13

Managing Your Files and Workspace

Stata and R both have commands that replicate many of your computer's operating system functions such as listing names of objects, deleting them, setting search paths, and so on. Learning how to use these commands is especially important because, like Stata, R stores its data in your computer's limited random access memory. You need to make the most of your computer's memory when handling large data sets or when a command is highly iterative.

13.1 Loading and Listing Objects

You can see what objects are in your workspace with the `ls` function. To list all objects such as data frames, vectors, and functions, use

```
ls()
```

The `objects` function does the same thing and its name is more descriptive, but `ls` is more widely used since it is the same command that UNIX, Linux, and MacOS X users can use to list the files in a particular directory or folder (without the parentheses).

When you first start R, using the `ls` function will tell you there is nothing in your workspace. How it does this is quite odd by Stata's standards. It tells you that the list of objects in memory is a character vector with zero values.

```
> ls()

character(0)
```

The file *myall.RData* contains all of the objects we created in Chapter 5, "Programming Language Basics." After loading that into our workspace using the `load` function, `ls` will show us the objects that are available.

```
> load("myall.RData")
> ls()

[1] "gender"   "mydata"   "mylist"   "mymatrix" "q1"
[6] "q2"       "q3"       "q4"       "workshop"
```

You can use the pattern argument to search for any regular expression. Therefore, to get a list of all objects that begin with the string *"my,"* you can use the following:

```
> ls(pattern="my")

[1] "mydata"   "mylist"   "mymatrix"
```

The ls function does not look inside data frames to see what they contain, and it does not even tell you when an object is a data frame. You can use many of the functions we have already covered to determine what an object is and what it contains.

To review, typing its name or using the print function will show you the whole object or at least something about it. What print shows you depends on the class of the object. The head and tail functions will show you the top or bottom few lines of vectors, matrices, tables, data frames, or functions.

The class function will tell you if an object is a data frame, list, or some other object. The names function will show you object names within objects such as data frames, lists, vectors, and matrices. The attributes function will display all of the attributes that are stored in an object such as variable names, the object's class, and any labels that it may contain.

```
> attributes(mydata)

$names

[1] "id"   "workshop" "gender"   "q1"   "q2"   "q3"   "q4"

$class

[1] "data.frame"

$row.names

[1] 1 2 3 4 5 6 7 8
```

The str function displays the *str*ucture of any R object in a compact form.

```
> str(mydata)

'data.frame':   8 obs. of  6 variables:
 $ workshop: Factor w/ 4 levels "R","Stata","SPSS",..:
    1 2 1 2 1 2 1 2

 $ gender  : Factor w/ 2 levels "f","m": 1 1 1 NA 2 2 2 2

 $ q1      : num  1 2 2 3 4 5 5 4

 $ q2      : num  1 1 2 1 5 4 3 5

 $ q3      : num  5 4 4 NA 2 5 4 5

 $ q4      : num  1 1 3 3 4 5 4 5
```

The str function works on functions too. The following is the structure it shows for the lm function.

```
> str( lm )

function (formula, data, subset, weights, na.action,

method = "qr", model = TRUE, x = FALSE, y = FALSE,

qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset, ...)
```

The ls.str function applies the str function to every object in your workspace. It is essentially a combination of the ls function and the str function. The following is the structure of all of the objects we had in our workspace as we wrote this paragraph.

```
> ls.str()

myCounts :  'table' int [1:2, 1:2] 2 2 1 2

myCountsDF : 'data.frame':      4 obs. of  3 variables:

 $ gender  : Factor w/ 2 levels "f","m": 1 2 1 2

 $ workshop: Factor w/ 2 levels "R","Stata": 1 1 2 2

 $ Freq    : int  2 2 1 2

mydata : 'data.frame':  8 obs. of  6 variables:

 $ workshop: int  1 2 1 2 1 2 1 2
 $ gender  : Factor w/ 2 levels "f","m": 1 1 1 NA 2 2 2 2

 $ q1      : int  1 2 2 3 4 5 5 4

 $ q2      : int  1 1 2 1 5 4 3 5

 $ q3      : int  5 4 4 NA 2 5 4 5

 $ q4      : int  1 1 3 3 4 5 4 5
```

Frank Harrell's Hmisc package has a contents function that is modeled after the SAS Contents procedure. It also lists names and other attributes as shown below. However, it works only with data frames.

```
> library("Hmisc")

Attaching package: 'Hmisc'
...

> contents(mydata)

Data frame:mydata   8 observations and 7 variables
Maximum # NAs:1

         Levels Storage NAs
id              integer   0
workshop        integer   0
gender        2 integer   1
q1              integer   0
q2              integer   0
q3              integer   1
q4              integer   0

+--------+------+
|Variable|Levels|
+--------+------+
| gender |  f,m |
+--------+------+
```

13.2 Understanding Your Search Path

Once you have data in your workspace, where exactly is it? It is in an *environment* called *.GlobalEnv*. The search function will show us where that resides in R's search path. Since the search path is affected by any packages or data files you load, we will start R with a clean workspace and load our practice data frame, mydata.

```
> setwd("/myRfolder")

> load("mydata.RData")

> ls()

[1] "mydata"
```

Now let us examine R's search path.

```
> search()

[1] ".GlobalEnv"        "package:stats" "package:graphics"

[4] "package:grDevices" "package:utils" "package:datasets"

[7] "package:methods"   "Autoloads"     "package:base"
```

Since our workspace, .GlobalEnv, is in position 1, R will search it first. By supplying no arguments to the `ls` function, we were asking for a listing of objects in the first position of the search path. Let us see what happens if we apply `ls` to different levels. We can either use the path position value, 1, 2, 3,... or their names.

```
> ls(1) #This uses position number.

[1] "mydata"

> ls(".GlobalEnv") #This uses name.

[1] "mydata"
```

The *package:stats* at level 2 contains some of R's built-in statistical functions. There are a lot of them, so let us use the `head` function to show us just the top few results.

```
> head( ls(2) )

[1] "acf"        "acf2AR"     "add.scope"  "add1"
[5] "addmargins" "aggregate"


> head( ls("package:stats") ) #Same result.

[1] "acf"        "acf2AR"     "add.scope"  "add1"
[5] "addmargins" "aggregate"
```

13.3 Attaching Data Frames

Understanding the search path is essential to understanding what the `attach` function really does. We will attach mydata and see what happens.

```
> attach(mydata)

> search()

 [1] ".GlobalEnv"        "mydata"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"

> ls(2)

[1] "gender"   "id"       "q1"       "q2"       "q3"
[6] "q4"       "workshop"
```

You can see that `attach` has made virtual copies of the variables stored in mydata and placed them in search position 2. When we refer to just "gender" rather than "mydata$gender," R looks for it in position 1 first. It does find anything with just that simple name even though mydata$gender is in that position. R then goes on to position 2 and finds it. This is the process that makes it so easy to refer to variables by their simple component names. It also makes them very confusing to work with if you create new variables! Let us say we want to take the square root of q4:

```
> q4 <- sqrt(q4)

> q4

[1] 1.000000 1.000000 1.732051 1.732051 2.000000 2.236068
[7] 2.000000 2.236068
```

This looks like it worked fine. However, let us list the contents of search positions 1 and 2 to see what really happened:

```
> ls(1)

[1] "mydata" "q4"

> ls(2)

[1] "gender"   "id"       "q1"       "q2"       "q3"
[6] "q4"       "workshop"
```

R created the new version of q4 as a separate vector in our main workspace. The copy of q4 that the `attach` function put in position 2 was never changed! Since search position 1 dominates, asking for q4 will cause R to show us the one in our workspace. Asking for mydata$q4 will cause R to go inside the data frame and show us the original, untransformed values.

There are two important lessons to learn from this:

1. If you want to create a new variable inside a data frame, either fully specify the name using `mydata$varname` or `mydata[ ,"varname"]`, or use the `tranform` function described in Section 10.1, *Transforming Variables.*
2. When two objects have the same name, R will always choose the object higher in the search path.

When the `attach` function places objects in position 2 of the search path (a position you can change but rarely need to), those objects will block, or *mask*, any others of the same name in lower positions (i.e., further toward the end of the search list). In the following example, I started with a fresh launch of R, loaded mydata, and attached it twice to see what happens.

```
> attach(mydata)
> attach(mydata)

The following object(s) are masked from mydata (position 3):
       gender id q1 q2 q3 q4 workshop

> search()

 [1] ".GlobalEnv"    "mydata"            "mydata"
 [4] "package:stats" "package:graphics"  "package:grDevices"
 [7] "package:utils" "package:datasets"  "package:methods"
[10] "Autoloads"     "package:base"
```

Note that above mydata is now in search positions 2 and 3. If you refer to any variable or any object, R has to settle which one you mean (they do not need to be identical, as in this example.) The message about masked objects in position 3 tells us that the second attach brought in variables with those names into position 2. The variables from the first attach were then moved to position 3, and those with common names were masked (all of them in this example). Therefore, we can no longer refer to them by their simple names; those names are already in use somewhere higher in the search path. In this case, the variables from the second attach went to position 2 and they will be used. However, if objects with any of those names were in our main workspace (not in a data frame), they would be used instead.

When we first learned about vectors, we created q1, q2, q3, and q4 as vectors and then formed them into a data frame. If we had left them as separate vectors in our main workspace, even the first attach would have given us a similar message. The vectors in position 1 would have blocked those with the same names in positions 2 and 3.

This masking effect can block access to any object. In Section 2.2, "Loading an Add-on Package," we saw that when two packages share a function name, the most recent one loaded from the library is the one R will use. You can avoid getting the warning about masked objects by first detaching packages or data frames you previously attached, before attaching the second.

13.4 Attaching Files

So far, we have only used the `attach` function with data frames. It can also be very useful with R data files. If you load a file, it brings all objects into your workspace. However, if you attach the file, you can bring in only what you need and then detach it.

For example, let us create a variable x and then add only the vector q4 from the file myall.RData, a file that contains the objects we created in Chapter 5, "Programming Language Basics." Recall that in that chapter, we created each of our practice variables first as vectors and then converted them to factors, a matrix, a data frame, and a list.

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

> attach("myall.RData")

> search()

 [1] ".GlobalEnv"        "file:myall.RData"  "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"

> q4 <- q4
```

The last statement looks quite odd! What is going on? The `attach` function loaded myall.RData, but put it at position 2 in the search path. R will place any variables you create in your workspace (position 1) and the attached copy allows R to find q4 in position 2. So it copies it from there to your workspace. Let us look at what we now have in both places.

```
> ls(1) # Your workspace.

[1] "q4" "x"

> ls(2) # The attached file.
```

```
[1] "gender"   "mydata"   "mylist"   "mymatrix" "q1"
[6] "q2"       "q3"       "q4"       "workshop"
```

```
> detach(2)
```

So we have succeeded in copying a single vector, q4, from a data frame into our workspace. The final `detach` removes `"file:myall.RData"` from the search path.

13.5 Removing Objects from Your Workspace

To delete an object from your workspace, use the `remove` function or the equivalent `rm` function as in

```
rm(mydata)
```

The `rm` function is one of the few functions that will accept multiple objects separated by commas; that is, the names do not have to be in a single character vector. In fact, the names *cannot* simply be placed into a single vector. We will soon see why.

Let us load myall.RData, so we will have lots of objects to remove.

```
> load(file="myall.RData")

> ls()

 [1] "mystats" "gender"   "mydata"   "mylist"   "mymatrix"
 [6] "q1"       "q2"       "q3"       "q4"       "workshop"
```

We do not need our vectors, workshop, gender, and the q variable since they are in our dataframe, mydata. To remove these extraneous variables, we can use

```
rm(workshop,gender,q1,q2,q3,q4)
```

If we had lots of variables, manually entering each name would get quite tedious. We can instead use any of the shortcuts for creating sets of variable names described in Chapter 7, "Selecting Variables." Let us use the `ls` function with its `pattern` argument to find all of the objects that begin with the letter "q."

```
> myQvars <- ls(pattern="q")

> myQvars

[1] "q1" "q2" "q3" "q4"
```

Now let us use the `c` function to combine workshop and gender with myQvars:

```
> myDeleteItems <- c("workshop","gender",myQvars)

> myDeleteItems

[1] "workshop" "gender"   "q1"       "q2"       "q3"
[6] "q4"
```

Note that myQvars is *not* enclosed in quotes in the first line. It is already a character vector that we are adding to the character values of "workshop" and "gender."

Finally, we can delete them all at once by adding the `list` argument to the `rm` function:

```
> rm(list=myDeleteItems)
>
> ls()
[1] "mydata"   "myDeleteItems" "mylist"   "mymatrix"
[5] "myQvars"  "mystats"
```

Finally, we can remove myQvars and myDeleteItems.

```
> rm(myQvars,myDeleteItems)

> ls()

[1] "mydata"   "mylist"   "mymatrix" "mystats"
```

It may appear that a good way to delete the list of objects in myDeleteItems would be to use

```
rm(myDeleteItems)
```

or, equivalently,

```
rm( c("workshop","gender","q1","q2","q3","q4") )
```

However, that would delete only the *list of item names*, not the items themselves! That is why the `rm` function needs a `list` argument when dealing with character vectors.

Once you are happy with the objects remaining in your workspace, you can save them all with

```
save.image("myFavorites.RData")
```

If you want to delete all of the visible objects in your workspace, you can do the following. Be careful, there is no "undo" function for this radical step!

```
myDeleteItems <- ls()

rm( list=myDeleteItems )
```

Doing this in two steps makes it clear what is happening, but, of course, you can nest these two functions. This approach looks quite cryptic at first, but I hope the above steps make it much more obvious what is occurring.

```
rm( list=ls() )
```

To conserve workspace by saving only the variables you need within a data frame, see Section 10.8, "Keeping and Dropping Variables." The `rm` function cannot drop variables stored within a data frame.

13.6 Minimizing Your Workspace

Removing unneeded objects from your workspace is one important way to save space. You can also use the `cleanup.import` function from Frank Harrell's `Hmisc` package. It automatically stores the variables in a data frame in their most compact form. You use it as

```
library("Hmisc")

mydata <- cleanup.import(mydata)
```

If you have not installed `Hmisc`, see Chapter 2, "Installing R and Add-on Packages," for details.

13.7 Setting Your Working Directory

Your working directory is the location R uses to retrieve or store files, if you do not otherwise specify the full path for filenames. On Windows, the default working directory is *My Documents*. On Windows XP or earlier, that is C:\Documents and Settings\username\My Documents. On Windows Vista or later, that is C:\Users\Yourname\My Documents. On Macintosh, the default working directory is /Users/username.

The `getwd` function will tell you the current location of your working directory:

```
> getwd()

[1] "C:/Users/Muenchen/My Documents"
```

Windows users can see and/or change their working directory by choosing *File>Change dir...*. R will then display a window that you use to browse to any folder you like.

On any operating system, you can change the working directory with the `setwd` function. This is the equivalent to the Stata `cd` command. Simply provide the full path between the quotes:

```
setwd("/myRfolder")
```

We discussed earlier that R uses the forward slash "/" even on computers running Windows. That is because within strings, R uses "`\t`," "`\n`" and "`\\`" to represent the single characters tab, newline, and backslash, respectively. In general, a backslash followed by another character may have a special meaning. So when using R on Windows, always specify the paths with either a single forward slash or two backslashes in a row. This book uses the single forward slash because that works with R on all operating systems.

You can set your working directory automatically by putting it in your .Rprofile. For details, see Appendix C, "Automating Your R Setup."

13.8 Saving Your Workspace

Throughout this book we manually save the objects we create, naming them as we do so. That is the way almost all other computer programs work. R also has options for saving your workspace automatically when you exit.

13.8.1 Saving Your Workspace Manually

To save the entire contents of your workspace, you can use the `save.image` function:

```
save.image(file="myWorkspace.RData")
```

This will save all your objects, data, functions, everything. Therefore, it is usually good to remove unwanted objects first, using the `rm` function. See Section 13.5, "Removing Objects from Your Workspace," for details.
If you are a Windows user, R does not automatically append the .RData extension, as do most Windows programs, so make sure you enter it yourself.

Later, when you start R, you can use *File>Load Workspace* to load it from the hard drive back into the computer's memory. You can also restore them using the `load` function.

```
load(file="myWorkspace.RData")
```

If you want to save only a subset of your workspace, the `save` function allows you to list the objects to save, separated by commas, before the file argument:

```
save(mydata, file="mydata.RData")
```

This is one of the few functions that can accept many objects separated by commas, so might save three as in the example below.

```
save(mydata, mylist, mymatrix, file="myExamples.RData")
```

It also has a `list` argument that lets you specify a character vector of objects to save.

You exit R by choosing *File>Exit* or by entering the function call `quit()` or just `q()`. R will then offer to save your workspace. If you have used either the `save` or the `save.image` functions recommended above, you should say "No."

13.8.2 Saving Your Workspace Automatically

Every time you exit R, it offers to save your workspace for you automatically. If you click "Yes," it stores it in a file named ".RData" in your working directory (see how to set that in the Section 13.7. The next time you start R from the same working directory, it automatically loads that file back into memory, and you can continue working.

While this method saves a little time, it also has problems. The name .RData is an odd choice, because most operating systems hide files that begin with a period. So, initially, you cannot copy or rename your project files! That is true on Windows, Macintosh, and Linux/UNIX systems. Of course, you can tell your operating system to show you such files (shown below).

Since all your projects end up in a file with the same name, it is harder to find the one you need via search engines or backup systems. If you accidentally moved an .RData file to another folder, you would not know which project it contained without first loading it into R.

13.9 Getting Operating Systems to Show You "*.RData*" Files

While the default workspace file, "*.RData*," is hidden on most operating system, you can tell them to show you those files.

To get Windows XP to show you .RData, in Windows Explorer uncheck the option below and *uncheck* the option *Tools> Folder Options> View> Hide extensions to known file types.* Then click *Apply to all folders.* Then click *OK.*

In Windows Vista, use the following selection: *Start>Control Panel> Appearance and Personalization> Folder Options> View> Show hidden files and folders.* Then click *OK.*

In Windows 7 or later, start File Explorer, then follow this menu path, and *uncheck* the option *Organize> Folder and search options> View> Hide extensions for known file types.* Then click *OK.*

Note that this will still not allow you to click on a filename like myProject.RData and rename it to just .RData. The Windows Rename message box will tell you "You must type a filename."

Linux/UNIX users can see files named .RData with the command "ls -a."

Macintosh users can see files named .RData by starting a terminal window with *Applications> Utilities> Terminal.* In the terminal window, enter

```
defaults write com.apple.finder AppleShowAllFiles TRUE
killall Finder
```

To revert back to normal file view, simply type the same thing, but with "FALSE" instead of "TRUE."

13.10 Organizing Projects with Windows Shortcuts

If you are a Windows user and like using shortcuts, there is another way to keep your various projects organized. You can create an R shortcut for each of your analysis projects. Then you right-click the shortcut, choose *Properties*, and set the *Start in folder* to a unique folder. When you use that shortcut to start R, on exit it will store the .RData file for that project. Although neatly organized into separate folders, each project workspace will still be in a file named .RData.

13.11 Saving Your Programs and Output

R users who prefer the graphical user interface can easily save programs, called scripts, and output to files in the usual way. Just click anywhere on the window you wish to save, choose *File>Save as*, and supply a name. The standard extension for R programs is ".R" and for output is simply ".txt". You can also save bits of output to your word processor using the typical cut/paste steps.

On Windows, R will not automatically append ".R" to each filename. You must specify that yourself. When you forget this, and you will, later choosing *File>Open script* will not let you see the file! You will have to specify "*.*" as the filename to get the file to appear.

R users who prefer to use the command-line interface often use text editors such as Emacs, or the one in JGR, that will check their R syntax for errors. Those files are no different from any other file created in a given editor.

Windows and Macintosh users can cut and paste graphics output into their word processors or other applications. Users of any operating system can rerun graphs, directing their output to a file. See Chapter 14, "Graphics Overview" for details.

13.12 Saving Your History

The R console displays function calls you enter (or that menus enter for you) and their output. It is a good idea to submit function calls from a script window (program editor), but sometimes you enter them directly into the console and then later realize that you need to save the program. You could save the input and output in the console window, but you would need to edit out the output to create a usable program.

R has a *history* file that saves all of the function calls in a given session. This is similar to the Stata LOG file. However, unlike Stata, the history file is not cumulative on Windows computers. It is cumulative on Linux and Macintosh, however.

You can save the current session's history to a file in your working directory with the `savehistory` function. To route the history to a different folder, use the `setwd` function to change it before using `savehistory`, or simply specify the file's full path in the `file=` argument.

```
savehistory(file="myHistory.Rhistory")
```

You can later recall it using the `loadhistory` function.

```
loadhistory(file="myHistory.Rhistory")
```

Note that the filename can be anything you like, but the extension should be ".Rhistory." In fact the entire filename will be simply ".Rhistory" if you do not provide one. We prefer to save a cumulative history file automatically. It takes little disk space and you never know when it will help you recover work that you thought you would never need. For details, see Appendix C, "Automating Your R Setup."

All of the file and workspace functions we have discussed are summarized in Table 13.1.

13.13 Large Data Set Considerations

All of the topics we have covered in this chapter are helpful for managing the amount of free space you have available in your workspace. Since R, like Stata, stores its data in your computer's random access memory, R cannot analyze huge data sets.

An exception to this is Thomas Lumley's `biglm` package [31], which processes data in "chunks" for some linear and generalized linear models.

When the physical memory in your computer fills up, it is possible to use use your computer's hard drive to simulate more memory (known as virtual memory), but it is extremely inefficient and time-consuming. Given the low cost of memory today this is much less of a problem than you might think. R can handle hundreds of thousands of records on a computer with 2 gigabytes of

Table 13.1. Workspace management functions

Function to perform	Example
List object names, including .First, .Last	`objects(all.names=TRUE)`
List object names, of most objects	`ls()` or `objects()`
List object attributes	`attributes(mydata)`
Load workspace	`load(file="myWorkspace.RData")`
Remove a variable from a data frame	`mydata$myvar <- NULL`
Remove all objects (non hidden ones)	`rm( list=ls() )`
Remove an object	`rm(mydata)`
Remove several objects	`rm(mydata, mymatrix, mylist)`
Save all objects	`save.image(file="myWorkspace.RData")`
Save some objects	`save(x,y,z,file="myObjects.RData)`
Show structure of all objects	`ls.str( all.names=TRUE )`
Show structure of most objects	`ls.str()`
Show structure of data frame only (requires Hmisc)	`contents(mydata)`
Show structure of objects by name	`str(mydata), str(lm)`
Store data efficiently (requires Hmisc)	`mydata <- cleanup.import(mydata)`
Working directory, getting	`getwd ()`
Working directory, setting	`setwd("/mypath/myfolder")` Even Windows uses forward slashes,

memory available to R. That is the current memory limit for a single process or program in 32-bit operating systems. To have 2 gigabytes free just for R, you would you would want to have perhaps 3 gigabytes of total memory so that your operating system would have the room it needs.

Operating systems capable of 64-bit memory spaces are the norm on newer systems. The huge amounts of memory they can handle mitigate this problem.

Another way around the limitation is to store your data in a relational database and use its facilities to generate a sample to analyze. A sample size of a few thousand is sufficient for many analyses.

However, if you need to ensure that certain small groups (e.g., those who have a rare disease, the small proportion that defaulted on a loan) then you may end up taking a complex sample, which complicates your analysis considerably. R has specialized packages to help analyze such samples, including `pps`, `sampfling`, `sampling`, `spsurvey`, and `survey`. See CRAN at `http://cran.r-project.org/` for details.

An alternative for R users is to purchase S-PLUS, a commercial package that has an almost identical language to R. S-PLUS has functions to handle what it calls "big data." It solves the problem in a way similar to that used in packages such as SAS and SPSS. However, while S-PLUS can run many R programs written using R's core functions, it cannot run R's add-on packages.

13.14 Example R Program for Managing Files and Workspace

Most chapters in this book end with the Stata and R programs that summarize the topics in the chapter. However this chapter has been very specific to R. Therefore, we present only the R program below.

```
# Filename: ManagingFilesWorkspace.R

ls()

setwd("/myRfolder")
load("myall.RData")
ls()

# List objects that begin with "my".
ls(pattern="my")

# Get attributes and structure of mydata.
attributes(mydata)
str(mydata)

# Get structure of the lm function.
str( lm )

# List all objects' structure.
ls.str()

# Use the Hmisc contents function.
install.packages("Hmisc")
library("Hmisc")
contents(mydata)

# ---Understanding Search Paths---
# After restarting R to purge it of
# packages added by loading Hmisc...

setwd("/myRfolder")
load("mydata.RData")
ls()
search()
```

```
ls(1) #This uses position number.
ls(".GlobalEnv") # This does the same using name.

head( ls(2) )
head( ls("package:stats") ) #Same result.

# See how attaching mydata change the path.
attach(mydata)
search()
ls(2)

# Create a new variable.
q4 <- sqrt(q4)
q4
ls(1)
ls(2)

# Attaching data frames.
detach(mydata)
attach(mydata)
attach(mydata)
search()

# Clean up for next example,
# or restart R with an empty workspace.
detach(mydata)
detach(mydata)
rm( list=ls() )

# Attaching files.
x <- c(1,2,3,4,5,6,7,8)
attach("myall.RData")
search()
q4 <- q4

ls(1) # Your workspace.
ls(2) # The attached file.
detach(2)

# Removing objects.
rm(mydata)
load(file="myall.RData")
ls()
# Example not run:
# rm(workshop,gender,q1,q2,q3,q4)
```

```
myQvars <- ls(pattern="q")
myQvars

myDeleteItems <- c("workshop","gender",myQvars)
myDeleteItems

myDeleteItems
rm( list=myDeleteItems )

ls()
rm( myQvars, myDeleteItems )
ls()

# Wrong!
rm(myDeleteItems)
rm( c("workshop","gender","q1","q2","q3","q4") )

save.image("myFavorites.RData")

# Removing all workspace items.
# The clear approach:

myDeleteItems <- ls()
myDeleteItems
rm( list=myDeleteItems )

# The usual approach:
rm( list=ls() )

# Setting your working directory.

getwd()
setwd("/myRfolder")

# Saving your workspace.

load(file="myall.RData")

# Save everything.
save.image(file="myPractice1.RData")

# Save some objects.
save(mydata,file="myPractice2.RData")
save(mydata,mylist,mymatrix,file="myPractice3.RData")
```

```
# Remove all objects and reload myPractice3.
rm( list=ls() )
load("myPractice3.RData")
ls()

# Save and load history.
savehistory(file="myHistory.Rhistory")
loadhistory(file="myHistory.Rhistory")
```