R FOR SAS AND SPSS USERS

Robert A. Muenchen

This 80-page document is an early version of the 686-page book by the same name. You can download the programs and data sets used in both documents at: http://r4stats.com

I thank the many R developers for providing such wonderful tools for free and all the r-help participants who have kindly answered so many questions. I'm especially grateful to the people who provided advice, caught typos and suggested improvements including: Patrick Burns, Peter Flom, Martin Gregory, Charilaos Skiadas and Michael Wexler.

INTRODUCTION

The goal of this document is to provide an introduction to R that that is tailored to people who already know either SAS or SPSS. For each of 27 fundamental topics, we will compare programs written in SAS, SPSS and the R language.

Since its release in 1996, R has dramatically changed the landscape of research software. There are very few things that SAS or SPSS will do that R cannot, while R can do a wide range of things that the others cannot. Given that R is free and the others quite expensive, R is definitely worth investigating.

It takes most statistics packages at least five years to add a major new analytic method. Statisticians who develop new methods often work in R, so R users often get to use them immediately. There are now over 800 add-on packages available for R.

R also has full matrix capabilities that are quite similar to MATLAB, and it even offers a MATLAB emulation package. For a comparison of R and MATLAB, see http://wiki.r-project.org/rwiki/doku.php?id=getting-started:translations:octave2r.

SAS and SPSS are so similar to each other that moving from one to the other is fairly straightforward. R however is totally different, making the transition confusing at first. I hope to ease that confusion by focusing on the similarities and differences in this document. It may then be easier to follow a more comprehensive introduction to R.

I introduce topics in a carefully chosen order so it is best to read this from beginning to end the first time through, even if you think you don't need to know a particular topic. Later you can skip directly to the section you need.

THE FIVE MAIN PARTS OF SAS AND SPSS

While SAS and SPSS offer many hundreds of functions and procedures, these fall into five main categories:

1. Data input and management statements that help you read, transform and organize your data.
2. Statistical and graphical procedures to help you analyze data.
3. An output management system to help you extract output from statistical procedures for processing in other procedures, or to let you customize printed output. SAS calls this the Output Delivery System (ODS), SPSS calls it the Output Management System (OMS).
4. A macro language to help you use sets of the above commands repeatedly.
5. A matrix language to add new algorithms (SAS/IML and SPSS Matrix).

SAS and SPSS handle each with different systems that follow different rules. For simplicity's sake, introductory training in SAS or SPSS typically focus on topics 1 and 2. Perhaps the majority of users never learn the more advanced topics. However, R performs these five functions in a way that completely integrates them all. So while we'll focus on topics 1 and 2 with when discussing SAS and SPSS, we'll discuss some of all five regarding R. Other introductory guides in R cover these topics in a much more balanced manner. When you finish with this document, you will want to read one of these; see the section ***Help and Documentation*** for recommendations.

The integration of these five areas gives R a significant advantage in power. This advantage is demonstrated by the fact that most R procedures are written using the R language. SAS and SPSS procedures are not written using their languages. R's procedures are also available for you to see and modify in any way you like.

While only a small percent of SAS and SPSS users take advantage of their output management systems, virtually all R users do. That is because R's is dramatically easier to use. For example, you can create and store a regression model with `myModel<-lm(y~x)`. You can get several diagnostic plots with `plot(myModel)` or compare two models with `anova(myModel1,mymodel2).` That is a very flexible approach!

The price R pays for this output management advantage is that the output to most procedures is sparse and does not appear in R as publication quality. Variable labels are not a part of the core system. They have been added on in the Hmisc package – a testament to R's amazing flexibility – but they are not used by most procedures. You can use functions from add-on packages to write out HTML or TEX files that you can then import into word processing tools. SPSS, and more recently SAS, make output that is publication quality by default, but harder to use as input to further analysis.

On the topic of matrix languages, SAS and SPSS offer them in a form that differs sharply from their main languages. For example, the way you select variables in the main SAS product bears no relation to how you select them in SAS/IML. In R, the matrix capabilities are completely integrated.

TYPOGRAPHIC & PROGRAMMING CONVENTIONS

In the examples below, I assume you know either SAS or SPSS and so include little explanation in those programs. The R programs are embellished with comments and variations.

Although R has many ways to generate practice data and has a variety of example data sets, all the examples below are based upon a text file that the first examples read and then store. Once you have run the import step for any or all packages, then the examples will work because they

read the data saved at that step. The examples use file paths appropriate for Microsoft Windows, but should be readily adaptable to any other system.

All programming code and R function names are written in: `this courier font.`

Names of other documents and menus are written in: ***this italic font.***

When learning a new language it can be hard to tell the commands from the names. To help differentiate, I CAPITALIZE commands in SAS and SPSS and use lower case for names. However R is case sensitive so I have to use the exact case that the program requires. So to help differentiate, I use the common prefix "my" in names like mydata or mysubset. While I prefer to use R names like my.subset, the period has special meaning in SAS and so I avoid it in the examples.

HELP AND DOCUMENTATION

The command `help.start()` or choosing ***HTML Help*** from the ***Help*** menu will yield a table of contents that points to help files, manuals, frequently asked questions and the like. To get help for a certain function such as `summary`, use `help(summary)` or prefix the topic with a question mark: `?summary`. To get help on an operator, enclose it in quotes as in `help("<-")` for the assignment operator. If you don't know the name of a command or operator, use `help.search("your search string")` to search the built-in help files.

The help files can be somewhat intimidating at first, since many of them assume you already know a lot about R. An easier way to learn more about R is by reading ***R for Beginners*** by Paradis or some of the other wonderful books available for free at http://cran.r-project.org/ under documentation. Another good choice is An Introduction to S and the Hmisc and Design Libraries by Alzola and Harrell at http://biostat.mc.vanderbilt.edu/twiki/pub/Main/RS/sintro.pdf. The most widely recommended statistics book on R is ***Modern Applied Statistics in S***, by Venables and Ripley. Note that R is almost identical to the S language and these books on S usually point out what the few differences are.

I highly recommend signing up for the R-help listserv at http://www.r-project.org/ under mailing lists. There you can learn a lot by reading answers to the myriad of questions people post. Also, if you post your own questions on the list, you're likely to get an answer in an hour or two. But please read the posting guide, http://www.R-project.org/posting-guide.html, before sending your first question.

Searching the web for information on R can be frustrating, since the string "R" rarely refers to the R Language. However, you can search just the R site using `RSiteSearch("your search string"), or go to` http://www.r-project.org/ and click ***search***. If you use the

Firefox web browser, there is a plug-in called Rsitesearch available at http://addictedtor.free.fr/rsitesearch/.

Graphical User Interfaces

The main R installation does not include a point-and-click graphical user interface (GUI) for running analyses, but you can learn about several at the main R web site, http://www.r-project.org/ under ***Related Projects*** and then ***R GUIs***. My favorite one is R commander, which looks similar to the SPSS GUI. It provides menus for many analytic and graphical methods and shows you the R commands that it enters, making it easy to learn the commands as you use it. You can learn more about R Commander from http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/ .

If you do data mining, you may be interested in the ***RATTLE*** user interface from http://rattle.togaware.com/. It is a point and click interface that writes and executes R programs for you.

Easing Into R

As any student of human behavior can tell you, few things guarantee success like immediate reinforcement. So a great way to ease your way into R is to continue to use SAS, SPSS or your favorite spreadsheet program to enter and manage your data, then use the commands below to import it and go directly to graphs and analyses. As you find errors in your data (and you know you will) you can go back to your other software, correct them and then import it again. It's not an ideal way to work but it does get you into R quickly.

A Few R Basics

Before reading any of the example programs below, you'll need to know a few things about R. What will become immediately apparent is how ***completely*** different R is. What will not be obvious is why these differences give it such an advantage.

SAS and SPSS both use one main data structure, the data set. Instead, R has many different data structures. The one that is most like a data set is called a ***data frame***. SAS and SPSS data sets are always viewed as a rectangle with ***variables*** in the columns and ***records*** in the rows. SAS calls these records ***observations*** and SPSS calls them ***cases***. R documentation uses variables and columns interchangeably. It usually refers to observations or cases as rows.

R data frames have a formal place for an ID variable it calls ***row labels***. SAS and SPSS users typically have an ID variable containing an observation/case number or perhaps a subject's name. But this variable is like any other unless you run a procedure that identifies observations. You can use R this way too, but procedures that identify observations may do so automatically if you set your ID variable to be official row labels. Also when you do that, the variable's original name (id, subject, ssn…) vanishes. The information is used automatically when it is needed.

Another data structure R uses frequently is the vector. A vector is a single-dimensional collection of numbers (numeric vector) or character values (character vector) like variable names.

Variable names in R can be any length consisting of letters, numbers or the period "." and should begin with a letter. Note that underscores are not allowed so my_data is not a valid name but my.data is. However, if you always put quotes around a variable (object) name, it can be any non-empty string. Unlike SAS, the period has no meaning in the name of a dataset. However given that my readers will often be SAS users, I avoid the use of the period. Case matters so you can have two variables, one named myvar and another named MyVar in the same data frame, although that is not a good idea! Some add-on packages, tweak names like the capitalized "`Save`" to represent a compatible, but enhanced, version of a built-in function like the lower-cased "`save`".

R has several operators that are different from SAS or SPSS. The assignment operator is not the equal sign you're used to, but is the two symbols, "`<-`" typed right next to each other. You can use it to put data into objects as in:
`mynames<-c("workshop","gender","q1","q2","q3","q4")`. The combine function, `c`, puts the names together into a single object called a character vector. R also uses the double equal signs "`==`" as a logical comparison operator as in, `gender=="f".` The # operator is used to begin a comment, which then continues until the end of the line. Semicolons can be used to enter multiple commands on a single line as in SAS, but they are not usually used to end a command.

Commands can begin and end anywhere on a line and additional spaces are ignored. You can continue a command on a new line so long as the fragment you leave behind is not already a complete command itself. Going to a new line after a comma is usually a safe bet, and as you will see, R commands are filled with commas making that convenient.

Let's look at one of the simplest R functions, `mean`. Saying `help(mean)` will tells us that the form of the function is `mean(x,trim=0,na.rm=FALSE)` where x is data frame, numeric vector or a date. We'll discuss this more in the section on selecting variables. Trim is the "argument" that tells R what percent of the extreme values to exclude before calculating the mean. The zero indicates the default value, i.e. none of your data will be trimmed unless you change that.

The `na.rm` argument appears in many R functions. R uses NA to represent Not Available, or missing values. The default value of "false" tells us that R will not remove them unless you tell it otherwise. So the result will be NA if any missing values are present! Many functions in R are set this way by default. That is quite the opposite from SAS and SPSS, which almost always assume you want to use all the data you have unless you tell it otherwise.

We can run this by naming each argument:
`mean(x=mydata, trim=.25,na.rm=TRUE)`. It will warn us that the second variable, gender, is not numeric but go ahead and compute the result. If we list every argument in order, we need not name them all. However, most people skip naming the first argument and then name the others and include them only if they wish to change their default values. For example, `mean(mydata,na.rm=TRUE)`.

Unlike SAS or SPSS the output in R does not appear nicely formatted and ready to publish. However you can use the functions in the prettyR and Hmisc packages to make the results of tabular output more ready for publication.

To run the examples below, download R from one of the "mirrors" at http://cran.r-project.org/ and install it. Start it and enter (or cut & paste) the examples into the console window at the > prompt. Or you can use File> New Script to enter the examples into and select some text and right-click it to submit or run the statements. If you are reading the PDF version of this document, you may not be able cut and paste (depends upon your PDF tools). An HTML version that makes cut/paste easy is also available at http://oit.utk.edu/scc/RforSASandSPSSusers.html.

INSTALLING ADD-ON PACKAGES

This is a very important topic in R. In SAS and SPSS installations, you usually have everything you have paid for installed at once. R is much more modular. The main installation will install R and a popular set of add-ons called libraries. Hundreds of other libraries are available to install separately from the Comprehensive R Archive Network, (CRAN). For a list of them with descriptions, see http://cran.r-project.org/ under Packages, but don't download them there.

R automates the download and installation process. Once you have chosen a package, choose Install Packages from the Packages menu. It will ask you which CRAN mirror site you want to use. Choose the nearest one. It will then show you the many packages available. Choose the one you want and it will download and install it for you.

Once it is installed, it is on the computer's hard drive. To use it, you must load it by choosing Load Package from the Packages menu. It will show you the names of all packages that are installed but not yet loaded. You can also load a package with the command library(packagename).

If the package contains example data sets, you can load them with the data command. Enter `data()` to see what is available and then `data(mydata)` to load one named, for example, mydata.

DATA ACQUISITION

This section gives a brief overview of data import and export, especially to and from SAS and SPSS. For a comprehensive discussion of data acquisition, see the *R Data Import/Export* manual.

In the example programs we will use, after importing data into R we will save it with the command `save.image(file="c:\\mydata.Rdata").` R uses the back slash to represent things like new lines `"\n"` so we use two in a row in filenames. Once saved, the following programs load it back into memory with the command `load(file="c:\\mydata.Rdata").`

For more details, see the section on ***Workspace Management***.

EXAMPLE TEXT FILES

We'll use the files below and read them several different ways. Note that the forward slash "/" has a special meaning in R, so you need to refer to the files as either "c:\\mydata…" or "c:/mydata". All our examples will use the "\\" form as it is more noticeable.

If you create these two files on your hard drive, then all of the examples of reading data will work. They will also save SAS, SPSS and R data sets that all the other examples will use. That way you can run them all by cutting and pasting the programs into any of these three packages. You can create these two files by using any text editor such as Notepad. Simply cut and paste the data into your editor and save the files on your C drive with the filenames below.

c:\mydata.csv	c:\mydata.txt (same, less names & commas)
id,workshop,gender,q1,q2,q3,q4 1,1,f,1,1,5,1 2,2,f,2,1,4,1 3,1,f,2,2,4,3 4,2,f,3,1, ,3 5,1,m,4,5,2,4 6,2,m,5,4,5,5 7,1,m,5,3,4,4 8,2,m,4,5,5,5	11f1151 22f2141 31f2243 42f31 3 51m4524 62m5455 71m5344 82m4555

THE R DATA EDITOR

R has a simple spreadsheet-style data editor. You access it by creating an empty data frame and then editing it:

```
mydata <- data.frame(id=0., workshop=0.,
```

```
  gender=" ", q1=0., q2=0., q3=0., q4=0.)
fix(mydata)
```

You can exit the editor and save changes by choosing File> Close or by clicking the X button. There is no File> Save option, which feels quite scary the first time you use it, but the data is indeed saved.

Note that the `fix` function actually calls the more aptly named `edit` function and then writes the data back to your original data frame as in: `mydata <- edit(mydata)`.

I strongly advise not using the edit function on data frames as I find it all too easy to begin editing with just `edit(mydata).` It will look identical but your changes will have nowhere to go and so will be lost when you exit. However, I mention it here because the `fix` function does not work when changing variable (column) or row names, but the `edit` function does.

READING DELIMITED TEXT FILES

Delimited text files use some delimiter, spaces, tabs or commas to separate each value. The default delimiter in R is tabs or plain spaces. Note that to use a blank space as a missing value, you must use either tabs or commas as delimiters.

Variable names are usually in the first row of a delimited file. You typically think of having a variable name for every column. However, if you are lacking just one variable name, R assumes you have an ID variable in the first column. It will then automatically assign it as the row names. Since most packages export data by writing out the names all variables including an ID (if one exists), the examples below do not use that feature.

We will read a comma separated value (CSV) file, c:\mydata.csv shown above. It is easiest if you use the approach that writes variable names in the top row, one for each variable including an ID variable if you have one. You can read this file using the following commands. Note that the print function in R prints the data frame as in `print(mydata).` Since this is done so frequently, it is the default function and so entering simply `mydata` in an interactive R session will also print it.

You can skip columns by specifying the "class" of each column to skip as NULL. However, the colClasses argument to do this requires you specify the classes of all columns, including any initial ID or row names variable. The classes include logical, integer, numeric, complex, character, raw), or "factor", "Date" or "POSIXct.

SAS	`* SAS Program to Read Delimited Text Files;`
	`DATA SASUSER.mydata;`
	`INFILE 'c:\mydata.csv' delimiter = ','`
	`  MISSOVER DSD lrecl=32767 firstobs=2 ;`
	`INPUT id workshop gender $ q1 q2 q3 q4;`

	PROC PRINT; RUN;
SPSS	* SPSS Program to Read Delimited Text Files. GET DATA /TYPE = TXT /FILE = 'C:\mydata.csv' /DELCASE = LINE /DELIMITERS = "," /ARRANGEMENT = DELIMITED /FIRSTCASE = 2 /IMPORTCASE = ALL /VARIABLES = id F2.1 workshop F1.0 gender A1.0 q1 F1.0 q2 F1.0 q3 F1.0 q4 F1.0 . LIST. SAVE OUTFILE='c:\mydata.sav'. EXECUTE.

R	# R Program to Read Delimited Text Files.
	# Default delimiters are tabs or spaces between values.
	# Note that "c:\\" in the file path is not a mistake.
	
	mydata<-read.table ("c:\\mydata.csv",header=TRUE,
	sep=",",row.names="id")
	print(mydata)
	
	# Now use colClasses to skip q1 and q2 with NULL.
	myCols<-
	read.table("c:/mydata.csv",header=TRUE,sep=",",row.names="id",
	colClasses=c("integer","integer","character","NULL","NULL",
	"integer","integer"))
	print(myCols)
	
	# Writes the file to disk.
	save.image(file="c:\\mydata.Rdata")

Using the example CSV file above, you can make it even easier to read. If you delete the name "ID" in the first row, you will have one fewer names than you have columns of data. When R sees this situation, it uses the first column as row names. I did not use that format because most programs will write the name of the ID variable when it exports data. You can then read such a file with the simple command:

```
mydata <- read.table("c:\\mydata.dat")
```

If your file has variable names in the first row but does ***not*** have an ID variable that can act as row labels, then use the form:

```
mydata <- read.table("c:\\mydata.dat", header=TRUE)
```

READING TEXT DATA WITHIN A PROGRAM

(DATALINES, CARDS, BEGIN DATA…)

Now that we have seen how to read a text file in the section above, we can more easily understand how to read data that is embedded within a program. R works by putting data into objects and then processing those objects with functions. In this case, we'll put the data into a character vector, named "mystring". Mystring will have only one really long value. Then we will read it just as we did in the previous example, but with `textConnection(mystring)` replacing `"c:\mydata.csv"` in the `read.table` function.

SAS	`* SAS Program to Read Data Within a Program;`
	`DATA SASUSER.mydata;`
	`INFILE DATALINES DELIMITER = ','`
	`  MISSOVER DSD firstobs=2 ;`
	`INPUT id workshop gender $ q1 q2 q3 q4;`
	`DATALINES;`
	`id,workshop,gender,q1,q2,q3,q4`
	` 1,1,f,1,1,5,1`
	` 2,2,f,2,1,4,1`
	` 3,1,f,2,2,4,3`
	` 4,2,f,3,1, ,3`
	` 5,1,m,4,5,2,4`
	` 6,2,m,5,4,5,5`
	` 7,1,m,5,3,4,4`
	` 8,2,m,4,5,5,5`
	`PROC PRINT; RUN;`

SPSS	`* SPSS Program to Read Data Within a Program.`
	`DATA LIST / id 2 workshop 4 gender 6 (A)`
	` q1 8  q2 10  q3 12  q4 14.`
	`BEGIN DATA.`
	` 1,1,f,1,1,5,1`
	` 2,2,f,2,1,4,1`
	` 3,1,f,2,2,4,3`
	` 4,2,f,3,1, ,3`
	` 5,1,m,4,5,2,4`
	` 6,2,m,5,4,5,5`
	` 7,1,m,5,3,4,4`
	` 8,2,m,4,5,5,5`
	`END DATA.`
	`LIST.`
	`SAVE OUTFILE='c:\mydata.sav'.`
	`EXECUTE.`

R	`# R Program to Read Data Within a Program.`
	
	`# This stores the data as one long text string.`
	`mystring<-`
	`("id,workshop,gender,q1,q2,q3,q4`

	1,1,f,1,1,5,1
	2,2,f,2,1,4,1
	3,1,f,2,2,4,3
	4,2,f,3,1, ,3
	5,1,m,4,5,2,4
	6,2,m,5,4,5,5
	7,1,m,5,3,4,4
	8,2,m,4,5,5,5")
	
	# This reads it just as a text file but processing it
	# first through the textConnection function.
	mydata<-read.table(textConnection(mystring),
	header=TRUE,sep=",",row.names="id")
	print(mydata)
	save.image(file="c:\\mydata.Rdata") #Writes the file to disk.

READING FIXED WIDTH TEXT FILES, 1 RECORD PER CASE

If you have a non-delimited text file with one record per case, you can read it using the following approach. R has nowhere near the flexibility in reading fixed-width text files that SAS and SPSS have. Other Open Source languages such as Perl or Python are extremely good at reading text files and converting them to a form that R can easily read.

This example adds quite a bit of complexity. Since file paths are seldom as simple as "c:\mydata.txt" I am storing it in a character vector named `myfile`. This approach also makes it easy to deal with long path names and also lets you put all the file references you use at the top of the file so you can change them easily.

Since fixed width text files seldom contain a header line of variable names, I store the names in another character vector, myVariableNames, and feed that to the col.names option in the read.fwf function.

In the example after this one, we will not need the workshop variable, so we will skip it here just to see how columns are skipped. By giving it a negative width of -1, we're telling R to skip one column. The `read.fwf` function is for Fixed Width Format files and it reads the data.

SAS	* SAS Program for Reading a Fixed-Width Text File, * 1 Record per Case; DATA SASUSER.mydata; INFILE 'c:\mydata.txt' MISSOVER; INPUT id 1-2 workshop 3 gender $ 4 q1 5 q2 6 q3 7 q4 8; RUN;
SPSS	* SPSS Program for Reading a Fixed-Width Text File, * 1 Record per Case. DATA LIST FILE='c:\mydata.txt' RECORDS=1

	/1 id 1-2 workshop 3 gender 4 (A) q1 5 q2 6 q3 7 q4 8. LIST. SAVE OUTFILE='c:\mydata.sav'. EXECUTE.

R	# R Program for Reading a Fixed-Width Text File,
	# 1 Record per Case.
	
	# Store the name of the file in a string variable.
	# Note that "c:\\" in the file path is not a mistake.
	myfile<-("c:\\mydata.txt")
	print(myfile)
	
	# Store the variable (column) names in a character vector.
	# Note that workshop does not appear on this list since we
	# are skipping it.
	myVariableNames<-c("id","gender","q1","q2","q3","q4")
	print(myVariableNames)
	
	# Store the widths of each variable in a numeric vector
	# -1 means there is a 1-column wide variable we want to skip,
	# which is workshop (just to see how to skip).
	myVariableWidths<-c(2,-1,1,1,1,1,1)
	print(myVariableWidths)
	
	# This is the part that actually reads the file.
	mydata<-read.fwf(
	file=myfile,
	width=myVariableWidths,
	col.names=myVariableNames,
	row.names="id",
	fill=TRUE,
	strip.white=TRUE)
	print(mydata)

READING FIXED WIDTH TEXT FILES, 2 RECORDS PER CASE

This section builds upon the one above, so don't start on this until you understand it first. This topic requires a new concept, a data structure called a ***list***. A list is a collection of objects. A data frame is itself a type of list. When reading one record per case, the widths of each variable are stored in a vector like:

`myVariableWidths<-c(2,-1,1,1,1,1,1).`

Now R needs one of these vectors for each record per observation. You tell R about these two vectors by combining them into a list using the form:

`list( myRecord1Widths, myRecord2Widths ).`

We will read the `mydata.txt` file again; this time as if it had two records for each case and line two contains the values for variables q5 through q8. We don't need the variable workshop

on the first line, nor do we need to read id, workshop or gender on the second line, so we'll skip those by using negative column widths.

Note that these programs do not save their files to disk as we will not use them in further examples.

SAS

```
* SAS Program for Reading Fixed Width Text Files,
* 2 Records per Case;
DATA temp; *We're not saving this one;
INFILE 'c:\mydata.txt' MISSOVER;
INPUT
  #1 id 1-2 workshop 3 gender 4  q1 5  q2 6  q3 7  q4 8
  #2                             q5 5  q6 6  q7 7  q8 8;
PROC PRINT;
RUN;
```

SPSS

```
* SPSS Program for Reading Fixed Width Text Files,
* 2 Records per Case.
DATA LIST FILE='c:\mydata.txt' RECORDS=2
 /1 id 1-2 workshop 3 gender 4 (A)  q1 5  q2 6  q3 7  q4 8
 /2                                 q5 5  q6 6  q7 7  q8 8.
LIST.
EXECUTE.
```

R

```
# R Program for Reading Fixed Width Text Files,
# 2 Records per Case.

# Store the name of the file in a string variable.
# Note that "c:\\" in the file path is not a mistake.
myfile<-("c:\\mydata.txt")
print(myfile)

# Store the variable (column) names in a vector:
myVariableNames<-c("id","group","gender","x1","x2","x3","x4",
                                   "x5","x6","x7","x8")
print(myVariableNames) #Optional, just checking.

# For multiple records, var widths for each record
# must be in separate vectors.
# The negative numbers on record 2 show we are skipping vars.
myRecord1Widths<-c( 2, 1, 1,1,1,1,1)
myRecord2Widths<-c(-2,-1,-1,1,1,1,1)

# You must then combine all the vectors containing
# variable widths into a single list:
myVariableWidths<-list(myRecord1Widths,myRecord2Widths)
print(myVariableWidths) #Prints both sets of widths.

#Now read the data:
mydata<-read.fwf(
```

	file=myfile, width=myVariableWidths, col.names=myVariableNames, row.names="id", na.strings="999", fill=TRUE, strip.white=TRUE) print(mydata)

IMPORTING DATA FROM SAS

R can read a SAS data set in xport format and, if you have SAS installed, directly from a regular SAS dataset with the extension `sas7bdat`. Although the foreign package is the most widely documented approach, it lacks important capabilities. Functions in the Hmisc package add the ability to read formatted values, variable labels and lengths.

SAS users rarely use the length statement, accepting the default storage method of double precision. This wastes a bit of disk space but saves programmer time. However since R saves all its data in memory, space limitations are far more important. If you use the length statement in SAS to save space, the `sasxport.get` function will take advantage of it.

You will need the foreign package for this example. It comes with R but must be loaded using the `library(foreign)` function. You also need the `Hmisc` package, which does not come with R but is very easy to install. For instructions, see the section, ***Installing Add-On Packages***.

The example below assumes you have a SAS xport format file. For much more information on reading SAS files, see ***An Introduction to S and the Hmisc and Design Libraries*** at http://biostat.mc.vanderbilt.edu/twiki/pub/Main/RS/sintro.pdf .

SAS Export	* SAS Program to Create Export Format File. * Something like this was done to create your * export format file. It would benefit from * labels, formats & length statements; LIBNAME To_R xport 'C:\mydata.xpt'; DATA To_R.mydata; SET SASUSER.mydata; RUN;

R Import	# R Program to Read a SAS Export File
	# SAS does not have to be installed on your computer.
	library(foreign) #Load the needed packages.
	library(Hmisc)
	mydata<-sasxport.get("c:\\mydata.xpt")
	print(mydata)

IMPORTING DATA FROM SPSS

Importing a data file from SPSS is done using the foreign package. It comes with R so you don't have to install it, but you do have to load it with the library command. The read.spss function is supposed to read both SPSS save files and portable files using exactly the same commands. However I have seen it work only intermittently on .sav files. Portable format files seem to work every time.

SPSS Export	`* SPSS Program to Create Export Format File.`
	`* Something like this was done to create your`
	`* portable format file.`
	
	`GET FILE='C:\mydata.sav'.`
	`EXPORT OUTFILE='c:\mydata.por'.`

R Import	`# R Program to Import an SPSS Data File.`
	`# This loads the needed package.`
	`library(Hmisc)`
	
	`# This Reads the SPSS file.`
	`mydata<-spss.get("c:\\mydata.por",use.value.labels=TRUE)`
	`print(mydata)`
	
	`# Save the data to a file.`
	`save.image(file="c:\\mydata.Rdata")`

EXPORTING DATA TO SAS & SPSS DATA SETS

Now let's use our example text file above to export data. The first example uses the default write.table function to create a text file with row and column names and values separated by tabs. Tabs have an advantage over commas as commas are used by some countries as decimal points, and exported text strings may also contain commas. By default it writes out the "m" and "f" for gender, including the quotes. Enter `help(write.table)` for output options.

The second two examples use the `write.foreign` function to write out a comma delimited text file along with a SAS or SPSS program file to read it. They require a library called `foreign`, which is loaded first with the library function. Note that these two examples write out the gender values as 1 and 2 for f and m respectively. See the section on ***Value Labels or Formats (& Measurement Level)*** if you want to change values. Note that the export to SPSS creates a syntax file that you must run to create the SPSS data file.

R export to text	`# R Program to Write a Text File.` `write.table(mydata,"c:/mydata2.txt",sep="\t")`

R export to	`# R Program to Write a SAS Export File`
	`# and a program to read it into SAS.`
	`library(foreign)`

SAS	`write.foreign(mydata,"c:/mydata2.txt","c:/mydata.sas", package="SAS")`

R export to
 	
SPSS
	`# R Program to Write an SPSS Export File`
	`# and a program to read it into SPSS.`
	`library(foreign)`
	`write.foreign(mydata,"c:/mydata2.txt","c:/mydata.sps", package="SPSS")`

SELECTING VARIABLES AND OBSERVATIONS

In SAS and SPSS, selecting ***variables*** for an analysis is simple while selecting ***observations*** is much more complicated. In R, these two processes are almost identical. As a result, variable selection in R is both more flexible and quite a bit more complex. However since you need to learn that complexity to select observations, it is not much added effort.

Selecting ***variables*** in SAS or SPSS is quite simple. Our example dataset contains the variables: `workshop, gender, q1, q2, q3, q4`. SAS lets you refer to them by individual name or in contiguous order separated by double dashes as in `workshop--q4.` SAS also uses a single dash to request variables that share a numeric suffix, `q1-q4`, regardless of their order in the data set. Selecting any variable beginning with a q is done with `q:`. SPSS allows you to list variables names individually or with contiguous variables separated by "to", as in `gender to q4`.

Selecting *observations* in SAS or SPSS requires the use of logical conditions with commands like IF, WHERE or SELECT IF. You never use that logic to select ***variables***. If you have used SAS or SPSS for long, you probably know dozens of ways to select observations, but you didn't see them all in the first introductory guide you read. With R, it is best to dive in and see them all because understanding them is the key to understanding other documentation, especially the help files.

SELECTING VARIABLES – VAR, VARIABLES=

Even though selecting variables and observations are done the same way, I'll discuss them in two different sections, with different example programs. This section focuses only on selecting variables.

Our example data frame has several important attributes:

- It has 6 variables or columns, which are automatically given index numbers of 1,2,3,4,5,6. In R you can abbreviate this as `1:6`. The colon operator isn't just shorthand as in `workshop to q4`. Entering `1:6` at the R console will cause it to actually generate the sequence, `1,2,3,4,5,6`.

- It has names: `workshop, gender, q1, q2, q3, q4`. They are stored within our data frame in an object called the ***names vector***. The `names` function accesses that vector, so entering `names(mydata)` will cause R to display them.

- Our data frame has two dimensions, rows and columns. These are referred to using square brackets as `mydata[rows,columns].` This section focuses on the second parameter, the columns (variables).

- Our data frame is also a list, with one dimension. You can address the elements of the list using two square brackets as in `mydata[[3]]` to select our third variable, `q1`.

R offers many ways to select variables (columns) from a data frame to use in an analysis. If you perform an analysis without selecting any variables, the R function will use all the variables if it can. That is much like SAS where you specify a data set but no VAR statement. For example, to get summary statistics on all variables (and all observations or rows), use `summary(mydata)`. You can substitute any of the examples below to choose a subset of variables. For example, `summary( mydata[ "q1"] )` would get a summary for just variable q1 using the data frame, mydata.

- You can select variables by ***index number* or a vector (column) of indexes**. For example, `mydata[ ,3]` selects all rows for the third variable or column, q1. If you leave out an index, it will assume you want them all. If you leave the comma out completely, R assumes you want a column, so `mydata[3]` is almost the same as `mydata[ ,3]` – both refer to our third variable, q1. Some functions require one approach or the other. See the section on ***Converting Data Structures*** for details.

  To select more than one variable using indexes, you must combine them into a numeric vector using the ***c*** function. So `mydata[ c(3,4,5,6) ]` selects variable 3 through 6. You will see this approach used many ways in R. You combine multiple objects into a single one in several ways to feed into functions that require a single object.

  The colon operator "`:`" can generate a numeric vector directly, so `mydata[3:6]` selects the same variables.

  If you use a negative sign on an index, you will exclude those columns. For example, `mydata[-c(3,4,5,6), ]` will ***exclude*** those variables. The colon operator can generate longer strings of numbers, but it's tricky. The form `-3:6` generates the values from -3 to +6 or
  -3,-2,-1,0,1,2,3,4,5,6. The isolate function `I()` in R exists to clarify such occasional confusion. You use it in the form, `mydata[ ,-I(3:6) ]` showing R that you want the minus sign to apply to the just the set of numbers from +3 through +6.

Selection by indexes is the most fundamental approach in R because all R's data structures always have them. They do not have to have names.

- You can select a column by ***name*** in quotes, as in `mydata["q1"].` R is still expecting the form `mydata[row,column]`, but when you supply only one parameter, it assumes it is the column. So `mydata[ ,"q1"]`works as well. If you have more than one name, you must combine them into a single character vector using the ***combine*** or ***c*** function. For example,
  `mydata[ c("q1","q2","q3","q4") ].`

  Unfortunately, the colon operator does not work directly with character prefixes, but you can paste the letter "q" onto the numbers you generate using that operator. This code generates the same list as the paragraph above and stores it in a character vector called myqs. You can use this approach to generate variable names to use in a variety of circumstances. Note that merely changing the 4 below to 400 would generate the sequence q1 to q400. The `sep=""` parameter tells R to separate the letter q and the generated numbers with nothing.
  ```
  myqs <-paste( "q", 1:4, sep="")
  summary( mydata[myqs] )
  ```

- You can select a column by a ***logical vector*** of TRUE/FALSE values. For example, `mydata[c(FALSE,FALSE,TRUE,FALSE,FALSE,FALSE)]` will select the third column, q1 because the 3rd value is TRUE and the 3rd column is q1. As in SAS or SPSS, the digits 1 and 0 can represent TRUE and FALSE. In R, the as.logical function tells R to do that. so we could also select the third variable with:
  `mydata[ as.logical( c(0,0,1,0,0,0) ) ]`

  It is unlikely that you would ever type in a logical vector like this. Instead, a logical statement such as `names(mydata)=="q1"` generates the logical vector you need. So `mydata[ colname(mydata)=="q1" ]` is another way of selecting q1. The "!" sign represents NOT so you can also use that vector to get all the variables except for q1 using either form:
  ```
  mydata[  names(mydata)!="q1" ]
  mydata[ !names(mydata)=="q1" ]
  ```
  You could also use the colon operator and the paste function demonstrated in the bullet above to generate long patterns of TRUE/FALSE values based on long sets of numerically suffixed names.

- You can select a column using $ notation, which has the form `mydata$myvar`, e.g. `mydata$q1`. This is referred to several ways in R, including "$ prefixing", "prefixing by dataframe$" or "$ notation". When you use this method to select multiple variables,

you need to combine them into a single object like a data frame, as in `summary( data.frame(mydata$q1,mydata$q2) )`. Having seen the `combine` function, your natural inclination might be to use it for multiple variables as in: `summary( c(mydata$q1,mydata$q2) )`. This would indeed make a single object, but certainly not the one a SAS or SPSS user expects. It would stack them both into a single variable with twice as many observations!

- You can select a variable from a data frame by its simple column name, e.g. just q1, but only if you attach the data frame first. Unlike SAS and SPSS, you can have many active datasets open and equally accessible at once. You can actually correlate X from one data frame with Y stored in another!

  After you submit the function, `attach(mydata)`, you can refer to just q1 and R will know which one you mean. This works when ***selecting*** existing variables but is best avoided when ***creating*** them. This is because any variable can also exist all by itself in R's workspace. So when adding new variables to a data frame, you need to use any of the above methods that make it absolutely clear where you want the variable stored. With this approach getting summary statistics on multiple variables might look like, `summary( data.frame(q1,q2) )`.

- You can select variables with the `subset` function. The main advantage to this is that it is the only built-in approach to selecting contiguous sets of variables such as q1-q4 (in SAS) or q1 to q4 (in SPSS). It follows the form, `subset(mydata, select=q1:q4)`
  For example, when used with the summary function, it would appear as
  `summary( subset(mydata,select=q1:q4) )`
  Note that the additional spaces added around the subset function help increase readability. R ignores them.

- You can select variables by using a list index as in `mydata[[3]]` to choose the third variable. This approach is usually used for under other circumstances. With this approach you cannot use the colon operator, so `mydata[[3:6]]` is invalid.

The examples below demonstrate many ways to select variables. To make it easier to see the result of the selection, we will use the print function. When working interactively, this is the default function, so `mydata["q1"]` and `print( mydata["q1"] )` are equivalent. However to give you the feel how the selection works in all functions, I use the longer form.

SAS	`* SAS Program for Selecting Variables;`
	`OPTIONS _LAST_=SASUSER.mydata;`
	`PROC PRINT; RUN;`
	`PROC PRINT; VAR workshop gender q1 q2 q3 q4; RUN;`
	`PROC PRINT; var workshop--q4; RUN;`
	`PROC PRINT; var workshop gender q1-q4; RUN;`

	* Creating a data set from selected variables;
	DATA SASUSER.myqs;
	  SET SASUSER.mydata(KEEP=q1-q4);
	  RUN;

SPSS	* SPSS Program for Selecting Variables. LIST. LIST VARIABLES=workshop,gender,q1,q2,q3,q4. LIST VARIABLES=workshop TO q4. * Creating a data set from selected variables. SAVE OUTFILE='c:\myqs.sav' /KEEP=q1 TO q4. EXECUTE.

R	# R Program for Selecting Variables.
	# Uses many of the same methods as selecting observations.
	load(file="c:\\mydata.Rdata")
	
	# This refers to no particular variables, so all are printed.
	print(mydata)
	
	#---SELECTING VARIABLES BY INDEX
	# These also select all variables by default.
	print(mydata[ ])
	print(mydata[,])
	
	# Select just the 3rd variable, q1.
	print(mydata[3]) #selects q1.
	
	# These all select the variables q1,q2,q3 and q4 by indexes.
	print(mydata[c(3,4,5,6)]) #selects q vars by their indexes.
	print(mydata[3:6]) # generates indexes with ":" operator.
	print(mydata[-c(1,2)]) #selects q vars by excluding others.
	print(mydata[-I(1:2)]) #colon operator could exclude many.
	
	# If you use a range of columns repeatedly, it is helpful
	# to store the whole range in a numeric vector.
	myindexes <- 3:6
	print(myindexes) #Just shows you the indexes.
	print(mydata[myindexes])
	print(mydata[-myindexes]) #Excludes 3:6 with minus sign.
	
	# The "which" function can find an index number for you.
	# Here we store that single index in myvar
	# The %in% function compares the names to the whole
	# set of names at once.
	myvars <- which(names(mydata) %in% c("q1","q2","q3","q4"))
	print(myvars) #Prints 3,4,5,6.
	print(mydata[,myvars])

```
# This displays the indexes for all variables.
# Column names are stored in mydata as a character vector.
# The "names" function extracts those names.
# The data.frame function makes it a data frame,
# which numbers them.
print( data.frame( myvars=names(mydata) ) )

#---SELECTING VARIABLES BY NAME (can’t exclude with minus sign)
mydata["q1"] #selects q1.
mydata[ c("q1","q2","q3","q4") ] #selects the q variables.

# The subset function makes selecting contiguous
# variables easy using the colon operator.
print( subset(mydata, select=q1:q4) )

# This approach saves a list of variable names to use.
myQnames<-c("q1","q2","q3","q4")
print(myQnames) #Prints just the names.
print(mydata[myQnames])

# This also saves a list of variable names to use,
# but it generates it by appending the letter q to the
# automatically generated sequence 1,2,3,4 from the
# form 1:4.
myQnames <-paste( "q", 1:4, sep="")
print(myQnames) #Prints the names.
print(mydata[myQnames]) #Prints chosen variables.

# This demonstrates again that we can select contiguous
# variables by their indexes. We repeat this here as a
# prelude to getting indexes by name.
print(mydata[,3:6])

# You can also store single index values in numeric vectors
# that have only that single number.
myA <- which( names(mydata)=="q1" )
print(myA) #prints the number 3.
myZ <- which( names(mydata)=="q4" )
print(myZ) #prints the number 6.
print(mydata[ ,myA:myZ ]) #prints variables 3 thru 6.

#---SELECTING VARIABLES BY LOGIC

# The "==" operator compares every member of a vector
# to a value, returning a logical vector of TRUE/FALSE values.
# That vector is then used to choose variables.
# Note that the vector length will match the number of variables
```

```
# so it cannot itself be stored as a variable
# in the data frame mydata. It will just be in the workspace.

# Manually create a vector to get just q1.
# You probably would not do this, but it demonstrates
# the basis for the next example.
# The as.logical function turns 1 & 0 into TRUE & FALSE.
myq <- as.logical( c(0,0,1,0,0,0) )
print(myq) #print the TRUE/FALSE values.
print( mydata[ myq ] ) #print q1.

# This generates the same logical vector automatically.
myq <- names(mydata)=="q1"
print(myq)
print( mydata[ myq ] )

# This uses the OR operator, "|" to select q1 through q4,
# and store the resulting logical vector in myqs.
myqs <- names(mydata)=="q1" |
        names(mydata)=="q2" |
        names(mydata)=="q3" |
        names(mydata)=="q4"
print(myqs)
print( mydata[myqs] )

# This uses the %in% operator to select q1 through q4.
# It is much shorter than the one above that uses OR.
myqs <- names(mydata) %in% c("q1","q2","q3","q4")
print(myqs)
print( mydata[myqs] )


# If you use the attach function to access mydata,
# you can refer to q1 by itself, but it's good to
# detach the data frame later. This approach is fine
# for reading data, but not advisable for writing it.
attach(mydata)
print(gender)
print(q1)
print( data.frame(q1,q2,q3,q4) )

#---CREATING A NEW DATA FRAME OF SELECTED VARIABLES

# Any of the above examples can place the selected
# variables into a new data frame. These four are equivalent
# as long as the data frame is attached.
myqs <- mydata[3:6]
myqs <- mydata[ c("q1","q2","q3","q4") ]
```

```
myqs <- data.frame(q1,q2,q3,q4)
myqs <- subset(mydata, select=q1:q4)
print(myqs)
```

SELECTING OBSERVATIONS – WHERE, IF, SELECT IF

It bears repeating that the approaches that R uses to select observations are, for the most part, the same as those discussed above for selecting variables. This section focuses only on selecting observations.

Our example data frame has several important attributes:

- Our data frame has 8 observations or rows, which are automatically given index numbers, or indexes of 1,2,3,4,5,6,7,8. In R you can abbreviate this as 1:8.
- It has row names, `"1","2","3","4","5","6","7","8".` They are stored within our data frame in an object in called the ***row names vector***. Note that the quotes around them show that they are stored as characters, not as numbers. So they ***cannot*** be abbreviated simply 1:8. You can abbreviate the with `as.character(1:8)`. Row names can also be character values that are easier to identify, such as "Ann","Bob","Carla"….
- Our data frame has two dimensions, rows and columns. These are referred to as mydata[rows,columns]. This section focuses on the first parameter, the rows.

There are many ways to select observations (rows) from a data frame. As in SAS or SPSS, if you select no rows, you'll use all the data. For example, to get summary statistics on all rows for all columns, use `summary( mydata )`. You can substitute any of the examples below to choose a subset of observations. For example,
`summary( mydata[mydata$gender=="m", ] )`
would get a summary for just the males, on all variables. The fact that the column position after the comma is blank tells R to use all the variables.

- You can select observations by ***index***. For example, `mydata[8 , ]` selects all the variables for row 8.

  To select more than one variable using indexes, you must combine them into a numeric vector using the ***combine*** or ***c*** function. So `mydata[ c(5,6,7,8),  ]` selects rows 5 through 8, which happen to be the males. The colon operator `":"` can generate a numeric vector directly, so `mydata[5:8, ]` selects the same observations.

  If you use a negative sign on an index, you will exclude those observations. For example

`mydata[-c(1,2,3,4), ]` will ***exclude*** the first four records, the females. The colon operator can abbreviate this as well, but it's tricky. The form `-1:4` generates the values from -1 to +4 or
-1,0,1,2,3,4. The isolate function in R exists to clarify such occasional confusion. You use it in the form, mydata[-I(1:4),] showing R that you want the minus sign to apply to the just the set of numbers 1,2,3,4.

- You can select observations by ***name*** in quotes, as in `mydata[ "1", ]` or `mydata["Ann", ]` (if you created such row names, more on that later). If you have more than one name, you must combine them into a single character vector using the combine or `c` function. For example,
  `mydata[ c("1","2","3","4"), ]` or
  `mydata[ c("Ann","Carla","Bob","Sue"), ]`
  Note that even if your names appear to be numbers, they are still stored characters. So you ***cannot*** abbreviate them using the form 1:8. However, you could generate them using the colon operator and force them to become character using the `as.character` function as in `as.character(1:8)`.

- You can select observations by a ***logical vector*** of TRUE/FALSE values. For example, `mydata[c(TRUE,TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,FALSE),]` will select the first four rows, the females. The ! sign represents NOT so you can also use that vector to get the males with
  `mydata[!c(TRUE,TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,FALSE),]`

  As in SAS or SPSS, the digits 1 and 0 can represent TRUE and FALSE. In R, the as.logical function tells R to do that. so we could also select the first four rows with:
  `mydata[ as.logical( c(1,1,1,1,0,0,0,0) ), ]`
  Note that the location of brackets, parentheses and commas starts to get rather tedious and error-prone at this point. A context-sensitive editor such as TINN-R or ESS would be a big help in avoiding errors.

  A logical statement such as `rownames(mydata)=="8"` generates a logical vector like the one in the paragraph above but with a single TRUE entry.
  So `mydata[ rowname(mydata)=="8", ]` is another way of selecting the 8th observation.

  The "!" sign represents NOT so you can also exclude only the 8th observation using either form:
  `mydata[  rownames(mydata)!="8", ]`
  `mydata[ !rownames(mydata)=="8", ]`

This is one place I find the form mydata$varname particularly appealing. If we want to select the females in our data frame, `mydata["gender"]=="f"` will create the logical vector we need. We can apply it in the form `mydata[ mydata["gender"]=="f", ]` but I find the style of `mydata[ mydata$gender=="f", ]` much less busy. Of course the easiest to read is `mydata[ gender=="f", ]` but that does require attaching the file.

- You can select observations using the `subset` function. You simply list your logical condition under the subset argument as in:
  `subset(mydata,subset=gender=="f")`

Note that when selecting variables, there is the $ prefix form, mydata$gender and the attached form of just gender. When selecting observations, these two have no equivalents.

SAS

```
* SAS Program to Select Observations;
PROC PRINT data=SASUSER.mydata;
  WHERE gender='m';
  RUN;
PROC PRINT data=SASUSER.mydata;;
  WHERE gender="m" & q4=5;

DATA SASUSER.males;
  SET SASUSER.mydata;
  WHERE gender="m";
  RUN;
DATA SASUSER.females;
  SET SASUSER.mydata;
  WHERE gender="f";
  RUN;
```

SPSS

```
* SPSS Program to Select Observations.
TEMPORARY.
SELECT IF(gender = "m").
LIST.
EXECUTE.

TEMPORARY.
SELECT IF(gender = "m" & q2 >= 5).
LIST.
EXECUTE.

TEMPORARY.
SELECT IF(gender = "m").
SAVE OUTFILE='C:\males.sav'.
EXECUTE .
```

	TEMPORARY. SELECT IF(gender = "f"). SAVE OUTFILE='C:\females.sav'. EXECUTE .

R	# R Program to Select Observations.
	
	load(file="c:\\mydata.Rdata")
	attach(mydata)
	print(mydata)
	
	#---SELECTING OBSERVATIONS BY INDEX
	# Print all rows.
	print(mydata[1:8,])
	
	# Just the males:
	print(mydata[5:8,])
	
	# Negative numbers exclude rows.
	# So this excludes the females in rows 1 through 4.
	# The isolate function is used to apply the minus
	# to 1,2,3,4 and prevent -1,0,1,2,3,4.
	print(mydata[-I(1:4),])
	
	# The which function can find the index numbers
	# of the true condition.
	which(gender=="m")
	
	# You can use those index numbers like this.
	print(mydata[which(gender=="m"),])
	
	# You can make the logic as complex as you like:
	print(mydata[which(gender=="m" & q4==5),])
	
	# You can save the indices to a numeric vector stored OUTSIDE the
	# original data frame. Otherwise how would you store the 5,6,7,8
	# values in a data frame that has 8 rows?
	
	happyGuys <- which(gender=="m" & q4==5)
	print(happyGuys)
	
	# Now you can use the happyGuys vector to select observations.
	# Note that even though happyGuys is a variable name, it is
	# not used in quotes.
	print(mydata[happyGuys,])
	
	#---SELECTING OBSERVATIONS BY LOGICAL VECTOR

```
# A logical comparison creates a logical vector that
# has a length equal to the original data frame.
# It will be TRUE for the males and FALSE for the females.
print(mydata$gender=="m")

# This selects males by putting the logical
# vector in the rows position.
print( mydata[mydata$gender=="m", ] )

# Since we have used attach(mydata) we can dispense
# with the mydata$ prefix on gender to choose the males.
print( mydata[gender=="m", ] )

# You can make the logic as complex as you like.
# When q4==5, the student is very satisfied overall.
print( mydata[ gender=="m" & q4==5, ] )

# When the logic gets complex, you might
# want to save the logical vector it to the dataset
# (just like an SPSS filter variable).
# Since a logical vector is as long as the original
# variables, the new variable is a good match,
# so we'll save it there.
mydata$happyGuys <- (gender=="m" & q4==5)
print(mydata) #to show how it is added to mydata.

# You can use the saved logical vector to select observations.
# Note that when we use a saved logical vector to select,
# it is not put in quotes. Note also that since happyGuys did
# not exist when the file was attached, you must use the full
# form mydata$happyGuys.
print( mydata[mydata$happyGuys, ] )

#---SELECTING OBSERVATIONS BY ROW NAME

# This shows what the row names look like.
# Since they're in quotes, they make up
# a character vector.
row.names(mydata)

# We can select rows by their row name, although this
# is rarely done compared to selecting variables by column name.
# This prints the first 4 cases selected by their row name.
print( mydata[ c("1","2","3","4"), ] )

# Having numbers as row names is not very clear.
# This assigns more useful row names.
mynames <- c("Ann","Cary","Sue","Carla",
```

```
             "Bob","Scott","Mike","Rich")
print(mynames)

# Store those new names in mydata.
row.names(mydata) <- mynames
print(mydata)

# This again selects by row name,
# but it's much more clear.
print( mydata[ c("Ann","Cary","Sue","Carla"), ] )

# If we have a set of row names we use often we can save it
# to a character vector. Note that this vector's length is
# not equal to the number of rows in the data frame, so it
# cannot be stored there.
myGals <- c("Ann","Cary","Sue","Carla")
print(myGals)

# And then use it to select rows. Note that it is
# not enclosed in quotes when used this way.
print( mydata[ myGals, ] )

# Let's pretend group has lots of values and we want to select
# only some groups by listing their values.
myGals <- row.names(mydata)=="Ann"  |
          row.names(mydata)=="Cary" |
          row.names(mydata)=="Sue"  |
          row.names(mydata)=="Carla"
print(myGals)

#The %in% function can shorten this up.
print( row.names(mydata) %in% c("Ann","Cary","Sue","Carla") )

# Just like our example for working with variable names
# we can use "which" to get index values
firstMale <- which( row.names(mydata)=="Bob" )
print(firstMale)
lastMale <- which( row.names(mydata)=="Rich" )
print(lastMale)

# This last one selects both rows and columns at once.
# All the rules that work for rows also work for columns.
mydata[firstMale:lastMale, c("q1","q2","q3","q4")]

#---CREATING A NEW DATA FRAME OF SELECTED OBSERVATIONS

# Any of these methods can put the selected
# observations in a new data frame.
```

```
myMales <- mydata[mydata$gender=="m", ]
print(myMales)
myFemales <-mydata[mydata$gender=="f", ]
print(myFemales)
```

SELECTING BOTH VARIABLES AND OBSERVATIONS

While the sections above have focused on selecting variables and observations separately, you can combine methods in most cases. For example, printing variables gender through q4 for just the females could be done with any of these approaches:

```
print( mydata[1:4,2:6] )
```

```
attach(mydata)
print( mydata[gender=="f", c("gender","q1","q2","q3","q4")] )
detach(mydata)
```

```
print( subset(mydata,subset=gender=="f",select=gender:q4) )
```

CONVERTING DATA STRUCTURES

In SAS and SPSS, there is only one data structure, the data set. Within that, there is only one structure, the variable. It seems absurdly obvious, but we need to say it: all SAS and SPSS procedures accept variables as input.

On the other hand, R has several data structures: data frames, vectors, lists and matrices. R takes advantage of these by having the same function (procedure) do different things depending on what you give it. So to control a function you have to be aware of what type of object you're giving it, the types of objects the function is able to accept, what it does with each and how to convert from one data structure to another.

We have seen several instances of this in our examples. We saw that the summary function cannot accept multiple variables of the form `summary(mydata$q1,mydata$q2)` but it will accept a single data frame that contains only those variables as in `summary( data.frame(mydata$q1,mydataq2) )`. However, some functions will indeed accept the first form.

In the section on ***Selecting Variables***, I said that `mydata[ ,3]` and `mydata[3]` were almost the same. Similarly, I said `mydata["q1"]` and `mydata[ ,"q1"]` were almost the same. For many functions those approaches are interchangeable. But some procedures are pickier than others and require very specific data structures. These commands pass the data in the form of a data frame `mydata[3], mydata["q1"].`
These pass the data in the form of a vector: `mydata[ ,3], mydata[ ,"q1"].`

I also said above that the same methods are used to select variables and observations. An exception to that is that selecting a column using the form `mydata[ ,3]` will pass the data as a data frame while selecting a row using the same type of notation, `mydata[3, ]` passes the data as a vector!

If you have having a problem figuring out which form of data you have, there are functions that will tell you. For example, `class(mydata)` will tell you its class is "data frame" and `mode(mydata)` will tell you "list". So functions that require either form will work with it. There are also a series of functions that test the status of an object, and they all begin with "is." For example, `is.data.frame(mydata[3])` will display TRUE but `is.vector((mydata[3])` will display FALSE.

Some of the functions you can use to convert from one structure to another are below.

DATA CONVERSION FUNCTIONS	
Vectors to data frame	`data.frame(x,y,z)`
Vectors to columns of a matrix	`cbind(x,y,z)`
Vectors to rows of a matrix	`rbind(x,y,z)`
Vectors combined into one long one	`c(x,y,z)`
Data frame to matrix	`as.matrix(mydataframe)`
Matrix to data frame	`as.data.frame(mymatrix)`
A vector to a matrix	`as.matrix(myvector)`
Matrix to one very long vector	`as.vector(mymatrix)`
List containing one vector to just a vector	`unlist(mylist)`

DATA MANAGEMENT

TRANSFORMING VARIABLES

Unlike SAS, R has no separation of phases like the data step and proc steps. It is more like SPSS where as long as you have data read in, you can modify it. In fact, you can even modify variables in the middle of procedures as in this example where we take the square root of q4 before getting summary statistics on it: `summary( sqrt(mydata$q4) )`.

R performs transformations such as adding or subtracting variables on the whole variable at once, as do SAS and SPSS. In other words, although R has loops, they are not needed for this type of manipulation. The basic transformations include sqrt for square root, log for natural logarithm, log10 for the base 10 logarithm and so on. The equivalent to the MEANS functions in SAS and SPSS is called `rowmeans`.

In the section ***Selecting Variables***, we saw various ways to select variables: by index, by column name, by logical vector, using the style `mydata$myvar`, by using simply the variable name after you have attached a data frame and using the `subset` function. Usually the best way to name a new variable is using the `mydata$varname` format. As for the right side of the equation, you can use that method too, but it is longer:

```
mydata$sum <- mydata$q1 + mydata$q2 + mydata$q3 + mydata$q4
```

If you do a lot of transformations, attaching the data frame will allow you use the non-prefixed names, simplifying things greatly. However if you are adding the new variable to your data frame, you definitely want to use the full `mydata$varname` form on the left side of the equation:

```
attach(mydata)
mydata$sum <- q1+q2+q3+q4
detach(mydata)
```

Another way to use the shorter names without attaching and detaching is with the transform function. It is the equivalent to attaching a data frame, performing as many transformations as you like using short variable names and then detaching the data. You can even use the shorter name form in the created variable(s). Strangely enough, it uses "=" instead of "<-". It looks like:

```
mydata <- transform(mydata, sum=q1+q2+q3+q4)
```

If you have many transformations, it's easier to read them on separate lines:

```
Mydata <- transform( mydata,
  sum=q1+q2+q3+q4,
  mean=sum/4)
```

You can create a new variable using the index method too. This would create a variable 7 where previously we had only 6. R will also give it a column name of V7. If you're using the index approach, it is probably easier to initialize a new variable by binding a new variable to mydata. The column bind function, `cbind`, lets you name the new variable, initialize it to zero and then bind it to mydata:

```
mydata<-data.frame( cbind( mydata, sum=0.) )
# Now put the sum into the new mydata$sum variable.
mydata[7]<-q1+q2+q3+q4
```

SAS	`* SAS Program for Transforming Variables;`
	`DATA SASUSER.mydata;`
	`SET  SASUSER.mydata;`
	`totalq=(q1+q2+q3+q4);`
	`logtot=log10(totalq);`
	`mean1=(q1+q2+q3+q4)/4;`
	`mean2=mean(of q1-q4);`
	`PROC PRINT; RUN;`

SPSS	`* SPSS Program for Transforming Variables.`
	`GET FILE='C:\mydata.sav'.`
	`COMPUTE Totalq=q1+q2+q3+q4.`
	`COMPUTE Logtot=lg10(totalq).`
	`COMPUTE Mean1=(q1+q2+q3+q4)/4.`
	`COMPUTE Mean2=MEAN(q1 TO q4).`
	`SAVE OUTFILE='C:\mydata.sav'.`
	`LIST.`
	`EXECUTE.`

R	`# R Program for Transforming Variables.`
	`load(file="c:\\mydata.Rdata")`
	`print(mydata)`
	
	`#Create a score called totalq`
	`mydata$totalq <- mydata$q1 + mydata$q2`
	`+ mydata$q3 + mydata$q4`
	
	`#Get the natural logarithm of totalq:`
	`mydata$logtot <- log( mydata$totalq )`
	
	`#Get the mean of the q variables.`
	`#If any is missing, the result is missing.`
	`mydata$meanq1  <- ( mydata$q1 + mydata$q2`
	`+ mydata$q3 + mydata$q4 )/4`
	
	`# Get the mean of the q variables a different way.`
	`# The result will be missing only if all qs are missing.`
	`# This selects the qs using the select function.`
	`myqs <- subset(mydata,select=c(q1,q2,q3,q4) )`
	`# This gets the row means, removing missing values.`
	`mydata$mymeanq2 <- rowMeans( myqs, na.rm=TRUE )`
	
	`print(mydata) #Prints the results.`
	`save.image(file="c:\\mydata.Rdata") #Saves the R file.`

CONDITIONAL TRANSFORMATIONS

Conditional transformations apply different formulas to various subgroups of the data. For example, the formulas for recommended daily allowances of vitamins differ for males and females.

Below are the logical operators for SAS, SPSS and R and how a few comparisons differ.

LOGICAL OPERATORS See also `help(Logic) and help(Syntax)`.			
	SAS	**SPSS**	**R**
Equals	= or EQ	= or EQ	==
Less than	< or LT	< or LT	<
Greater than	> or GT	> or GT	>
Less or equal	<= or LE	<= or LE	<=
Greater or equal	>= or GE	>= or GE	>=
Not equal	^=, <> or NE	~= or NE	!=
And	& or AND	& or AND	&
Or	| or OR	| or OR	|
0<=x<=1	0<=x<=1	(X >= 0) AND (X <= 1)	(x >= 0) & (x<=1)
Missing value size	Missing is less than all numbers	Comparisons with missing are set to NA	Comparisons with missing are set to NA
Symbol to represent missing in comparisons	"."	MISSING or SYSMIS	is.na(x) (Note that ==NA can never be true.)

The examples below demonstrate a variety of conditional transformations.

SAS	* SAS Program for Conditional Tranformations; DATA SASUSER.mydata; SET SASUSER.mydata; If q4= 5 then x1=1; else x1=0; If q4>=4 then x2=1; else x2=0; If workshop=1 & q4>=5 then x3=1; else x3=0; If gender="f" then scoreA=2*q1+q2; Else scoreA=3*q1+q2; If workshop=1 and q4>=5 then scoreB=2*q1+q2; Else scoreB=3*q1+q2;

SPSS	*SPSS Program for Conditional Transformations.
	GET FILE=("c:\mydata.sav")
	COMPUTE X1=0.
	IF (q4 EQ 5) X1=1.
	
	COMPUTE X2=0.
	IF (q4 GE 4) X2=1.
	
	COMPUTE X3=0.
	IF (gender EQ 'f' AND Q4 GE 5) X3=1.
	
	COMPUTE scoreA=3*q1+q2.
	IF (gender='f') scoreA=2*q1+q2.
	
	COMPUTE scoreB=3*q1+q2.
	IF (workshop EQ 1 AND q4 GE 5) scoreB=2*q1+q2.
	EXECUTE.

R	# R Program for Conditional transformations.
	load(file="c:\\mydata.Rdata")
	print(mydata)
	attach(mydata) #Makes this the default dataset.
	
	#Create a series of dichotomous 0/1 variables
	
	# The new variable q4SAgree will be 1 if q4 equals 5, otherwise zero.
	# It identifies the people who strongly agree with question 4.
	mydata$q4Sagree <- ifelse(q4 == 5, 1,0)
	print(mydata)
	
	# The new variable, q4agree will be 1 if q4
	# is greater than or equal to 4.
	# It identifies people who agree with question 4.
	mydata$q4agree <- ifelse(q4 >= 4, 1,0)
	print(mydata)
	
	# An example with more complex logic,
	# which must be enclosed in parenthesis:

```
# The variable workshop1q4ge5 will be 1
# when workshop 1 has q4 greater than or equal to 4,
# i.e. the people only in workshop1 agree to item 5.
mydata$workshop1agree <- ifelse( (workshop == 1 & q4 >=4 ) , 1,0)
print(mydata)

# Conditional tranformation uses different formulas for
# males & females. However, it specifies only the female
# condition, assuming male is true whenever female is false.
# So if gender were missing, they would get the male code.
# The structure is ifelse(logic, WhatToDoIfTrue, WhatToDoIfFalse).

mydata$score <- ifelse( gender=="f" ,
   (mydata$score <- (2*q1)+q2),  #for females.
   (mydata$score <- (3*q1)+q2) ) #for males.
mydata

# This example also assigns the different formulas for males & females.
# However, it checks for the male gender so it will not assume missing
# genders are male. The nesting involved is quite a bit more complex
# than the example above.

mydata$score <- ifelse( gender=="f" ,
   (mydata$score1 <- 2*q1+q2),
   ifelse( mydata$gender=="m",
      (mydata$score <- 3*q1+q2), NA )
   )
print(mydata)

# Conditional formula using more complex logic in parenthesis.
# If workshop or q4 are missing, it will use the second formula.

mydata$score2 <- ifelse( (workshop==1 & q4>=4) ,
   (mydata$score <- 2*q1+q2),
   (mydata$score <- 3*q1+q2) )
print(mydata)

# Stop making mydata the default data frame.
detach(mydata)
```

CONDITIONAL TRANSFORMATIONS TO ASSIGN MISSING VALUES

R represents missing values with NA, for Not Available. The letters NA are also an object in R that you can use to assign missing values. Unlike SAS, the value used to store the NA is not the smallest number your computer can store, so logical comparisons such as x<0 will result in NA when x is missing.

When importing data, blanks are read as missing (when blanks are not used as delimiters) as is the string NA. But if you have other values, you will of course have to tell R which values are missing. The `read.table` function provides an argument, `na.strings,` that allows you to set missing values. However, it applies the values to all variables, which is unlikely to be of use. For example a 2-column variable such as years of education may have 99 represent missing, but the variable age may have 99 as a valid value.

Periods that represent missing values in SAS cause R to read the whole variable as a character vector. So you have to first fix the missing values and then convert it to numeric using the `as.numeric()` function.

Note that since any logical comparison on NAs results in an NA outcome, even `q1==NA` will not be TRUE when q1 is indeed NA. So if you wanted to substitute another value such as the mean, you would need to use the `is.na` function. It will be TRUE when a value is NA:
`mydata[is.na(mydata$q1),"q1"] <- mean(mydata$q1,rm.na=TRUE)`

SAS	`* SAS Program to Assign Missing Values;`
	`data SASUSER.mydata;`
	`set  SASUSER.mydata;`
	
	`*Convert 9 to missing, one at a time.`
	`if q1=9 then q1=.;`
	`if q2=9 then q2=.;`
	`if q3=9 then q2=.;`
	`if q4=9 then q4=.;`
	
	`*This does the same thing but is quicker for lots of vars;`
	`array q q1-q4;`
	`do over q;`
	`  if q=9 then q=.;`
	`end;`

SPSS	`* SPSS Program to Assign Missing Values.`
	`GET FILE=("c:\mydata.sav")`
	`MISSING q1 TO q4 (9).`
	`SAVE OUTFILE=("c:\mydata.sav")`

R	`# R Program to Assign Missing Values.`
	`# NA is the missing value code in R.`
	
	`load(file="c:\\mydata.Rdata")mydata`
	`attach(mydata)`
	
	`# This finds the rows where q1==9 and chooses the variable`
	`# q1 by its column number, 3. There are no nines in the data,`
	`# but you can put a few there in q1 manually in the data editor`
	`# using fix(mydata).`
	`fix(mydata)`

```
mydata[q1==9,3] <- NA
print(mydata)

#This uses q1==9 to select the rows
# and chooses the variable using its name, q1.
mydata[q1==9,"q1"] <- NA
print(mydata)

#This uses the ifelse function on the whole of q1 at once.
mydata$q1 <- ifelse( q1 == 9, NA, mydata$q1)
print(mydata)

# This replaces every value 9 of EVERY variable
# in mydata, be careful how you use it!
mydata[mydata==9] <- NA
print(mydata)

#--------------------------------------------
# Below are more complex examples that let
# you convert many variables at once.
#--------------------------------------------

#Create a function that replaces 9s with NAs.
my9isNA <- function(x) {x[x==9] <- NA; x}

#Apply the function to a single variable:
mydata$q1 <- my9isNA(q1)
print(mydata)

# This assigns the missing values for a
# manually entered list of variables.
# The lapply function applies a function
# to every member of a list (a data frame is a type of list).
mydata[c("q1","q2","q3","q4")] <- lapply(
  mydata[c("q1","q2","q3","q4")],
  my9isNA)
print(mydata)

# This does contiguous variables without naming them all.
# If you know their column numbers, use this approach.
# q1 is the 3rd column and q4 is the 6th, so use 3:6.
mydata[ 3:6 ] <- lapply( mydata[ 3:6 ], my9isNA)
mydata

# Here is a way to handle many contiguous variables
# by name rather than by column number.
# We use the which function to find out the column
```

	`# number of each variable name.`
	`A <- which( names(mydata)=="q1" )`
	`Z <- which( names(mydata)=="q4" )`
	`mydata[ A:Z ] <- lapply( mydata[ A:Z ], my9isNA)`
	`mydata`

MULTIPLE CONDITIONAL TRANSFORMATIONS

Conditional transformations apply different formulas to different subsets of the data. For example, the formulas for recommended daily allowances of vitamins differ for males and females. If you have only one formula to apply to each subgroup, read the section above on conditional transformations. The example below shows how to handle a set of formulas for each subgroup.

Note that I do not use `attach(mydata)` which would allow me to refer to variables by their short name, e.g. gender. That is because it is possible to create a variable outside of a data frame. When creating variables, it is best to stick with the form of the name that includes the data frame, e.g. `mydata$gender`.

The R program below creates two new (rather silly) scores, score1 and score2. It does it using a logical vector (variable) that is TRUE for the observations we want the formulas to apply to and FALSE otherwise. The section ***Selecting Observations*** has much more information on logical vectors.

SAS	`* SAS Program for Multiple Conditional Transformations;`
	`DATA SASUSER.mydata;`
	`SET  SASUSER.mydata;`
	`IF gender="m" THEN DO;`
	`  score1 = (1.1*q1)+q2;`
	`  score2 = (1.2*q1)+q2;`
	`END;`
	`ELSE IF gender="f" THEN DO;`
	`  score1 = (2.1*q1)+q2;`
	`  score2 = (2.2*q1)+q2;`
	`END;`
	`RUN;`

SPSS	`* SPSS Program for Multiple Conditional Transformations.`
	`DO IF (gender EQ 'm').`
	`+  COMPUTE score1 = (2*q1)+q2.`
	`+  COMPUTE score2 = (3*q1)+q2.`
	`ELSE IF (gender EQ 'f').`
	`+  COMPUTE  score1 = (20*q1)+q2.`
	`+  COMPUTE  score2 = (30*q1)+q2.`
	`END IF.`

	EXECUTE.

R	# R Program for Multiple Conditional Transformations.
	# Read the file into a data frame and print it.
	load(file="c:\\mydata.Rdata")
	print(mydata)
	
	# Use column bind to add two new columns to mydata.
	# Not necessary for this example, but handy to know.
	mydata <- cbind(mydata, score1 = 0, score2 = 0)
	
	attach(mydata)
	mydata$guys <- gender=="m" #Creates a logical vector for males.
	mydata$gals <- gender=="f" #Creates another for females.
	print(mydata)
	
	# Applies basic math to the group selected by the logic.
	mydata$score1[gals]<- 2*q1[gals] + q2[gals]
	mydata$score2[gals]<- 3*q1[gals] + q2[gals]
	mydata$score1[guys]<-20*q1[guys] + q2[guys]
	mydata$score2[guys]<-30*q1[guys] + q2[guys]
	print(mydata)
	
	# This step uses NULL to delete the guys & gals vars.
	mydata$gals <- mydata$guys <- NULL
	print(mydata)

RENAMING VARIABLES (…AND OBSERVATIONS)

In SAS and SPSS, you are never aware of where the variable names are stored or how. You just know they are in the data set somewhere. Renaming is simply a matter of matching the new name to the old name. In R however, both row and column names are stored in character vectors within the data frame. In essence, they are just another form of variable that you can manipulate.

You can see the names in the data editor, and changing them there by hand is a very easy way to rename them. The `rename` function that comes with the `reshape` package allows you to change names without knowing about the `name` and `row.name` vectors and so is also quite easy to use.

Those approaches are demonstrated first in the examples below. I then use several other approaches to rename variables that make it much more clear how R stores the names and how to change them.

SAS	* SAS Program for Renaming Variables; DATA SASUSER.mydata; RENAME q1-q4=x1-x4;

	*or; *RENAME q1=x1 q2=x2 q3=x3 q4=x4; RUN;
SPSS	* SPSS Program for Renaming Variables. GET FILE='C:\mydata.sav'. RENAME VARIABLES (Q1=X1)(Q2=X2)(Q3=X3)(Q4=X4). EXECUTE.

R	# R Program for Renaming Variables.
	load(file="c:\\mydata.Rdata")
	print(mydata)
	
	#---This uses the data editor.
	#Make the changes by clicking on the names in the spreadsheet, then closing it.
	fix(mydata)
	print(mydata)
	
	# Restore original names for next example.
	names(mydata)<-c("group", "gender", "q1", "q2", "q3", "q4")
	
	#---This method is most like other languages such as SPSS or SAS.
	# It's easy to understand but doesn't help you understand what
	# R is actually doing. It requires the reshape library.
	library(reshape)
	mydata <- rename(mydata, c(q1="x1")) #Note the new name is one in quotes.
	mydata <- rename(mydata, c(q2="x2"))
	mydata <- rename(mydata, c(q3="x3"))
	mydata <- rename(mydata, c(q4="x4"))
	print(mydata)
	
	# Restore original names for next example.
	names(mydata)<-c("group", "gender", "q1", "q2", "q3", "q4")
	
	#---Simplest renaming with no packages required.
	# With this approach, you simply list every name in order,
	# even those you don't need to change.
	
	names(mydata) <- c("group", "gender", "x1", "x2", "x3", "x4")
	print(mydata)
	
	# Restore original names for next example.
	names(mydata)<-c("group", "gender", "q1", "q2", "q3", "q4")
	
	#---The edit function actually generates a list of names for you.
	# If you edit them and close the window, they will change.
	names(mydata) <- edit(names(mydata))
	print(mydata)

```
# Restore original names for next example.
names(mydata)<-c("workshop", "gender", "q1", "q2", "q3", "q4")

#---This method uses the row index numbers.
# First, extract the names to work on.
mynames<-names(mydata)

# Now print the names. Data.frame adds index numbers.
print(data.frame(mynames))
mynames[3]<-"q1"
mynames[4]<-"q2"
mynames[5]<-"q3"
mynames[6]<-"q4"
names(mydata)<-mynames #Put the new names back in as the column
names.
print(mydata)

# Restore original names for next example.
names(mydata)<-c("group", "gender", "q1", "q2", "q3", "q4")

#---Here's the exact same example, but now you don't need to know
# the order of each variable i.e. that V1 is column [2]

mynames<-names(mydata) #Make a copy to work on
print(mynames)

mynames[ mynames=="q1" ] <-"x1"
mynames[ mynames=="q2" ] <-"x2"
mynames[ mynames=="q3" ] <-"x3"
mynames[ mynames=="q4" ] <-"x4"
print(mynames)

# Finally replace the names with the new ones.
names(mydata) <- mynames
print(mydata)

# Restore original names for next example.
names(mydata)<-c("group", "gender", "q1", "q2", "q3", "q4")


#---You can do the steps above without working on a copy,
# but I find it VERY confusing to read.
names(mydata)[names(mydata)=="q1"] <-"x1"
names(mydata)[names(mydata)=="q2"] <-"x2"
names(mydata)[names(mydata)=="q3"] <-"x3"
names(mydata)[names(mydata)=="q4"] <-"x4"
print(mydata)
```

```
# Restore original names for next example.
names(mydata)<-c("group", "gender", "q1", "q2", "q3", "q4")


#---This approach works well for lots of numbered
# variable names names like x1,x2...
# First we'll see what the names look like.
print( names(mydata) )

# Next we'll generate x1,x2,x3,x4 to replace q1,q2,q3,q4.
myXs <-paste( "x", 1:4, sep="")
print(myXs)

# Now we want to find out where to put the new names.
# We can see they are the 3rd thru 6th positions, but
# let's assume we want R to find that out for us.
# This finds the index number for q1 and q4.
myA <- which( names(mydata)=="q1" )
print(myA)
myZ <- which( names(mydata)=="q4" )
print(myZ)

# Replace q1 thru q4 at index values A thru Z with
# the character vector of new names.
names( mydata[ ,myA:myZ] ) <- myXs
print(mydata)

#remove the unneeded objects.
rm(myXs,myA,myZ)
```

RECODING VARIABLES

Recoding is just a simpler way of doing a set of similar IF/THEN conditional transformations. It is often used in survey research to collapse 5-point Likert scale items to simpler 3-point Disagree/Neutral/Agree scales to summarize results. It is also often used to reverse the scale of negatively worded items so that a large numeric value has the same meaning across all items. Scale reversals are more easily done by subtracting each score from 6 as in q1r=6-q1. That results in 6-5=1, 6-1=5 and so on. The example below just collapses the scale.

The recode function used is in the cars library, which you'll have to install before running this. See ***Installing Add-On Packages*** for details.

SAS doesn't have a separate recode facility but it does offers a similar capability using its value label formats. That has the useful feature of applying the formats in categorical analyses and ignoring it otherwise. For example, PROC FREQ will use the format but PROC MEANS will ignore

it. You can also recode the data with a series of IF/THEN statements. Both methods are shown below. For simplicity, I leave the value labels out of the SPSS and R programs. Those are demonstrated in the section ***Value Labels or Formats (& Measurement Level).***

For recoding continuous variables into categorical, see the cut2 function in the Hmisc library. For choosing optimal cut points with regard to a target variable, see the rpart function or the tree function in Hmisc.

SAS

```
* SAS Program for Recoding Variables;
DATA SASUSER.mydata;
INFILE 'c:\mydata.csv' delimiter = ','
   MISSOVER DSD LRECL=32767 firstobs=2 ;
INPUT id workshop gender $ q1 q2 q3 q4;
PROC PRINT; RUN;;
PROC FORMAT;
  VALUE Agreement 1="Disagree" 2="Disagree"
                  3="Neutral"
                  4="Agree"    5="Agree"; run;
DATA SASUSER.mydata;
  SET SASUSER.mydata;
  ARRAY q q1-q4;
  ARRAY qr qr1-qr4; *r for recoded;
  DO i=1 to 4;
     qr{i}=q{i};
     if q{i}=1 then qr{i}=2;
     else
     if q{i}=5 then qr{i}=4;
  END;
  FORMAT q1-q4 q1-q4 Agreement.;
RUN;

* This will use the recoded formats automatically;
PROC FREQ; TABLES q1-q4; RUN;

* This will ignore the formats;
* Note high/low values are 1/5;
PROC UNIVARIATE; VAR q1-q4; RUN;

* This will use the 1-3 codings, not a good idea!;
* High/Low values are now 2/4;
PROC UNIVARIATE; VAR qr1-qr4;
RUN;
```

SPSS

```
* SPSS Program for Recoding Variables.
GET FILE='C:\mydata.sav'.
RECODE q1 to q4 (1=2) (5=4).
SAVE OUTFILE='C:\myleft.sav'.
```

	EXECUTE .

R	# R Program for Recoding Variables.
	load(file="c:\\mydata.Rdata")
	print(mydata)
	attach(mydata)
	
	library(cars)
	mydata$q1<-recode(q1,"1=2;5=4")
	mydata$q2<-recode(q2,"1=2;5=4")
	mydata$q3<-recode(q3,"1=2;5=4")
	mydata$q4<-recode(q4,"1=2;5=4")
	mydata
	
	# This does the same thing but for large sets
	# of variables.
	# Generate two sets of var names to use.
	myQnames <- paste("q", 1:4, sep="")
	myQRnames <- paste("qr", 1:4, sep="")
	print(myQnames) #The original names.
	print(myQRnames) #The names for the recoded variables.
	
	# Extract the q variables to a separate data frame.
	myQRvars <- mydata[,myQnames]
	print(myQRvars)
	
	# Rename all the variables with F for Factor.
	names(myQRvars) <- myQRnames
	print(myQRvars)
	
	# Create a function to apply the labels to lots of variables.
	myRecoder <- function(x) { recode(x,"1=2;5=4") }
	
	# Here's how to use the function on one variable.
	print(myRecoder(myQRvars$qr1))
	
	#Apply it to all the variables. This method works.
	myQRvars[,myQRnames] <- lapply(myQRvars[,myQRnames],
	myRecoder)
	print(myQRvars)
	
	# Save it back to mydata if you want.
	mydata <- cbind(mydata,myQRvars)
	print(mydata)
	summary(mydata)

KEEPING AND DROPPING VARIABLES

In SAS you can use the KEEP and DROP statements to determine which variables to save in your data set. The SPSS equivalent is the DELETE VARIABLES statement. In R, the methods discussed in the Selecting Variables section perform this function as well. One additional feature in R is the NULL object, which you can use to delete variables in data frames without making new versions of the data. To use it simply apply it in any valid assignment such as:
`mydata$varname<-NULL.`

Note that the remove function deletes only whole objects, so you will get an error message if you try:
`rm(mydata$varname)  #This does NOT work.`

The example below keeps the variables only the left side of our data frame. In the later example on joining files, we'll do the same for the other variables then join the two back together.

SAS	`* SAS Program for Keeping and Dropping Variables;`
	`DATA myleft; SET mydata;`
	`  KEEP id workshop gender q1 q2;`
	`RUN;`
	
	`*or equivalently;`
	`DATA myleft; SET mydata;`
	`  DROP q3 q4;`
	`RUN;`

SPSS	`* SPSS Program for Keeping and Dropping Variables.`
	`GET FILE='C:\mydata.sav'.`
	`DELETE VARIABLES q3 to q4.`
	`SAVE OUTFILE='C:\myleft.sav'.`
	`EXECUTE.`

R	`* R Program for Keeping and Dropping Variables.`
	`load(file="c:\\mydata.Rdata")`
	`# Copy the data so we don't destroy the original.`
	`mysubset<-mydata`
	
	`# Then drop q3 and q4 using NULL.`
	
	`mysubset$q3 <- mysubset$q4 <- NULL`
	
	`print(mysubset)`

BY OR SPLIT FILE PROCESSING

When you want to repeat an analysis for every level of a variable, you can use the BY statement in SAS or SPLIT FILE command in SPSS. R does this with the `"by"` function. It has three required

parameters, the data frame name, the factor variable(s) to split on and the analytical function. After those parameters are supplied, any additional parameter settings are passed to the analytical function. The examples below use the summary function to get basic stats by gender and then by both gender and workshop.

SAS and SPSS both require you to sort the data by the factor variable(s), but R does not.

SAS

```
* SAS Program for By or Split File Processing;
PROC SORT DATA=SASUSER.mydata;
  BY gender;
PROC MEANS DATA=SASUSER.mydata;
  BY gender;
```

SPSS

```
* SPSS Program for By or Split File Processing;
GET FILE="C:\mydata.sav".
SORT CASES BY gender .
SPLIT FILE
  SEPARATE BY gender .
DESCRIPTIVES
  VARIABLES=q1 q2 q3 q4
  /STATISTICS=MEAN STDDEV MIN MAX .
```

R

```
# R Program for By or Split File Processing.
load(file="c:\\mydata.Rdata")
print(mydata)
attach(mydata) #Makes this the default dataset.

# Get summary stats of observations and all variables.
summary(mydata)

# Get summary stats for each value of gender
# for all variables.
by(mydata, gender, summary)

# Get summary stats for each value of gender,
# for only the variables chosen by column name.
by( mydata[c("q1","q2","q3","q4")] , gender, summary)

# Multiple categorical variables must be used in a list.
# The data.frame function will get them there.
# Data need not be sorted by workshop and gender.
by(mydata[c("q1","q2","q3","q4")],
  data.frame(workshop,gender), summary)

# This can seem much simpler by breaking it into pieces.
myVars <- c("q1","q2","q3","q4")
myBys <- data.frame(workshop,gender)
by( mydata[myVars], myBys, summary)
```

STACKING / CONCATENATING / ADDING DATA SETS

The examples below first split mydata into separate data sets for males and females. Then it shows how to put them back together. SAS calls this concatenation, SPSS calls it adding files and R, with its row/column orientation calls it binding rows.

SAS

```
* SAS Program for Stacking/Concatenating/Adding Data Sets;
DATA males;   SET mydata; WHERE gender=1; RUN;
DATA females; SET mydata; WHERE gender=0; RUN;
*Put them back together again;
DATA both;
  SET males females;
RUN;
```

SPSS

```
* SPSS Program for Stacking/Concatenating/Adding Data Sets.
GET FILE='C:\mydata.sav'.
SELECT IF(gender = "f").
SAVE OUTFILE='C:\females.sav'.
EXECUTE .

GET FILE='C:\mydata.sav'.
SELECT IF(gender = "m").
SAVE OUTFILE='C:\males.sav'.
EXECUTE .

GET FILE='C:\females.sav'.
ADD FILES /FILE=*
 /FILE='C:\males.sav'.
EXECUTE.
```

R

```
# R Program for Stacking/Concatenating/Adding Data Sets.
load(file="c:\\mydata.Rdata")print(mydata)
attach(mydata)

#Put only males in a data frame.
males <- mydata[gender=="m", ]
print(males)

#Put only females in another data frame.
females <- mydata[gender=="f", ]
print(females)

#Bind their rows together with the rbind function.
both<-rbind(females,males)
print(both)
```

JOINING / MERGING DATA FRAMES

One of the most frequently used data manipulation methods is joining or merging two data sets. If you have a one-to-many join, it will create a row for every possible match. A common example

is a short data frame containing household-level information such as family income joined to a longer data set of individual family member variables. A complete record of each family member along with their household income will result. Duplicates in more than one data frame are possible, but should be studied carefully for errors.

In the example below, builds on the keeping/dropping variables example above. We'll start with mydata, make two copies (left and right) containing different variables and then join them back together to recreate the original file.

SAS

```
* SAS Program for Joining/Merging Data Sets.
DATA myleft; SET mydata; KEEP id workshop gender q1 q2;
PROC SORT; BY id workshop; RUN;

DATA myright; SET mydata; KEEP id q3 q4;
PROC SORT; BY id workshop; RUN;

DATA both; MERGE myleft myright; BY id workshop; RUN;
```

SPSS

```
* SPSS Program for Joining/Merging Data Sets.
GET FILE='C:\mydata.sav'.
DELETE VARIABLES q3 to q4.
SAVE OUTFILE='C:\myleft.sav'.
EXECUTE .

GET FILE='C:\mydata.sav'.
DELETE VARIABLES workshop to q2.
SAVE OUTFILE='C:\myright.sav'.
EXECUTE .

GET FILE='C:\myleft.sav'.
MATCH FILES /FILE=*
 /FILE='C:\myright.sav'
 /BY id.
EXECUTE.
```

R

```
# R Program for Joining/Merging Data Sets.

#Note that row.names="id" is not used when reading
# the table below. That is because we need to match
# on ID so we keep it as a variable.
mydata<-read.table("c:\\mydata.csv",header=TRUE,sep=",")
print(mydata)

#Create a data frame keeping the left two q variables.
myleft<-mydata[ c("id","workshop","gender","q1","q2") ]
print(myleft)

#Create a data frame keeping the right two q variables.
myright<-mydata[ c("id","workshop","q3","q4") ]
```

```
print(myright)

#Merge the two dataframes by ID.
#Since "workshop" is in both, and is not used
# to merge the dataframes, R will save both
# and name them workshop.x and workshop.y
# Don't save it in both to avoid this.
both<-merge(myleft,myright,by="id")
print(both)

#Merge dataframes by both ID and workshop.
#Since there are no unsed variables, it recreates
#the original dataframe perfectly.
both<-merge(myleft,myright,by=c("id","workshop"))
print(both)

#Merge dataframes by both ID and workshop,
#while allowing them to have different names in each dataframe.
#They're the same here, but they could be different.
both<-merge(myleft,myright,by.x=c("id","workshop"),
                           by.y=c("id","workshop"))
print(both)
```

AGGREGATING OR SUMMARIZING DATA

We often have to work on data that is a summarization of other data. For example, we might work on family data that was collapsed from family member data. Below we will create a file containing the mean of the q1 variable by gender and then by workshop and gender. We print each data frame for clarity. The last example merges the summarized data back to our original file and so builds upon the previous examples in the section, ***Joining / Merging Data Frames***.

A much more extensive aggregation approach is available in Hadley Wickham's delightful reshape package documented at http://cran.r-project.org/doc/packages/reshape.pdf.

SAS

```
* SAS Program for Aggregating/Summarizing Data;

* Get means of q1 for each gender;
PROC SUMMARY DATA=SASUSER.mydata MEAN NWAY;
  CLASS GENDER;
  VAR q1;
  OUTPUT OUT=SASUSER.myAgg;
  RUN;
PROC PRINT; RUN;
DATA SASUSER.myAgg;
  SET SASUSER.myAgg;
  WHERE _STAT_='MEAN';
```

	KEEP gender q1; RUN; PROC PRINT; RUN; *Get means of q1 by workshop and gender; PROC SUMMARY DATA=SASUSER.mydata MEAN NWAY; CLASS WORKSHOP GENDER; VAR Q1; OUTPUT OUT=SASUSER.myAgg;RUN; PROC PRINT; RUN; *Strip out just the mean and matching variables; DATA SASUSER.myAgg; SET SASUSER.myAgg; WHERE _STAT_='MEAN'; KEEP workshop gender q1; RENAME q1=meanQ1; RUN; PROC PRINT; RUN; *Now merge aggregated data back into mydata; PROC SORT DATA=SASUSER.mydata; BY workshop gender; RUN: PROC SORT DATA=SASUSER.myAgg; BY workshop gender; RUN: DATA SASUSER.mydata2; MERGE SASUSER.mydata SASUSER.myAgg; BY workshop gender; PROC PRINT; RUN;
SPSS	* SPSS Program for Aggregating/Summarizing Data. * Get mean of q1 by gender. GET FILE='C:\mydata.sav'. AGGREGATE /OUTFILE='C:\myAgg.sav' /BREAK=gender /q1_mean = MEAN(q1). GET FILE='C:\myAgg.sav'. LIST. EXECUTE. * Get mean of q1 by workshop and gender. GET FILE='C:\mydata.sav'. AGGREGATE /OUTFILE='C:\myAgg.sav' /BREAK=workshop gender /q1_mean = MEAN(q1). GET FILE='C:\myAgg.sav'. LIST.

	EXECUTE. * Merge aggregated data back into mydata. GET FILE='C:\mydata.sav'. SORT CASES BY workshop (A) gender (A) . MATCH FILES /FILE=* /TABLE='C:\myAgg.sav' /BY workshop gender. SAVE OUTFILE='C:\mydata.sav'. EXECUTE.

R	# R Program for Aggregating/Summarizing Data.
	load(file="c:\\mydata.Rdata")
	print(mydata)
	attach(mydata)
	* Load packages we need. Must have installed beforehand.
	library(Hmisc)
	library(reshape)
	
	# R's built-in function is aggregate.
	# It creates new names for the variables.
	# Note gender must be enclosed in the list function,
	# even though it is a single object.
	# First just gender.
	myAgg<-aggregate(q1, by=list(gender), FUN=mean, na.rm=TRUE)
	print(myAgg)
	# Now workshop and gender.
	myAgg<-aggregate(q1, by=list(workshop,gender), FUN=mean,
	na.rm=TRUE)
	print(myAgg)
	
	# The summarize function from the Hmisc library
	# is much nicer. It keeps the original variable names
	# and labels too, if any exist.
	library(Hmisc)
	# First just gender.
	myAgg<-summarize(q1, by=gender, FUN=mean, na.rm=TRUE)
	print(myAgg)
	# Now workshop and gender.
	myAgg<-summarize(q1, by=llist(workshop,gender), FUN=mean,
	na.rm=TRUE)
	print(myAgg)
	
	# This merges the aggregated values back to mydata.
	# First we rename q1 to mean.q1.
	myAgg<-rename(myAgg, c(q1="mean.q1"))
	print(myAgg)
	# Now merge it back with mydata.
	mydata2<-merge(mydata,myAgg,by=c("workshop","gender"))

	`print(mydata2)`

Reshaping Variables to Observations and Back

A common data management problem is reshaping data from “wide” format to “long” and back. If we assume our variables q1,q2,q3,q4 are the same item measured at four times, this is the standard wide format for repeated measures data. Converting this to the long format consists of writing out four records, each of which has just one measure, we'll call it Y, and a counter variable, often called time, that goes 1,2,3,4. So in the simplest case, two variables will replace as many as there are repeats through time.

Going from wide to long is just the reverse. SPSS makes this process very easy to do with their ***Restructure Data Wizard***. It actually generated the SPSS program below. The SAS approach is quite complex and takes a bit of study. Hadley Wickham's excellent `reshape` package in R is quite powerful and easy to use. It uses the analogy of melting your data so that you can cast it into a different mold. In addition to reshaping, the package makes quick work of a wide range of aggregation problems.

SAS

```
* SAS Program to Reshape Data.
* First go from "wide" to "long" format;
data SASUSER.mydata;
infile 'c:\mydata.csv' delimiter = ','
   MISSOVER DSD lrecl=32767 firstobs=2 ;
input id workshop gender $ q1 q2 q3 q4;
run;

DATA SASUSER.mylong;
SET  SASUSER.mydata;
  ARRAY q{4} q1-q4;
  DO i=1 to 4;
     y=q{i};
     question=i;
     output;
  END;
KEEP id workshop gender question y;
PROC PRINT; RUN;;

PROC SORT DATA=SASUSER.mylong;
  BY id question;
  RUN;

* Now go from "long" back to "wide";
DATA SASUSER.mywide;
SET  SASUSER.mylong;
  BY id;
RETAIN q1-q4;
```

	ARRAY q{4} q1-q4;
	IF FIRST.id THEN DO i=1 to 4;
	  q{i}=.;
	  q{i}=y;
	END;
	IF LAST.id THEN OUTPUT;
	DROP question y i;
	PROC PRINT; RUN;

SPSS	* SPSS Program to Reshape Data. * Going from our "wide" format to "long". GET FILE='C:\mydata.sav'. VARSTOCASES /MAKE Y FROM q1 q2 q3 q4 /INDEX = Question(4) /KEEP = id workshop gender /NULL = KEEP. SAVE OUTFILE='C:\mywide.sav'. EXECUTE. * Going from our "long" format to "wide". GET FILE='C:\mywide.sav'. CASESTOVARS /ID = id workshop gender /INDEX = Question /GROUPBY = VARIABLE. SAVE OUTFILE='C:\mylong.sav'. EXECUTE.

R	# R Program to Reshape Data.
	load(file="c:\\mydata.Rdata")
	print(mydata)
	
	# We need an ID variable for this exercise.
	# We can extract it from rownames with this.
	mydata$subject <- as.numeric(rownames(mydata))
	# Or we could generate it from scratch like this.
	mydata$subject <- 1:8
	print(mydata)
	
	library(reshape)
	mylong<-melt(mydata,id=c("subject","workshop","gender"))
	print(mylong)
	
	# By leaving "value" out, it becomes the measure
	# for variable.
	mywide<-cast(mylong, subject+workshop+gender~variable)
	print(mywide)

SORTING DATA FRAMES

Sorting is one of the areas that R differs most from SAS and SPSS. It does not directly sort a data frame. Instead, it determines the order of the sorted rows and then applies them to do the sort.

Consider the names Ann, Eve, Cary, Dave, Bob. They are almost sorted in ascending order. Since the number of names is small, it is easy to determine the order that the names would require to be sorted. We need the 1st name, Ann, followed by the 5th name, Bob, followed by the 3rd name, Cary, the 4th name, Dave and finally the 2nd name, Eve. The `order` function would get those index values for us: 1 5 3 4 2.

One way to select rows from a data frame is to use the form `mydata[rows,columns]`. If you leave them all out, as in `mydata[ , ]` then you'll get all rows and all columns. You can select some rows as we have done elsewhere to select the females in the first 4 records with `mydata[ c(1,2,3,4), ]`. We can select them in reverse order with `mydata[ c(4,3,2,1), ].`

If we applied that idea to the indexes in our name example, we could get `mydata[ c(1,5,3,4,2) , ]` to print (or save) them in order. Since the `order` function determines the indexes of the sorted order automatically, we could do the same thing with `mydata [ order(name), ]`.

SAS	`* SAS Program to Sort Data;`
	`PROC SORT  DATA=SASUSER.mydata; BY workshop; RUN;`
	`PROC PRINT DATA=SASUSER.mydata; RUN;`
	`PROC SORT  DATA=SASUSER.mydata; BY gender workshop; RUN;`
	`PROC PRINT DATA=SASUSER.mydata; RUN;`
	`PROC SORT  DATA=SASUSER.mydata;`
	`  BY workshop descending gender; RUN;`
	`PROC PRINT DATA=SASUSER.mydata; RUN;`

SPSS	`* SPSS Program to Sort Data.`
	`SORT CASES BY workshop (A).`
	`LIST.`
	`EXECUTE.`
	`SORT CASES BY gender (A) workshop (A).`
	`LIST.`
	`EXECUTE.`
	`SORT CASES BY workshop (D) gender (A).`
	`LIST.`
	`EXECUTE.`

R	`# R Program to Sort Data.`
	`# Load our data into the workspace.`
	`load(file="c:\\mydata.Rdata")`
	`print(mydata)`
	
	`# Simply print the first four records.`

```
print( mydata[ c(1,2,3,4), ] )

# Print them again in reverse order by
# entering the index values backwards.
print( mydata[ c(4,3,2,1), ] )

# Sort the data by workshop.
# The order function will find the indexes that will sort.
mydataSorted<-mydata[ order(mydata$workshop), ]
print(mydataSorted)

# Now sort by gender then workshop:
mydataSorted<-mydata[ order( mydata$gender, mydata$workshop ), ]
print(mydataSorted)

# The default sort order is ascending. Prefix any variable
# with a minus sign to get descending order.
mydataSorted<- mydata[ order( -mydata$workshop,mydata$gender ), ]
print(mydataSorted)
```

VALUE LABELS OR FORMATS (& MEASUREMENT LEVEL)

This section blends two topics because in R they are inseparable. In both SAS and SPSS assigning labels to values is independent of the variable's measurement scale. In R, value labels can only be assigned to variables whose level is ***factor***.

In SAS a variable's measurement level of nominal, ordinal or interval is not set in advance. Instead, listing the variable on a specific statement such as BY or CLASS will tell SAS that you wish to view it as categorical.

SAS' use of value labels is handled in a two-step process. First, Proc Format creates a "format" for every unique set of labels and then the Format statement assigns a format to each variable or set of variables (see example below). The formats are stored outside the data set in a format library.

In SPSS, the VARIABLE LEVEL statement sets measurement level but this is merely a convenience to help the graphical user interface work well. So SPSS will not, for example, show you nominal variables in the DESCRIPTIVES procedure's dialog box, but if you enter them into the programming language, it will accept them. As with SAS, special commands tell SPSS how you want to view the scale of the data. These include BY and GROUPS.

Independently, the VALUE LABEL command sets labels for each level and the labels are stored within the data set itself.

R has the measurement levels of ***factor*** for nominal data, ***ordered factor*** for ordinal data and ***numeric*** for interval or scale data. You set these in advance and then the statistical and graphical procedures use them in the appropriate way automatically.

In our example text file, data gender was entered as “m” and “f” so R assigns the values assigned 2 and 1 since f precedes m in the alphabet. For character data, those defaults are often sufficient. However you can use factor() to change either. The values assigned follow the order on the levels argument so below with “m” coming first, it would be associated with 1 and “f” with 2. The labels argument follows the order of the levels. This example sets “m” as 1, “f” as 2 and uses the fully written out labels.

```
mydata$genderF <- factor( mydata$gender,
  levels=c("m","f"),labels=c("Male","Female") )
```

There is a danger in setting variable labels with character data that does not appear at all in SAS or SPSS. In the statement above, if we set `levels=c("m","F")` instead, the females would all be set to missing (NA) because the actual values are lower case. There are no capital F’s in the data! This danger applies of course to other more obvious misspellings.

Of course R has no way to automatically identify numeric factors, such as our workshop variable. It too is a categorical measure, but initially R assumes it is numeric because it is entered as 1 and 2. The factor function converts it to a factor and lets you optionally assign labels. Factor labels in R are stored in the data frame itself. We can convert it to a factor with the command,

```
mydata$workshop <- factor( mydata$workshop,
  levels=c(1,2,3,4),
  labels=c("R","SAS","SPSS","Stata") )
```

Note that our data only contain the values 1 and 2 but that is fine. You can also convert a numeric variable to a factor without assigning labels using just `mydata$workshop<-factor(mydata$workshop).`

R’s approach saves you work with character data and is no more work with numeric data than SAS or SPSS. R never needs a character variable converted to numeric form to work with certain functions as SPSS does.

But there is a downside to R's approach. What if you want to view variables both ways? For example, survey researchers usually want to get frequencies on their Likert scales that go from 1 to 5 and they want the mean score as well. They often also want to average them together to make new scores. The Hmisc package’s `describe` function provides both frequencies and means.

Many other functions, such as summary, require that you convert the variable from factor to numeric (or vice versa) to get it to do what you want. Several conversion functions help with this

problem: `as.factor`, `as.character` and `as.numeric`. For example, we can get `summary` to get frequencies rather than means, etc. by using `summary(as.factor(q1))`. If q1 were converted to a factor already and we wanted summary to get means, it requires two conversions. The first, `as.character`, extracts the original values that had been stored in character from. The second converts the character values of "1", "2","3","4","5" to the numeric ones, 1,2,3,4,5: `summary( as.numeric( as.character(q1) ) ).`

The examples below demonstrate a variety of approaches for dealing with factors and their labels. One example uses the Hmisc package, so if you haven't installed it, follow the directions under ***Installing Add-on Packages.***

SAS	`* SAS Program to Assign Value Labels (formats);`
	`PROC FORMAT;`
	`VALUES workshop_f 1="Control" 2="Treatment"`
	`VALUES $gender_f "m"="Male" "f"="Female";`
	`VALUES agreement`
	`1='Strongly Disagree'`
	`2='Disagree'`
	`3='Neutral'`
	`4='Agree'`
	`5='Strongly Agree'.;`
	`DATA SASUSER.mydata; SET SASUSER.mydata;`
	`FORMAT workshop workshop_f. gender gender_f.`
	`q1-q4 agreement.;`

SPSS	`* SPSS Program to Assign Value Labels.`
	
	`GET FILE="c:\mydata.sav".`
	`VARIABLE LEVEL workshop (NOMINAL)`
	`/q1 TO q4 (SCALE).`
	`VALUE LABELS workshop 1 'Control' 2 'Treatment'`
	`/q1 TO q4`
	`1 'Strongly Disagree'`
	`2 'Disagree'`
	`3 'Neutral'`
	`4 'Agree'`
	`5 'Strongly Agree'.`
	`SAVE OUTFILE="C:\mydata.sav".`

R	`# R Program to Assign Value Labels & Factor Status.`
	
	`# By default, group was read in as numeric and gender as factor.`
	`# That is because gender is character data.`
	`load(file="c:\\mydata.Rdata")`
	`attach(mydata)`
	`print(mydata)`
	
	`# Note that summary will treat group as numeric by default,`

```
# but it assumes gender is a factor and just counts its
# levels. It erroneously reports the mean workshop for now.
summary(mydata)

# Now change workshop into a factor:
mydata$workshop <- factor( mydata$workshop,
  levels=c(1,2,3,4),
  labels=c("R","SAS","SPSS","Stata") )
# Print data to see not all levels need be present.
print(mydata)

# Now see that summary only counts workshop attendance.
summary(mydata)

# Use describe function from Hmisc package.
# Unlike summary, it gets both frequencies and means
# on the q variables. It also calculates percents.
# You must have installed the Hmisc library first.
library(Hmisc)
describe(mydata)

# Show how levels are matched to values:
unclass(mydata$gender)

# Now change the order so m is 1 and f is 2.
# Note that if the values had been capitalized
# this would actually generate missing values!
mydata$genderF <- factor( mydata$gender,
  levels=c("m","f"),labels=c("m","f") )

# Print gender and gender Flipped to see that they match.
print( mydata[ ,c("gender","genderF")] )

# Extract the underlying values of each.
mydata$genderNums  <- as.numeric(mydata$gender)
mydata$genderFNums <- as.numeric(mydata$genderF)
# See the numbers are indeed flipped.
print(mydata)

# Create copies of q variables to use as factors
# so we can count them.
# Store levels to use repeatedly.
myQlevels <- c(1,2,3,4,5)
# Store labels to use repeatedly.
myQlabels <- c("Strongly Disagree",
               "Disagree",
               "Neutral",
               "Agree",
```

```
                "Strongly Agree")
# Now create a new set of variables as factors.
mydata$q1f <- factor(q1, myQlevels, myQlabels)
mydata$q2f <- factor(q2, myQlevels, myQlabels)
mydata$q3f <- factor(q3, myQlevels, myQlabels)
mydata$q4f <- factor(q4, myQlevels, myQlabels)

# Get summary and see that workshops are now counted.
summary( mydata[ c("q1f","q2f","q3f","q4f") ] )

#---This approach makes copies of the q variables for use
# as factors, but now using a more automated approach
# that is easier if you have lots of variables.

# Create copies of q variables to use as factors
# so we can count them (repetitive for clarity).
myQlevels <- c(1,2,3,4,5)
myQlabels <- c("Strongly Disagree",
               "Disagree",
               "Neutral",
               "Agree",
               "Strongly Agree")
print(myQlevels)
print(myQlabels)

# Generate two sets of var names to use.
myQnames  <- paste( "q",  1:4, sep="")
myQFnames <- paste( "qf", 1:4, sep="")
print(myQnames) #The original names.
print(myQFnames)  #The names for new factor variables.

# Extract the q variables to a separate data frame.
myQFvars <- mydata[ ,myQnames]
print(myQFvars)

# Rename all the variables with F for Factor.
names(myQFvars) <- myQFnames
print(myQFvars)

# Create a function to apply the labels to lots of variables.
myLabeler <- function(x) { factor(x, myQlevels, myQlabels) }

# Here's how to use the function on one variable.
summary( myLabeler(myQFvars["qf1"]) )

# Apply it to all the variables.
myQFvars[ ,myQFnames] <-
  lapply( myQFvars[ ,myQFnames ], myLabeler )
```

	# Get summary again, this time with labels.
	summary(myQFvars)
	
	# You can even join the new variables to mydata.
	# (this gives us two labeled sets if you ran the example above too.)
	mydata<-cbind(mydata,myQFvars)
	print(mydata)

VARIABLE LABELS

Perhaps the most fundamental feature missing from the main R distribution is support for variable labels. It is a testament to its openness and flexibility that such a fundamental feature could be added on by user-written package. Frank Harrell's `Hmisc` package does just that. It adds an attribute to the dataset of "label" and stores the labels there, even converting them from SAS automatically (but not SPSS). As amazing as this addition is, the fact that variable labels were not included in the main distribution means that most procedures do not take advantage of what `Hmisc` adds. The many wonderful functions in the `Hmisc` package do of course.

A second approach to variable labels is to store them as a character variable. That is the approach used by the `prettyR` package.

Finally, you can use illegal variable names of any length by enclosing them in quotes. This has the advantage of working with all R functions. Unfortunately it also makes selecting variables by name much less practical. You can type out the long name or use a search function `grep` to find key words in your labels. You can of course still select variables using index numbers.

The R program below demonstrates all three approaches to variable labels.

SAS	* SAS Program for Variable Labels;
	DATA SASUSER.mydata; SET SASUSER.mydata ;
	LABEL
	  Q1="The instructor was well prepared"
	  Q2="The instructor communicated well"
	  Q3="The course materials were helpful"
	  Q4="Overall, I found this workshop useful";
	PROC FREQ; TABLES q1-q4; RUN;
	RUN;

SPSS	* SPSS Program for Variable Labels.
	VARIABLE LABELS
	  Q1 "The instructor was well prepared"
	  Q2 "The instructor communicated well"
	  Q3 "The course materials were helpful"
	  Q4 "Overall, I found this workshop useful".
	FREQUENCIES VARIABLES=q1 q2 q3 q4.
	EXECUTE.

R	# R Program for Variable Labels.
	
	load(file="c:\\mydata.Rdata")
	print(mydata)
	
	# First we'll use the Hmisc approach that is most
	# like SAS or SPSS.
	label(mydata$q1)<-"The instructor was well prepared."
	label(mydata$q2)<-"The instructor communicated well."
	label(mydata$q3)<-"The course materials were helpful."
	label(mydata$q4)<- "Overall, I found this workshop useful."
	# Note that the Hmisc describe function uses the labels.
	describe(mydata)
	# However, the built-in summary function ignores the labels.
	summary(mydata)
	
	#Assign long variable names to act as variable labels.
	names(mydata) <- c("Workshop","Gender",
	  "The instructor was well prepared.",
	  "The instructor communicated well.",
	  "The course materials were helpful.",
	  "Overall, I found this workshop useful.")
	names(mydata)
	
	# Assign long variable names to act as variable labels.
	# Note here that you must get all the name of mydata
	# and FROM THEM select 3:6.
	names(mydata)[3:6] <- c(
	  "The instructor was well prepared.",
	  "The instructor communicated well.",
	  "The course materials were helpful.",
	  "Overall, I found this workshop useful.")
	names(mydata)
	# Now summary and all R functions will use the long names.
	summary(mydata)
	
	# You can still select variables by name,
	# but only if you like to type!
	summary(mydata["Overall, I found this workshop useful."])
	
	# It is more realistic to select them using indexes.
	summary(mydata[3:6])
	
	# You can also use search for strings in column names using
	# the grep function.
	myvars<-grep('instructor',names(mydata))
	# This shows you that variables 3 and 4 contain 'instructor'.
	print(myvars)

```
summary ( mydata[myvars] )
```

WORKSPACE MANAGEMENT

The way R stores its data is very different from SAS and SPSS. While in use, the data is stored in the computer's random access memory rather than on a hard drive. This means that R cannot handle datasets as large as SAS or SPSS. 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 RAM. That is the current RAM limit in today's 32-bit version of Windows. You can use virtual memory to go up to 3 gigabytes but that slows things down considerably.

Operating systems capable of 64-bit memory spaces are becoming more popular. The huge amounts of memory they can handle mitigate this problem. One way around the limitation is to store your data in a relational database and use its facilities to generate a random sample to work with. Another alternative is to use S-PLUS, a commercial package that has an almost identical language with extensions to handle "big data".

After each R session, it offers to save the workspace. If you click yes, it stores it in a file named "***c:\Program Files\R\R 2.4.1\.Rdata***". The next time you start R it automatically loads the file back into memory and you can continue working.

The name .***Rdata*** isn't very easy to work with in Windows since it hides file extensions by default, and an extension is anything that follows a period. So the entire filename "***.Rdata***" is an extension and hidden from view! To get windows to show you this file, in Explorer uncheck the option below and click ***Apply to all folders***:
Tools>Folder Options>View>[ ]Hide extensions to known file types.

If you want to avoid using the name .Rdata, you can at any time choose Save Workspace from the File menu in R and name it anything you like with the extension .Rdata. Then when you start R, you will have to use Load Workspace to load it from the hard drive back into the computer's memory. You can also use R functions to save and load your workspace. To save your workspace, use the command `save.image(file="c:\\mydata.Rdata")`, where mydata is any filename you like. If you want to save only a subset of your workspace, you can use the command `save(mydata,file="c:\\mydata.Rdata").`
Save is one of the few functions that can accept many objects separated by commas, so might save three with save(mydata,myVars,myObs,file="c:\\myExamples.Rdata). Regardless of how you saved your workspace, the command to load that workspace back into memory is `load(file="mydata.Rdata").`

There is another way to keep your various projects organized if you like using shortcuts. You 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, upon 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. Finding the particular file you need via search engines or backup systems is hampered by this approach. If you accidentally moved an .Rdata file to another folder, you would not know what was in it without loading it into R.

You can see what objects are in your workspace with the `ls` function. To list all objects use `ls()`. If you want to see if just one object exists, you can enter `ls(mydata)`. To delete an object, use the remove function as in `rm(mydata)`. To remove all the objects in your workspace, combine the two functions as in `rm( list=ls() ).`

To help conserve valuable workspace size, you can use the `cleanup.import` function in the `Hmisc` package. It automatically stores the variables in their most compact form. You use it as `mydata<-cleanup.import(myata).` See ***Installing Add-on Packages*** for more details.

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

WORKSPACE MANAGEMENT FUNCTIONS

Store data efficiently (requires Hmisc)	`mydata<-cleanup.import(mydata)`
List object contents (requires Hmisc)	`contents(mydata)`
List all objects	`ls()`
List just mydata	`ls(mydata)`
Remove an object	`rm(mydata)`
Remove several	`rm(mydata,myvars,myobs)`
Remove all objects	`rm( list=ls() )`
Remove a variable from a data frame	`mydata$myvar <- NULL`
Save all objects	`save.image(file="c:\\mydata.Rdata")`

Save some	`save(mydata,x,y,z,file="c:\\myExamples.Rdata)`
Load workspace	`load(file="c:\\mydata.Rdata")`

GRAPHICS

R can make a very wide range of graphs, and its default settings are usually excellent. I will only demonstrate a few simple graphs and refer you to other sources for more information. There are also wonderfully comprehensive collections of graphs and the R programs to make them at the R Graph Gallery http://addictedtor.free.fr/graphiques/ and the Image Browser at http://bg9.imslab.co.jp/Rhelp/.

You can also use the demo capability in R to have it show you a sample of what it can do. Enter the commands below in the R console.

```
demo(graphics)
demo(persp)
library(lattice)
demo(lattice)
```

R does not have dynamic visualization capabilities like those in SAS/INSIGHT, but it does have a link to the excellent Ggobi package available for free at http://www.ggobi.org/. The `rggobi` package links the two and it is available at http://www.ggobi.org/rggobi/.

If you are interested in SPSS' extremely flexible Graphics Production Language (GPL), you will want to read about Hadley Wickham's `ggplot` package. Unfortunately the examples below do not yet include `ggplot`.

The examples below demonstrate a histogram, bar plot and scatter plot. While SAS and SPSS assume your data need summarizing and you use options to tell them when it is not, R assumes just the opposite. The function `barplot(c(40,60))` makes a chart with just those two bars. But the command `barplot(gender)` also assume you want to see every unsummarized value. You'll get 8 bars, one for every subject, with the height of the bars either 1 or 2 since those level codes were assigned by R (in alphabetical order) when it read the variable gender. So to get a plot that summarizes how many males and females there are includes the summary function `barplot( summary(gender) ).`

If we try to create a similar barplot for workshop, and workshop has not been converted to a factor, R says, `"Error in barplot.default(summary(workshop)) : 'height' must be a vector or a matrix."`

That is not a particularly useful message. If you have not already made workshop into a factor (see section on ***Value Labels or Formats (& Measurement Level)*** you can do so in the middle of the command: `barplot( summary( as.factor(workshop) ) )`

The command `barplot(gender)` will yield a bar of height 1 or 2 for every observation! Since the summary function counts factor levels (and gender is a factor) a bar plot showing the total number of males and females is done with `barplot( summary(gender) )`.

SAS

```
* Basic Graphics in SAS;

OPTIONS _LAST_=SASUSER.mydata;

* Histogram of q1;
PROC GCHART; VBAR q1; RUN;

* Bar charts of workshop & gender;
PROC GCHART; VBAR workshop gender; RUN;

* Scatter plot of q1 by q2;
PROG GPLOT; PLOT q2*q1; RUN;

Scatter plot matrix of all vars but gender;
* Gender would need to be recoded as numeric;
PROC INSIGHT;
SCATTER workshop q1-q4 * workshop q1-q4;
RUN;
```

SPSS

```
*Basic Graphics in SPSS using both legacy commands and GPL.

GET FILE="C:\mydata.sav".

* Legacy SPSS statements for histogram of q1.
GRAPH  /HISTOGRAM=q1 .

* GPL statements for histogram of q1.
GGRAPH
  /GRAPHDATASET NAME="graphdataset" VARIABLES=q1 MISSING=LISTWISE
  REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
 SOURCE: s=userSource(id("graphdataset"))
 DATA: q1=col(source(s), name("q1"))
 GUIDE: axis(dim(1), label("q1"))
 GUIDE: axis(dim(2), label("Frequency"))
 ELEMENT: interval(position(summary.count(bin.rect(q1))) ,
shape.interior(
  shape.square))
END GPL.
```

```
* Legacy SPSS statements for bar chart of gender.
GRAPH  /BAR(SIMPLE)=COUNT BY gender .

* GPL statements for bar chart of gender.
GGRAPH
  /GRAPHDATASET NAME="graphdataset" VARIABLES=gender
COUNT()[name=
  "COUNT"] MISSING=LISTWISE REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
 SOURCE: s=userSource(id("graphdataset"))
 DATA: gender=col(source(s), name("gender"), unit.category())
 DATA: COUNT=col(source(s), name("COUNT"))
 GUIDE: axis(dim(1), label("gender"))
 GUIDE: axis(dim(2), label("Count"))
 SCALE: cat(dim(1))
 SCALE: linear(dim(2), include(0))
 ELEMENT: interval(position(gender*COUNT),
shape.interior(shape.square))
END GPL.


* Legacy syntax for scatterplot of q1 by q2.
GRAPH   /SCATTERPLOT(BIVAR)=q1 WITH q2.

* GPL syntax for scatterplot of q1 by q2.
GGRAPH
  /GRAPHDATASET NAME="graphdataset" VARIABLES=q1 q2
MISSING=LISTWISE
  REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
 SOURCE: s=userSource(id("graphdataset"))
 DATA: q1=col(source(s), name("q1"))
 DATA: q2=col(source(s), name("q2"))
 GUIDE: axis(dim(1), label("q1"))
 GUIDE: axis(dim(2), label("q2"))
 ELEMENT: point(position(q1*q2))
END GPL.
* Chart Builder.
GGRAPH
  /GRAPHDATASET NAME="graphdataset" VARIABLES=workshop q1 q2 q3
q4
  MISSING=LISTWISE REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
```

```
 SOURCE: s=userSource(id("graphdataset"))
 DATA: workshop=col(source(s), name("workshop"))
 DATA: q1=col(source(s), name("q1"))
 DATA: q2=col(source(s), name("q2"))
 DATA: q3=col(source(s), name("q3"))
 DATA: q4=col(source(s), name("q4"))
 TRANS: workshop_label = eval("workshop")
 TRANS: q1_label = eval("q1")
 TRANS: q2_label = eval("q2")
 TRANS: q3_label = eval("q3")
 TRANS: q4_label = eval("q4")
 GUIDE: axis(dim(1.1), ticks(null()))
 GUIDE: axis(dim(2.1), ticks(null()))
 GUIDE: axis(dim(1), gap(0px))
 GUIDE: axis(dim(2), gap(0px))
 ELEMENT: point(position((

workshop/workshop_label+q1/q1_label+q2/q2_label+q3/q3_label+q4/q4
_label)*(

workshop/workshop_label+q1/q1_label+q2/q2_label+q3/q3_label+q4/q4
_label)))
END GPL.


* Legacy SPSS statements for scatterplot matrix of all but
gender.
* Gender cannot be used until it is recoded numerically.
GRAPH   /SCATTERPLOT(MATRIX)=workshop q1 q2 q3 q4.
execute.

* GPL statements for scatterplot matrix of workshop to q4
excluding gender.
* Gender cannot be used in this context.
GGRAPH
  /GRAPHDATASET NAME="graphdataset" VARIABLES=workshop q1 q2 q3
q4
  MISSING=LISTWISE REPORTMISSING=NO
  /GRAPHSPEC SOURCE=INLINE.
BEGIN GPL
 SOURCE: s=userSource(id("graphdataset"))
 DATA: workshop=col(source(s), name("workshop"))
 DATA: q1=col(source(s), name("q1"))
 DATA: q2=col(source(s), name("q2"))
 DATA: q3=col(source(s), name("q3"))
 DATA: q4=col(source(s), name("q4"))
 TRANS: workshop_label = eval("workshop")
 TRANS: q1_label = eval("q1")
```

	TRANS: q2_label = eval("q2") TRANS: q3_label = eval("q3") TRANS: q4_label = eval("q4") GUIDE: axis(dim(1.1), ticks(null())) GUIDE: axis(dim(2.1), ticks(null())) GUIDE: axis(dim(1), gap(0px)) GUIDE: axis(dim(2), gap(0px)) ELEMENT: point(position((workshop/workshop_label+q1/q1_label+q2/q2_label+q3/q3_label+q4/q4_label)*(workshop/workshop_label+q1/q1_label+q2/q2_label+q3/q3_label+q4/q4_label))) END GPL.

R	#Basic Graphics in R.
	
	load(file="c:\\mydata.Rdata")
	print(mydata)
	attach(mydata) #Makes this the default dataset.
	
	hist(q1)
	
	barplot(c(40,60))
	barplot(summary(gender))
	
	# These statements generate an error since workshop
	# is not a factor yet.
	barplot(summary(workshop))
	
	# Now use as.factor function to make workshop a factor.
	barplot(summary(as.factor(workshop)))
	
	# Scatterplot of q1 by q2.
	plot(q1,q2)
	
	# Scatterplot matrix of whole data frame.
	plot(mydata)

ANALYSIS

This section demonstrates some very basic hypothesis tests. Since the examples use our practice dataset, some of the tests are rather absurd. See Modern Applied Statistics in S and An Introduction to S and the Hmisc and Design Libraries for much more thorough and interesting coverage.

When performing a single type of analysis in SAS or SPSS, you prepare your commands and then submit them to get all your results at once. You could save some of the output using SAS' Output Delivery System or SPSS' Output Management System, but few people do. R on the other hand shows you very little output with each command and everyone uses its integrated output management capabilities. For example when running linear regression `mymodel<-lm(q4~q1,q2,q3)` will run and save your model, but tell you nothing about it. Following that with additional functions such as `summary(mymodel)`, `anova(mymodel)` and `plot(mymodel)` actually display the results.

In analysis of variance, SAS and SPSS default to partial (type III) sums of squares (SS) and F tests. Instead, R provides sequential ones. R's t-tests for individual parameters are the same as SAS' and SPSS', so you have both sets of p-values. If you want partial Fs, you can square the t values provided. Since it's easy to save a model in R you can compare any two with `anova(fullModel,reducedModel)`. This approach can test for the additional effect of a single variable, or a set of variables. Stepwise models are done via the `add1`, `drop1` and `stepAIC` functions.

The R examples require installing the Hmisc and prettyR packages beforehand. See details in ***Installing Add-on Packages.***

SAS

```
* SAS Program of Basic Statistical Tests;

libname Cdrive 'C:\';
data temp;
  set Cdrive.mydata;
  * pretend q2 and q1 are the same score
    measured at two times & subtract;
  myDiff=q2-q1; run;
* Basic stats in compact form;
proc means; var q1-q4; run;
* Basic stats of every sort;
proc univariate; var q1-q4; run;
* Frequencies & percents;
proc freq; tables workshop--q4; run;

*---Measures of association;

* Pearson correlations;
proc corr; var q1-q4; run;

* Spearman correlations;
proc corr spearman; var q1-q4; run;

* Linear regression;
proc reg;
```

	model q4=q1-q3;
	run;
	
	*---Group comparisons;
	
	* Chi-square;
	proc freq;
	tables workshop*gender/chisq; run;
	
	* Independent samples t-test;
	proc ttest;
	class gender;
	var q4; run;
	
	* Nonparametric version of above
	using Wilcoxon/Mann-Whitney test;
	proc npar1way;
	class gender;
	var q4; run;
	
	* Paired samples t-test (silly one);
	proc ttest;
	paired q1*q2; run;
	
	* Nonparametric version of above using
	Signed Rank test;
	proc univariate;
	var myDiff;
	run;
	
	*Oneway Analysis of Variance (ANOVA);
	proc glm;
	class workshop;
	model q4=workshop;
	means workshop gender / tukey; run;
	
	*Nonparametric version of above using
	Kruskal-Wallis test;
	proc npar1way;
	class workshop;
	var q4; run;

SPSS	* SPSS Program of Basic Statistical Tests. GET FILE='C:\mydata.sav'. DATASET NAME DataSet2 WINDOW=FRONT. * Descriptive stats in compact form. DESCRIPTIVES VARIABLES=q1 q2 q3 q4

```
  /STATISTICS=MEAN STDDEV VARIANCE MIN MAX SEMEAN .

* Descriptive stats of every sort.
EXAMINE VARIABLES=q1 q2 q3 q4
  /PLOT BOXPLOT STEMLEAF NPPLOT
  /COMPARE GROUP
  /STATISTICS DESCRIPTIVES EXTREME
  /CINTERVAL 95
  /MISSING PAIRWISE
  /NOTOTAL.
EXECUTE.

* Frequencies and percents.
FREQUENCIES VARIABLES=workshop gender q1 q2 q3 q4
  /ORDER=  ANALYSIS .
EXECUTE.

*---Measures of association.

* Person correlations.
CORRELATIONS
  /VARIABLES=q1 q2 q3 q4
  /PRINT=TWOTAIL NOSIG
  /MISSING=PAIRWISE .
EXECUTE.

* Spearman correlations.
NONPAR CORR
  /VARIABLES=q1 q2 q3 q4
  /PRINT=SPEARMAN TWOTAIL NOSIG
  /MISSING=PAIRWISE .
EXECUTE.

* Linear regression.
REGRESSION
  /MISSING LISTWISE
  /STATISTICS COEFF OUTS R ANOVA
  /CRITERIA=PIN(.05) POUT(.10)
  /NOORIGIN
  /DEPENDENT q4
  /METHOD=ENTER q1 q2 q3  .
EXECUTE.

*---Group comparisons;

* Chisquare.
CROSSTABS
  /TABLES=workshop  BY gender
```

```
  /FORMAT= AVALUE TABLES
  /STATISTIC=CHISQ
  /CELLS= COUNT ROW
  /COUNT ROUND CELL .
EXECUTE.

* Independent samples t-test.
T-TEST
  GROUPS = gender('m' 'f')
  /MISSING = ANALYSIS
  /VARIABLES = q4
  /CRITERIA = CI(.95) .
EXECUTE.

* Nonparametric version of above using
  Wilcoxon/Mann-Whitney test.
* SPSS requires a numeric form of gender for this procedure.
AUTORECODE
  VARIABLES=gender  /INTO genderN
  /PRINT.
NPAR TESTS
  /M-W= q4   BY genderN(1 2)
  /MISSING ANALYSIS.
EXECUTE.

* Paired samples t-test.
T-TEST
  PAIRS = q1  WITH q2 (PAIRED)
  /CRITERIA = CI(.95)
  /MISSING = ANALYSIS.
EXECUTE.

* Nonparametric version of above using
  Wilcoxon Signed Rank test.
NPAR TEST
  /WILCOXON=q1  WITH q2 (PAIRED)
  /MISSING ANALYSIS.
EXECUTE.

* Oneway analysis of variance (ANOVA).
UNIANOVA q4  BY workshop
  /METHOD = SSTYPE(3)
  /INTERCEPT = INCLUDE
  /POSTHOC = workshop ( TUKEY )
  /PRINT = ETASQ HOMOGENEITY
  /CRITERIA = ALPHA(.05)
  /DESIGN = workshop .
```

	* Nonparametric version of above using
	  Kruskal Wallis test.
	NPAR TESTS
	  /K-W=q4 BY workshop(1 3)
	  /MISSING ANALYSIS.
	EXECUTE.

R	# R Program of Basic Statistical Tests.
	
	load("c:\\mydata.Rdata")
	# You must install Hmisc and prettyR before running.
	library(foreign)
	library(Hmisc)
	library(prettyR)
	
	# Descriptive stats and frequencies.
	summary(mydata)
	
	# Means, freqencies & percents using describe
	# function from Hmisc package.
	describe(mydata)
	
	# Frequencies & percents using the freq function
	# from the prettyR package.
	freq(mydata)
	
	#---Measures of association.
	
	# Pearson correlations.
	
	# The rcorr function from the Hmisc package gives
	# output most like SAS & SPSS. It yields correlations,
	# n's and p-values. Note its use of the column bind
	# function, cbind.
	rcorr(cbind(q1,q2,q3,q4))
	
	# The cor function is built in but does not give
	# tests of significance.
	cor(data.frame(q1,q2,q3,q4),method="pearson",use="pairwise")
	
	# The cor.test function is built in and will give
	# the correlation, p-value and confidence intervals,
	# but only for two variables at a time.
	cor.test(q1,q2,use="pairwise")

```
# Spearman correlations using the Hmisc rcorr function.
rcorr( cbind(q1,q2,q3,q4), type='spearman')

# Linear regression.
myRegModel<-lm(q4~q1+q2+q3,data=mydata)
summary(myRegModel)
anova(myRegModel)
plot(myRegModel)
termplot(myRegModel)

#---Group comparisons.

# Crosstabulation and chi-square is available
# via the xtab function in the prettyR package.
# It provides output similar to SAS & SPSS.
xtab(~workshop+gender,data=mydata,chisq=TRUE)

# Crosstabulation & chi-square using R's built
# in functions. The output lacks percents and
# marginal totals.
myWG<-table(workshop,gender)
print(myWG)
chisq.test(myWG)

# Independent samples t-test.
t.test(q4[gender=='m'],q4[gender=='f'] )

# Nonparametric version of above
  using Wilcoxon/Mann-Whitney test.
wilcox.test(q4[gender=='m'],q4[gender=='f'] )

# Paired samples t-test (silly one).
t.test(q1,q2,paired=TRUE)

# Nonparametric version of above using
wilcox.test(q1,q2,paired=TRUE)

#Oneway Analysis of Variance (ANOVA).
myModel<-aov(q4~workshop,data=mydata)
summary(myModel)
anova(myModel
plot(myModel)
termplot(myModel)
```

```
#Nonparametric oneway ANOVA using
# the Kruskal-Wallis test.
kruskal.test(q4,workshop)
```

SUMMARY

As we have seen, R differs from SAS and SPSS in many ways. Below is a summary chart of some of the most important differences. You will find far more detail on each subject in the main body of this paper.

	SAS & SPSS	R
Output Management System	Output management systems are rarely used for routine analyses.	Output is routinely and easily passed through additional functions to get more results.
Macro language	A separate language used mainly for repetitive tasks or adding new functionality. Added capabilities are not run the same way as built in procedures.	An integral language that is used routinely. Also used to automate repetitive tasks and add new functionality. Added capabilities are run the same way as built-in ones.
Matrix Language	A separate language used only to add new features. Added features are not run the same way as built in procedures.	An integral part of R that you use even when selecting variables or observations. Added capabilities are run the same way as built-in ones.
Variable Labels	Built in. Used by all procedures.	Added on. Used by few procedures.
Publishing Results	See it formatted immediately in any style you choose. Quick cut & paste to word processor maintains table status and style. Can also export to a file.	Process output with additional procedures that route formatted output to a file. You don't see it formatted until you import it to a word processor or text formatter.
Data Size	Limited only by disk size.	Data must fit into the computer's RAM memory.
Data Structure	Rectangular data set.	Rectangular data frame, vector, matrix, list.

Choosing data	All the data for an analysis must be in a single data set.	Analyses can freely combine variables from different data frames or other structures.
Choosing Variables	Uses the simple lists of variable names in the form of x,y,z; a to z; a-z	Uses wide variety of selection by index value, variable name, logical condition (same as when selecting observations).
Choosing Observations	Uses logical conditions in IF, SELECT IF, WHERE	Uses wide variety of selection by index value, variable name, logical condition (same as when selecting variables).
Converting Data Structures to Match Procedure or Function	All procedures accept all variables.	Original data structure plus variable selection method determines structure of variables passed to analyses. Use of conversion functions may be required.
Controlling Procedure or Function	Statements such as CLASS and MODEL and options control the procedure.	The data's structure (its mode) determines what is done, along with options (arguments)

IS R HARDER TO USE?

As mentioned in ***The Five Main Parts of SAS and SPSS***, many SAS and SPSS users use only the data input and management commands along with procedures for graphics and analysis. They don't get the advantages offered by output management systems, macro languages or matrix languages. Many introductory SAS and SPSS documents focus only on those parts.

We have seen that R has many complexities that SAS and SPSS lack such as: multiple data structures, a very wide range of variable selection methods, functions that accept variables stored only in certain data structures or selected in certain ways, data structure conversion tools and workspace management commands.

So if you are a SAS or SPSS user who has happily avoided the complexities of output management, macros and matrix languages, R is indeed harder to use. On the other hand, it makes those added features so much easier to use than SAS or SPSS that you may find yourself more eager to expand your horizons. The added power of R and its free price make it well worth the added effort it takes to learn.

CONCLUSION

We have only touched on the full capabilities of R but now you should be ready to dive into a more comprehensive introductory manual or directly into a chapter on regression or ANOVA without being totally lost.

I hope to improve this document as time goes on, so please drop me a line at BobM@utk.edu or (865) 974-5230 to tell me what you think. How can I improve this for the next person? Negative comments are often the most useful, so don't be shy.

Have fun working with R!