skip leading lines in the input file, control the conversion of individual columns, fill out short rows, limit the number of lines, and control the quoting of strings. See the R help page for details.

See Also

See Recipe 4.9. See the R help page for `read.table`, which is the basis for `read.csv`.

4.9 Writing to CSV Files

Problem

You want to save a matrix or data frame in a file using the comma-separated values format.

Solution

The `write.csv` function can write a CSV file:

```
> write.csv(x, file="filename", row.names=FALSE)
```

Discussion

The `write.csv` function writes tabular data to an ASCII file in CSV format. Each row of data creates one line in the file, with data items separated by commas (`,`):

```
> print(tbl)
  label lbound ubound
```

```
1   low  0.000  0.674
2   mid  0.674  1.640
3  high  1.640  2.330
> write.csv(tbl, file="table-data.csv", row.names=T)
```

This example creates a file called `table-data.csv` in the current working directory. The file looks like this:

```
"label","lbound","ubound"
"low",0,0.674
"mid",0.674,1.64
"high",1.64,2.33
```

Notice that the function writes a column header line by default. Set `col.names=FALSE` to change that.

If we do not specify `row.names=FALSE`, the function prepends each row with a label taken from the `row.names` attribute of your data. If your data doesn't have row names then the function just uses the row numbers, which creates a CSV file like this:

```
"","label","lbound","ubound"
"1","low",0,0.674
"2","mid",0.674,1.64
"3","high",1.64,2.33
```

I rarely want row labels in my CSV files, which is why I recommend setting `row.names=FALSE`.

The function is intentionally inflexible. You cannot easily change the defaults because it really, really wants to write files in a valid CSV format. Use the `write.table` function to save your tabular data in other formats.

A sad limitation of `write.csv` is that it cannot append lines to a file. Use `write.table` instead.

See Also

See Recipe 3.1 for more about the current working directory and Recipe 4.14 for other ways to save data to files.

4.10 Reading Tabular or CSV Data from the Web

Problem

You want to read data directly from the Web into your R workspace.

Solution

Use the `read.csv`, `read.table`, and `scan` functions, but substitute a URL for a file name. The functions will read directly from the remote server:

```
> tbl <- read.csv("http://www.example.com/download/data.csv")
```

You can also open a connection using the URL and then read from the connection, which may be preferable for complicated files.

Discussion

The Web is a gold mine of data. You could download the data into a file and then read the file into R, but it's more convenient to read directly from the Web. Give the URL to `read.csv`, `read.table`, or `scan` (depending upon the format of the data), and the data will be downloaded and parsed for you. No fuss, no muss.

Aside from using a URL, this recipe is just like reading from a CSV file (Recipe 4.8) or a complex file (Recipe 4.12), so all the comments in those recipes apply here, too.

Remember that URLs work for FTP servers, not just HTTP servers. This means that R can also read data from FTP sites using URLs:

```
> tbl <- read.table("ftp://ftp.example.com/download/data.txt")
```

See Also

See Recipes 4.8 and 4.12.

4.11 Reading Data from HTML Tables

Problem

You want to read data from an HTML table on the Web.

Solution

Use the `readHTMLTable` function in the `XML` package. To read all tables on the page, simply give the URL:

```
> library(XML)
> url <- 'http://www.example.com/data/table.html'
> tbls <- readHTMLTable(url)
```

To read only specific tables, use the `which` parameter. This example reads the third table on the page:

```
> tbl <- readHTMLTable(url, which=3)
```

Discussion

Web pages can contain several HTML tables. Calling `readHTMLTable(url)` reads all tables on the page and returns them in a list. This can be useful for exploring a page, but it's annoying if you want just one specific table. In that case, use `which=`*n* to select the desired table. You'll get only the *n*th table.

The following example, which is taken from the help page for `readHTMLTable`, loads all tables from the Wikipedia page entitled "World population":

```
> library(XML)
> url <- 'http://en.wikipedia.org/wiki/World_population'
> tbls <- readHTMLTable(url)
```

As it turns out, that page contains 17 tables:

```
> length(tbls)
[1] 17
```

In this example we care only about the third table (which lists the largest populations by country), so we specify `which=3`:

```
> tbl <- readHTMLTable(url, which=3)
```

In that table, columns 2 and 3 contain the country name and population, respectively:

```
> tbl[,c(2,3)]
                Country / Territory    Population
1  Â People's Republic of China[44] 1,338,460,000
2                           Â India 1,182,800,000
3                   Â United States   309,659,000
4                       Â Indonesia   231,369,500
5                          Â Brazil   193,152,000
6                        Â Pakistan   169,928,500
7                      Â Bangladesh   162,221,000
8                         Â Nigeria   154,729,000
9                          Â Russia   141,927,297
10                          Â Japan   127,530,000
11                         Â Mexico   107,550,697
12                     Â Philippines    92,226,600
13                         Â Vietnam    85,789,573
14                         Â Germany    81,882,342
15                        Â Ethiopia    79,221,000
16                          Â Egypt    78,459,000
```

Right away, we can see problems with the data: the country names have some funky Unicode character stuck to the front. I don't know why; it probably has something to do with formatting the Wikipedia page. Also, the name of the People's Republic of China has "[44]" appended. On the Wikipedia website, that was a footnote reference, but now it's just a bit of unwanted text. Adding insult to injury, the population numbers have embedded commas, so you cannot easily convert them to raw numbers. All these problems can be solved by some string processing, but each problem adds at least one more step to the process.

This illustrates the main obstacle to reading HTML tables. HTML was designed for presenting information to people, not to computers. When you "scrape" information off an HTML page, you get stuff that's useful to people but annoying to computers. If you ever have a choice, choose instead a computer-oriented data representation such as XML, JSON, or CSV.

The `readHTMLTable` function is part of the `XML` package, which (by necessity) is large and complex. The `XML` package depends on a software library called `libxml2`, which you will need to obtain and install first. On Linux, you will also need the Linux package `xml2-config`, which is necessary for building the R package.

See Also

See Recipe 3.9 for downloading and installing packages such as the `XML` package.

4.12 Reading Files with a Complex Structure

Problem

You are reading data from a file that has a complex or irregular structure.

Solution

- Use the `readLines` function to read individual lines; then process them as strings to extract data items.
- Alternatively, use the `scan` function to read individual tokens and use the argument `what` to describe the stream of tokens in your file. The function can convert tokens into data and then assemble the data into records.

Discussion

Life would be simple and beautiful if all our data files were organized into neat tables with cleanly delimited data. We could read those files using `read.table` and get on with living.

Dream on.

You will eventually encounter a funky file format, and your job—no matter how painful—is to read the file contents into R. The `read.table` and `read.csv` functions are line-oriented and probably won't help. However, the `readLines` and `scan` functions are useful here because they let you process the individual lines and even tokens of the file.

The `readLines` function is pretty simple. It reads lines from a file and returns them as a list of character strings:

```
> lines <- readLines("input.txt")
```

You can limit the number of lines by using the `n` parameter, which gives the number of maximum number of lines to be read:

```
> lines <- readLines("input.txt", n=10)       # Read 10 lines and stop
```

The scan function is much richer. It reads one token at a time and handles it according to your instructions. The first argument is either a filename or a connection (more on connections later). The second argument is called what, and it describes the tokens that scan should expect in the input file. The description is cryptic but quite clever:

`what=numeric(0)`
: Interpret the next token as a number.

`what=integer(0)`
: Interpret the next token as an integer.

`what=complex(0)`
: Interpret the next token as complex number.

`what=character(0)`
: Interpret the next token as a character string.

`what=logical(0)`
: Interpret the next token as a logical value.

The scan function will apply the given pattern repeatedly until all data is read.

Suppose your file is simply a sequence of numbers, like this:

```
2355.09 2246.73 1738.74 1841.01 2027.85
```

Use what=numeric(0) to say, "My file is a sequence of tokens, each of which is a number":

```
> singles <- scan("singles.txt", what=numeric(0))
Read 5 items
> singles
[1] 2355.09 2246.73 1738.74 1841.01 2027.85
```

A key feature of scan is that the what can be a list containing several token types. The scan function will assume your file is a repeating sequence of those types. Suppose your file contains triplets of data, like this:

```
15-Oct-87 2439.78 2345.63 16-Oct-87 2396.21 2,207.73
19-Oct-87 2164.16 1677.55 20-Oct-87 2067.47 1,616.21
21-Oct-87 2081.07 1951.76
```

Use a list to tell scan that it should expect a repeating, three-token sequence:

```
> triples <- scan("triples.txt", what=list(character(0),numeric(0),numeric(0)))
```

Give names to the list elements, and scan will assign those names to the data:

```
> triples <- scan("triples.txt",
+                 what=list(date=character(0), high=numeric(0), low=numeric(0)))
Read 5 records
> triples
$date
[1] "15-Oct-87" "16-Oct-87" "19-Oct-87" "20-Oct-87" "21-Oct-87"

$high
[1] 2439.78 2396.21 2164.16 2067.47 2081.07
```

```
$low
[1] 2345.63 2207.73 1677.55 1616.21 1951.76
```

The `scan` function has many bells and whistles, but the following are especially useful:

`n=`*`number`*
: Stop after reading this many tokens. (Default: stop at end of file.)

`nlines=`*`number`*
: Stop after reading this many input lines. (Default: stop at end of file.)

`skip=`*`number`*
: Number of input lines to skip before reading data.

`na.strings=`*`list`*
: A list of strings to be interpreted as NA.

An Example

Let's use this recipe to read a dataset from StatLib, the repository of statistical data and software maintained by Carnegie Mellon University. Jeff Witmer contributed a `dataset` called `wseries` that shows the pattern of wins and losses for every World Series since 1903. The dataset is stored in an ASCII file with 35 lines of comments followed by 23 lines of data. The data itself looks like this:

```
1903  LWLlwwwW    1927  wwWW      1950  wwWW      1973  WLwllWW
1905  wLwWW       1928  WWww      1951  LWlwwW    1974  wlWWW
1906  wLwLwW      1929  wwLWW     1952  lwLWLww   1975  lwWLWlw
1907  WWww        1930  WWllwW    1953  WWllwW    1976  WWww
1908  wWLww       1931  LWwlwLW   1954  WWww      1977  WLwwlW

.
. (etc.)
.
```

The data is encoded as follows: L = loss at home, l = loss on the road, W = win at home, w = win on the road. The data appears in column order, not row order, which complicates our lives a bit.

Here is the R code for reading the raw data:

```
# Read the wseries dataset:
#     - Skip the first 35 lines
#     - Then read 23 lines of data
#     - The data occurs in pairs: a year and a pattern (char string)
#
world.series <- scan("http://lib.stat.cmu.edu/datasets/wseries",
                     skip = 35,
                     nlines = 23,
                     what = list(year = integer(0),
                                 pattern = character(0)),
                     )
```

The `scan` function returns a list, so we get a list with two elements: `year` and `pattern`. The function reads from left to right, but the dataset is organized by columns and so the years appear in a strange order:

```
> world.series$year
 [1] 1903 1927 1950 1973 1905 1928 1951 1974 1906 1929 1952
[12] 1975 1907 1930 1953 1976 1908 1931 1954 1977 1909 1932

.
. (etc.)
.
```

We can fix that by sorting the list elements according to year:

```
> perm <- order(world.series$year)
> world.series <- list(year    = world.series$year[perm],
+                      pattern = world.series$pattern[perm])
```

Now the data appears in chronological order:

```
> world.series$year
 [1] 1903 1905 1906 1907 1908 1909 1910 1911 1912 1913 1914
[12] 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925

.
. (etc.)
.

> world.series$pattern
 [1] "LWLlwwwW" "wLwWW"    "wLwLwW"   "WWww"     "wWLww"
 [6] "WLwlWlw"  "WWwlw"    "lWwWlW"   "wLwWlLW"  "wLwWw"
[11] "wwWW"     "lwWWw"    "WWlwW"    "WWllWw"   "wlwWLW"
[16] "WWlwwLLw" "wllWWWW"  "LlWwLwWw" "WWwW"     "LwLwWw"

.
. (etc.)
.
```

4.13 Reading from MySQL Databases

Problem

You want access to data stored in a MySQL database.

Solution

1. Install the `RMySQL` package on your computer.
2. Open a database connection using the `dbConnect` function.
3. Use `dbGetQuery` to initiate a `SELECT` and return the result sets.
4. Use `dbDisconnect` to terminate the database connection when you are done.

Discussion

This recipe requires that the `RMySQL` package be installed on your computer. That package requires, in turn, the MySQL client software. If the MySQL client software is not already installed and configured, consult the MySQL documentation or your system administrator.

Use the `dbConnect` function to establish a connection to the MySQL database. It returns a connection object which is used in subsequent calls to `RMySQL` functions:

```
library(RMySQL)
con <- dbConnect(MySQL(), user="userid", password="pswd",
                 host="hostname", client.flag=CLIENT_MULTI_RESULTS)
```

Setting `client.flag=CLIENT_MULTI_RESULTS` is necessary to correctly handle multiple result sets. Even if your queries return a single result set, you must set `client.flag` this way because MySQL might include additional status result sets after your data.

The username, password, and host parameters are the same parameters used for accessing MySQL through the *mysql* client program. The example given here shows them hard-coded into the `dbConnect` call. Actually, that is an ill-advised practice. It puts your password out in the open, creating a security problem. It also creates a major headache whenever your password or host change, requiring you to hunt down the hard-coded values. I strongly recommend using the security mechanism of MySQL instead. Put those three parameters into your MySQL configuration file, which is *$HOME/.my.cnf* on Unix and *C:\my.cnf* on Windows. Make sure the file is unreadable by anyone except you. The file is delimited into sections with markers such as [client]. Put the parameters into the [client] section, so that your config file will contain something like this:

```
[client]
user = userid
password = password
host = hostname
```

Once the parameters are defined in the config file, you no longer need to supply them in the `dbConnect` call, which then becomes much simpler:

```
con <- dbConnect(MySQL(), client.flag=CLIENT_MULTI_RESULTS)
```

Use the `dbGetQuery` function to submit your SQL to the database and read the result sets. Doing so requires an open database connection:

```
sql <- "SELECT * from SurveyResults WHERE City = 'Chicago'"
rows <- dbGetQuery(con, sql)
```

You will need to construct your own SQL query, of course; this is just an example. You are not restricted to `SELECT` statements. Any SQL that generates a result set is OK. I generally use `CALL` statements, for example, because all my SQL is encapsulated in stored procedures and those stored procedures contain embedded `SELECT` statements.

Using `dbGetQuery` is convenient because it packages the result set into a data frame and returns the data frame. This is the perfect representation of an SQL result set. The result set is a tabular data structure of rows and columns, and so is a data frame. The result set's columns have names given by the SQL `SELECT` statement, and R uses them for naming the columns of the data frame.

After the first result set of data, MySQL can return a second result set containing status information. You can choose to inspect the status or ignore it, but you must read it. Otherwise, MySQL will complain that there are unprocessed result sets and then halt. So call `dbNextResult` if necessary:

```
if (dbMoreResults(con)) dbNextResult(con)
```

Call `dbGetQuery` repeatedly to perform multiple queries, checking for the result status after each call (and reading it, if necessary). When you are done, close the database connection using `dbDisconnect`:

```
dbDisconnect(con)
```

Here is a complete session that reads and prints three rows from my database of stock prices. The query selects the price of IBM stock for the last three days of 2008. It assumes that the username, password, and host are defined in the *my.cnf* file:

```
> con <- dbConnect(MySQL(), client.flag=CLIENT_MULTI_RESULTS)
> sql <- paste("select * from DailyBar where Symbol = 'IBM'",
+              "and Day between '2008-12-29' and '2008-12-31'")
> rows <- dbGetQuery(con, sql)
> if (dbMoreResults(con)) dbNextResults(con)
> print(rows)
  Symbol        Day       Next OpenPx HighPx LowPx ClosePx AdjClosePx
1    IBM 2008-12-29 2008-12-30  81.72  81.72 79.68   81.25      81.25
2    IBM 2008-12-30 2008-12-31  81.83  83.64 81.52   83.55      83.55
3    IBM 2008-12-31 2009-01-02  83.50  85.00 83.50   84.16      84.16
  HistClosePx  Volume OpenInt
1       81.25 6062600      NA
2       83.55 5774400      NA
3       84.16 6667700      NA
> dbDisconnect(con)
[1] TRUE
```

See Also

See Recipe 3.9 and the documentation for `RMySQL`, which contains more details about configuring and using the package.

R can read from several other RDBMS systems, including Oracle, Sybase, PostgreSQL, and SQLite. For more information, see the *R Data Import/Export* guide, which is supplied with the base distribution (Recipe 1.6) and is also available on CRAN at *http://cran.r-project.org/doc/manuals/R-data.pdf*.

4.14 Saving and Transporting Objects

Problem

You want to store one or more R objects in a file for later use, or you want to copy an R object from one machine to another.

Solution

Write the objects to a file using the `save` function:

```
> save(myData, file="myData.RData")
```

Read them back using the `load` function, either on your computer or on any platform that supports R:

```
> load("myData.RData")
```

The `save` function writes binary data. To save in an ASCII format, use `dput` or `dump` instead:

```
> dput(myData, file="myData.txt")
> dump("myData", file="myData.txt")          # Note quotes around variable name
```

Discussion

I normally save my data in my workspace, but sometimes I need to save data outside my workspace. I may have a large, complicated data object that I want to load into other workspaces, or I may want to move R objects between my Linux box and my Windows box. The `load` and `save` functions let me do all this: `save` will store the object in a file that is portable across machines, and `load` can read those files.

When you run `load`, it does not return your data per se; rather, it creates variables in your workspace, loads your data into those variables, and then returns the names of the variables (in a list). The first time I used `load`, I did this:

```
> myData <- load("myFile.RData")     # Achtung! Might not do what you think
```

I was extremely puzzled because `myData` did not contain my data at all and because my variables had mysteriously appeared in my workspace. Eventually, I broke down and read the documentation for `load`, which explained everything.

The `save` function writes in a binary format to keep the file small. Sometimes you want an ASCII format instead. When you submit a question to a mailing list, for example, including an ASCII dump of the data lets others re-create your problem. In such cases use `dput` or `dump`, which write an ASCII representation.

Be careful when you save and load objects created by a particular R package. When you load the objects, R does not automatically load the required packages, too, so it will not "understand" the object unless you previously loaded the package yourself. For instance, suppose you have an object called `z` created by the `zoo` package, and

suppose we save the object in a file called *z.RData*. The following sequence of functions will create some confusion:

```
> load("z.RData")      # Create and populate the z variable
> plot(z)              # Does not plot what we expected: zoo pkg not loaded
```

We should have loaded the `zoo` package *before* printing or plotting any `zoo` objects, like this:

```
> library(zoo)         # Load the zoo package into memory
> load("z.RData")      # Create and populate the z variable
> plot(z)              # Ahhh. Now plotting works correctly
```