Chapter 1
Getting Started with R and R Commander

"In my view R is such a versatile tool for scientific computing that anyone contemplating a career in science, and who expects to [do] their own computations that have a substantial data analysis component, should learn R."
(John Maindonald, R-help, January 2006)

1.1 Downloading and Opening

In these online sections I will work with the most recent version of R at the time of writing, Version 2.10.1, released in December 2009. Two major versions of R are released every year, so by the time you read this there will doubtless be a more updated version of R to download, but new versions of R mainly deal with fixing bugs, and I have found as I have used R over the years that the same commands still work even with newer versions of R. R was first created by Robert Gentleman and Ross Ihaka at the University of Auckland, New Zealand, but now is developed by the R Development Core Team.

The R Console is command-line driven, and few of its functions are found in the drop-down menus of the console. I have found it helpful to work with a graphical user interface created for R called R Commander, and will show you in this chapter how to get started using both of these. R Commander is highly useful for those new to R, but once you understand R better you will want to use the console to customize commands and also to use commands that are not available in R Commander.

1.1.1 Downloading and Installing R

You must have access to the internet to first download R. Navigate your internet browser to the R home page: http://www.r-project.org/. Under some graphics designed to demonstrate the graphical power of R, you will see a box entitled "Getting Started." Click on the link to go to a CRAN mirror. In order to download R you will need to select a download site, and you should choose one near you. It may be worth noting, however, that I have occasionally found some sites to be faulty at times. If one mirror location does not seem to work, try another one.

It is worth emphasizing here that you must *always* choose a CRAN mirror site whenever you download anything into R. This applies to your first download, but also after you have R running and you later want to add additional R packages.

After you have navigated to any download site, you will see a box entitled "Download and Install R." Choose the version that matches the computing environment you use (Linux, Mac, and Windows are available). For Windows, this choice will take you to a place where you will see "base" and "contrib" highlighted (see Figure 1.1). Click on "base." On the next page

the first clickable link will be larger than all others and as of this writing says “Download R 2.10.1 for Windows” (this version of R will change by the time you read this book, as R is being upgraded all the time; however, this should not generally result in much change to the information I am giving you in this book—R will still work basically the same way).

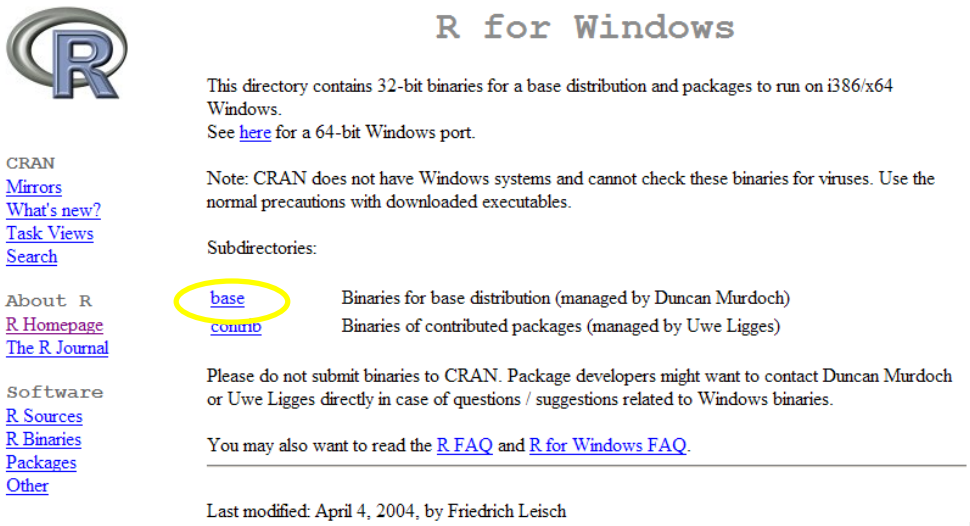


Figure 1.1 Downloading R from the internet.

Follow the normal protocol then for downloading and installing executable files on your computer. Do not worry about downloading the “contrib” binaries, which are shown in Figure 1.1 below the “base” choice. This is not actually something you or I would download.

Once the executable is downloaded on to your computer, double-click on it (or it may run automatically), and a Setup Wizard will appear, after you have chosen a language. You can either follow the defaults in the wizard or customize your version of R, which is explained in the following section.

1.1.2 Customizing R

In Windows, when you run the Setup Wizard, you will click a series of buttons and eventually come to Startup option. You have the option to customize, as shown in Figure 1.2. I recommend clicking Yes.

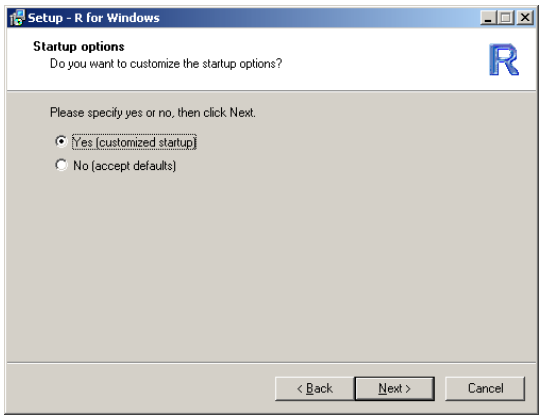


Figure 1.2 Startup options in R.

On the screen after the one shown in Figure 1.2, the Display Mode option, I recommend that you choose the SDI option. R's default is MDI, meaning that there is one big window (see Figure 1.3), but I find it much easier to use R when graphics and the help menu pop up in separate windows (as they do for the SDI choice). Once you have chosen SDI, I have no other recommendations about further customization and I just keep the default choices in subsequent windows.

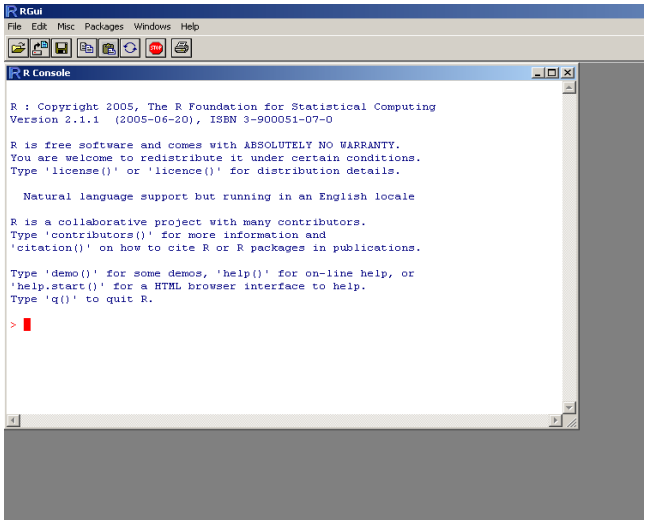


Figure 1.3 Multiple windows interface in R.

Once you have installed R, go ahead and open it. You will see the RGui screen (this stands for "R Graphical User Interface"). If you have not changed the setting to the SDI option, you will see that inside the RGui there is a smaller box called R Console, as shown in Figure 1.3. If you have changed to the SDI option, you will see a format where the R Console is the only window open, as in Figure 1.5.

You may want to further customize R by changing font sizes, font colors, etc. To accomplish this, navigate in the R Console, using the drop-down menus, to EDIT > GUI PREFERENCES. You will see a window as in Figure 1.4, called the Rgui Configuration Editor. If you have not done so before, you could change options to set the style to single windows (SDI). I also like to change the settings here to allocate more memory so that I can see all of my R commands in one session (I've put in "99999999" in the "buffer chars" line, and "99999" in the "lines" box). You might also want to change the default colors of the text commands here as well.

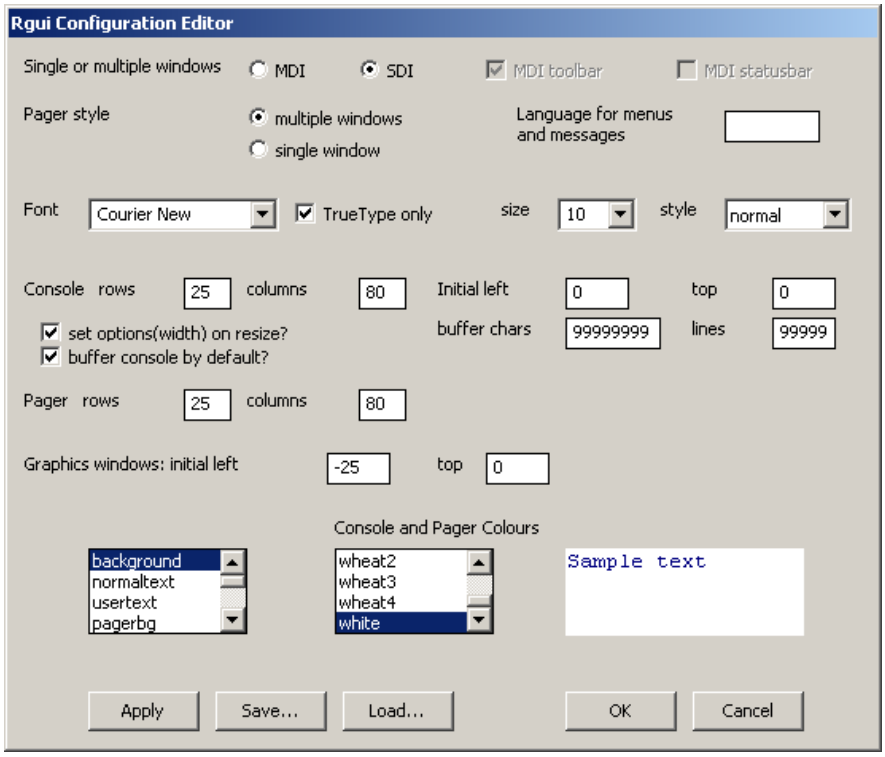


Figure 1.4 Customizing R with the Rgui Configuration Editor.

In order to change these values permanently, you will need to save them. Press "Save" and navigate to PROGRAM FILES > R > (R-2.10.1 >) ETC (the step in parentheses may not be necessary). Chose the Rconsole file and press "Save." A dialogue box will ask if you want to replace the previous file. Click on "Yes." Close down all windows and restart R to see the changes (you do not need to save the workspace image when you exit).

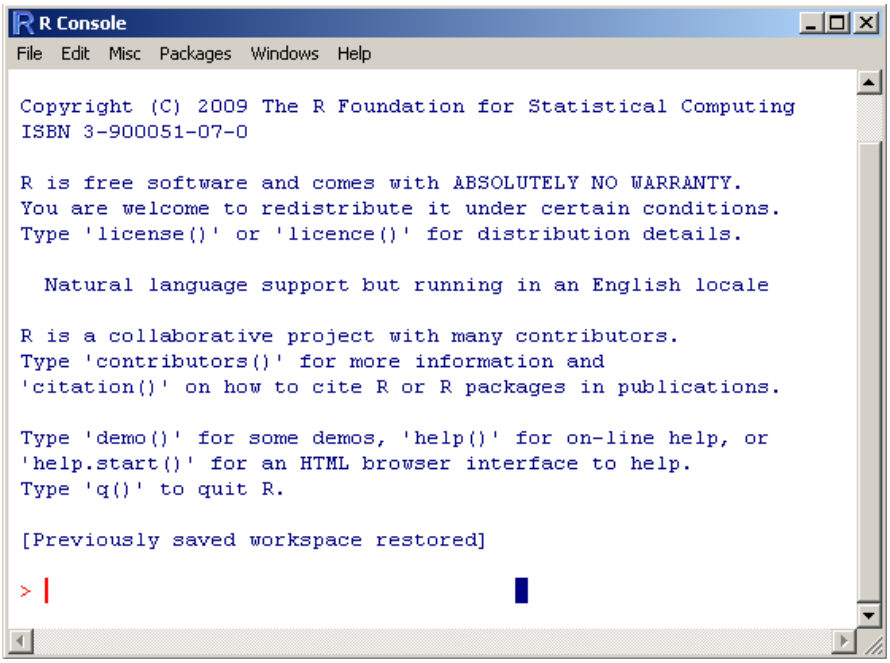


Figure 1.5 Single window interface in R.

At this point, you will still be wondering what you should do! The next step that I recommend is to download a more user-friendly graphical interface that works in concert with R called **R Commander**, created by John Fox**{** XE "R Commander" **}**. Readers familiar with SPSS and other drop-down menu type programs will initially feel more comfortable using R Commander than R. The R environment is run with scripts, and in the long run it can be much more advantageous to have scripts that you can easily bring back up as you perform multiple calculations. But to get started I will be walking the reader through the use of R Commander as much as possible.

1.1.3 Loading Packages and R Commander

In the R Console, navigate to the menu PACKAGES > INSTALL PACKAGE(S). You will need to select your CRAN mirror site first in order to continue with the installation. Once you have made your choice for a CRAN mirror site, a list of packages will pop up. For now, just scroll down alphabetically to the "r"s and choose Rcmdr. Press OK and the package will load. Instead of doing the menu route, you can also type the following command in the R Console:

```
install.packages("Rcmdr")
```

If you have not picked a CRAN mirror during your session, this window will automatically pop up after you type in your command. You must choose a CRAN mirror in a specific physical location to download from before you can continue with your installation.

Later on you will be asked to download other packages, and of course you might like to explore other packages on your own. There are hundreds of packages available, some for very specific purposes (like a recent one for dealing with MRI and fMRI data). One good

way to explore these is to use the RGui drop-down menu to go to HELP > HTML HELP. This will open a separate window in your internet browser (although you do not need to be connected to the internet for it to work). Click on the Packages link, and you will see a package index that gives a brief description of each package. Further clicking on specific packages will take you to pages that give you documentation for each command. At this point, most of this documentation will be cryptic and frustrating, so I do not recommend browsing this area just yet. As you become more familiar with R, however, this may be a helpful site.

Once R Commander has been downloaded, start it by typing the following line in the RGui window:

```
library(Rcmdr)
```

This must be typed exactly as given above; there is no menu command to obtain it. Notice that, because R is a scripted environment (such as Unix), spelling and punctuation matter. The "R" of R commander must be capitalized, and the rest of the letters ("cmdr") must be lower-case. The Arial font is used in this book to indicate actual commands used in the R interface.

The first time you try to open R Commander, it will tell you that you need to download some additional files; just follow the directions for this, and install from the CRAN site, not a local directory. Once those files are downloaded, use the library() command again (or type the up ↑ arrow until you come back to your previous command), and R Commander will open. The R Commander window is shown in Figure 1.6.

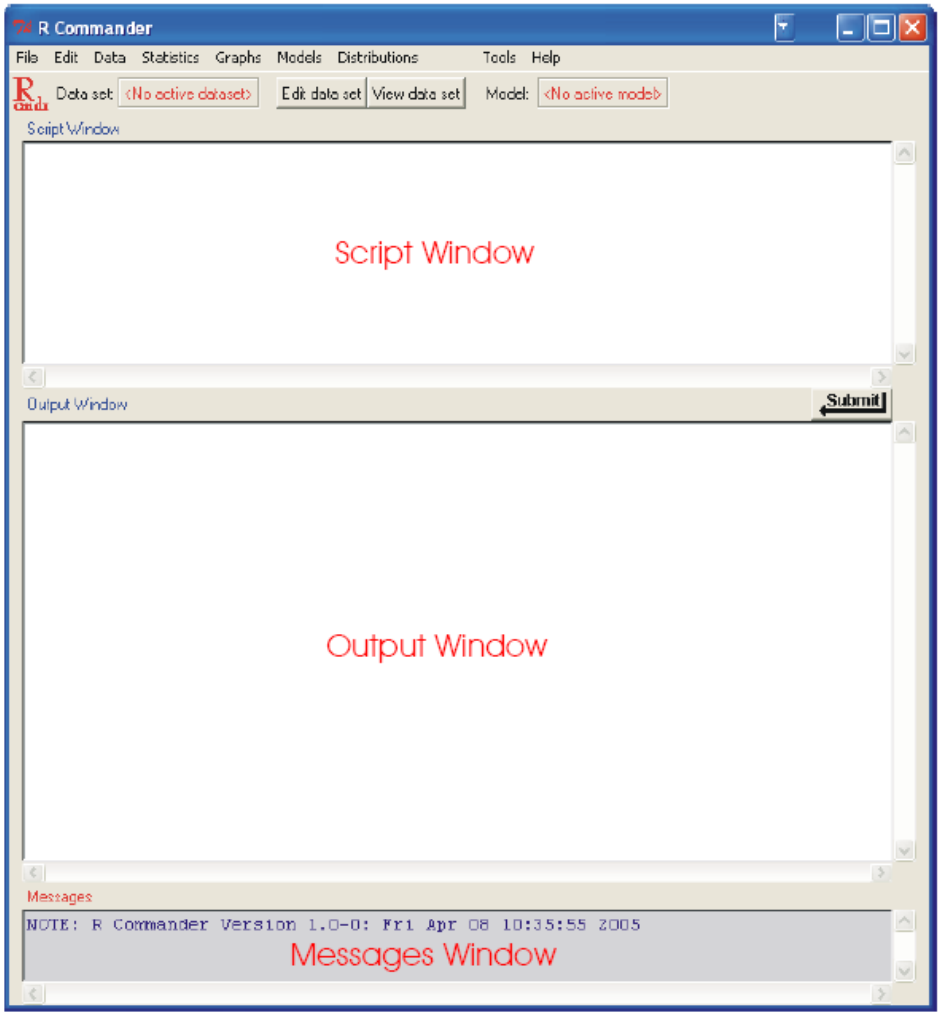


Figure 1.6 The R Commander graphical user interface (GUI) taken from Fox (2005).

1.2 Working with Data

While you can certainly learn something about R without any data entered, my first inclination is to get my own data into the program so that I can start working with it. This section will explain how to do that.

1.2.1 Entering Your Own Data

R has ways that you can type in your own data. However, Verzani (2004) notes that entering data provides a "perfect opportunity for errors to creep into a data set" (p. 23). If at all possible, it is better to be able to read your data in from a database. But sometimes this is not possible and you will need to type your data in.

The most intuitive way to enter data will be using R Commander (remember, to open R Commander from the R Console, type library(Rcmdr) first!). First navigate to the DATA > NEW DATA SET window. Replace the default title "Dataset" and enter a name for your data set (see Figure 1.7). I named mine with my last name, but without a hyphen. Remember that, whether you capitalize or not, you will need to enter the name for your data set in exactly the same way when you call it up again, so for my name I will have to remember that the initial "L" as well as the "H" of Hall is capitalized. If you enter an illegal character in the name, the "New Data Set" dialogue box will reset itself. The restrictions are similar to those for other types of programs like SPSS (no numbers initially, certain kinds of punctuation like a slash "/" not allowed, etc.).

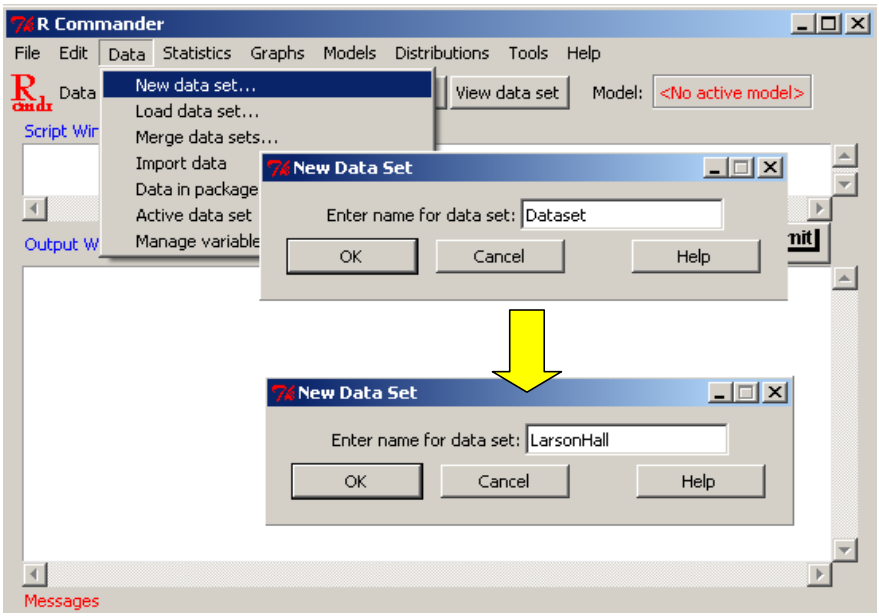


Figure 1.7 Setting up a new spreadsheet for data in R Commander.

After you enter the name and click OK, a spreadsheet like the one in Figure 1.8 automatically appears. In the picture below, I clicked on the first column, "var1," and a box popped up to let me enter a new name for the variable and choose whether it would be numeric or character. A **variable** is a collection of data that are all of the same sort. There are two major types of variables in R: **character{** XE "character variables" **}** and **numerical variables{** XE "numerical variables" **}**. Character variables are non-numerical strings. These cannot be used to do statistical calculations, although they will be used as grouping variables. Therefore, make any variables which divide participants into groups "character" variables. Be careful with your variable names. You want them to be descriptive, but if you use commands in R you will also need to type them a lot, so you don't want them to be too long.

Once you have set your variable names, you are ready to begin entering data in the columns.

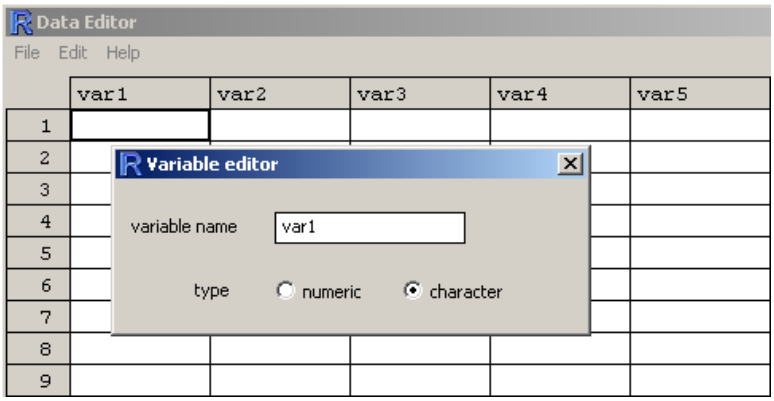


Figure 1.8 Entering data directly into R through the data editor.

In order to follow the same steps using the R Console, first create a data frame, giving it whatever name you want. Here I name it "Exp1."

```
Exp1=data.frame()
```

Now give a command that lets you enter data into your newly created data frame:

```
fix(Exp1)
```

> Tip: If you do enter your data directly into R or R Commander it might be worth making a simple .txt file that will explain your variable names more descriptively. Then save that file wherever you store your data files. Sometimes when you come back to your data years later, you cannot remember what your variable names meant, so the .txt file can help you remember.

After you do this, the same spreadsheet as was shown in Figure 1.8 will appear in a window and you can edit it just the way described above.

To make more columns appear in your spreadsheet, simply use the arrow keys to move right and more columns will appear. You can close this spreadsheet and the data frame will still be available as long as the same session of R is open. However, you will need to save the data before you exit from R if you want it to be available to you at a later time.

If you want to go back later and edit any information, this is simple to do by using either the "Edit data set" button along the top of R Commander, or the `fix(Exp1)` command as given above, and typing over existing data. The only way to add more columns or rows after you have created your spreadsheet is by adding these to the outermost edges of the existing spreadsheet; in other words, you cannot insert a row or column between existing rows or columns. This does not really matter, since data can later be sorted and rearranged as you like. It is possible, however, to delete entire columns of variables in R Commander by using the DATA > MANAGE ACTIVE VARIABLES IN DATA SET > DELETE VARIABLES from data set command.

Actually, anything which can be done through a drop-down menu in R Commander can also be done by means of commands in R. R Commander puts the command that it used in the top "Script Window."

1.2.2 Importing Files into R through R Commander

Importing files into R can most easily be done by using R Commander. Click on DATA > IMPORT DATA. A list of data types that can be imported will be shown, as in Figure 1.9.

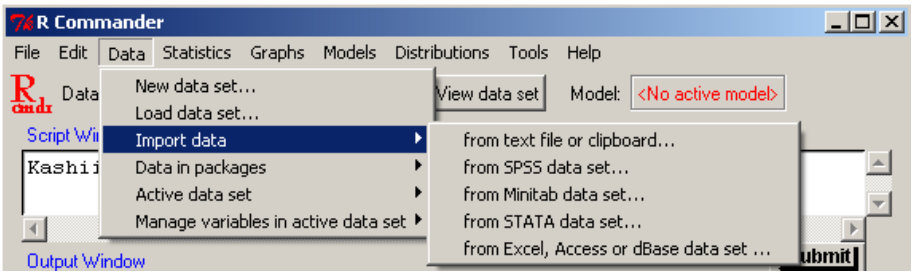


Figure 1.9 Importing data through R Commander.

R Commander has commands to import text files (including .txt, .csv, .dif, and .sylk), SPSS, Minitab, STATA, and Microsoft Excel, Access, or dBase files. These will be imported as data frames (this information will be more important later on when we are working with commands which need the data to be arranged in certain ways). Data from SAS, Systat and S-PLUS can also be imported (see the RData ImportExportManual.pdf under the HELP > MANUALS (IN PDF) menu for more information).

You may need to experiment a couple of times to get your data to line up the way you want. I will demonstrate how to open a .csv file. The box in Figure 1.10 will appear after I have made the choice DATA > IMPORT DATA > FROM TEXT FILE, CLIPBOARD OR URL. The .csv file is comma delimited, so when I open up the dialogue box for text files I need to make the appropriate choices, as seen in Figure 1.10.

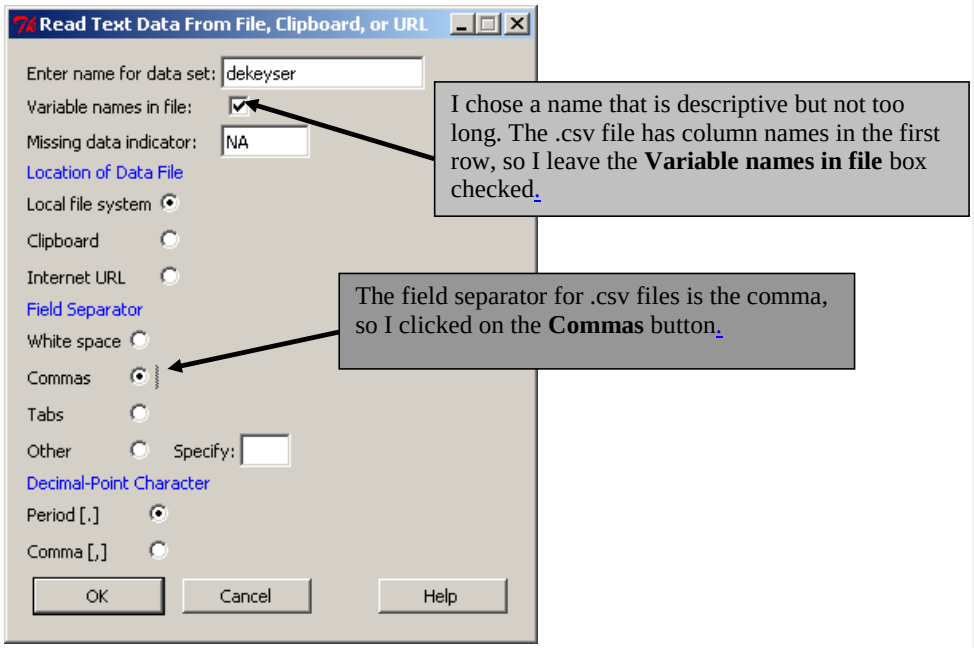


Figure 1.10 Importing text files into R through R Commander.

Once you choose the correct settings, R Commander lets you browse to where the files are on your computer (if you are uploading from your local file system).

Getting data into R using line commands is much more complicated. It is something I never do without R Commander, but it can be done. For more information on this topic, use the R Data Import/Export manual, which can be found by going to the R website, clicking on the link to "Manuals" on the thin left-side panel, and then finding the manual in the list of links online.

Getting Data into R

Using R Commander menu commands:

DATA > NEW DATA SET Opens a new spreadsheet you can fill in
DATA > IMPORT DATA > FROM TEXT FILE, CLIPBOARD OR URL Imports data

Using R line command:

```
Exp1=data.frame() #create a data frame and name it (replace underline)
fix(Exp1)         #enter data in your new data set
```

Formatted: Font: Italic

Formatted: Font: Italic

Deleted: ¶

1.2.3 Viewing Entered Data

Once you have your data entered into R, you can look at it by clicking on the "View Data Set" button along the top of the R Commander box (see Figure 1.11). To make sure that the

data set you want is the active data set, make sure it is listed in the "Data Set" box along the top of R Commander. If the data you want is not currently active, just click in the box to select the correct data set.

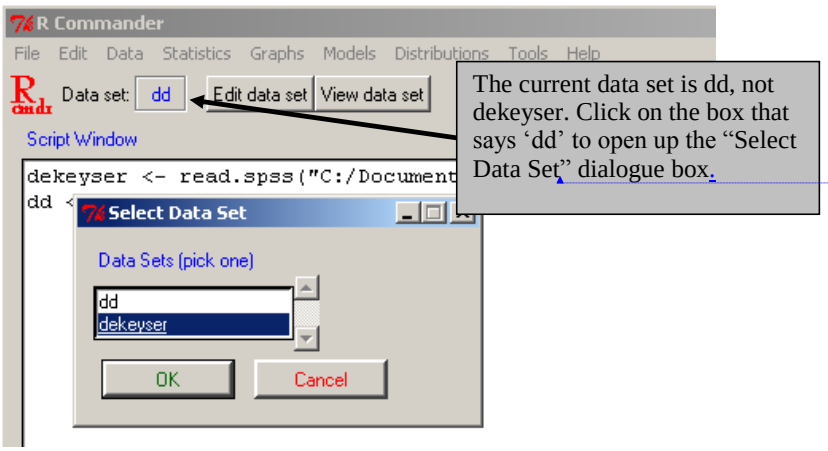


Figure 1.11 Selecting a data set in R Commander.

You can also get a print-out of any data set in the R Console by simply typing the name of the data set:

```
> dekeyser
   AGE GJTSCORE   STATUS
1    8       170 Under 15
2   11       181 Under 15
3    9       198 Under 15
4   11       194 Under 15
5   13       196 Under 15
```

You can see any one of these variables (which are in columns) by typing the name of the data set followed by a "**${** XE "$" **}**" symbol and the name of the variable, like this:

```
> dekeyser$GJTSCORE
 [1] 170 181 198 194 196 193 199 195 197 194 183 183 196 197 199 136 132 139 153
[20] 141 147 175 170 126 110 119 134 118 167 122 126 143 129 190 143 175 153 184
[39] 129 125 146 111 127 123 151 143 176  76 164 152 162 177 174 186 132 155 155
```

If you have forgotten the names of the variables, in R you can call up the names with the names(dekeyser) command.

```
> names(dekeyser)
[1] "AGE"      "GJTSCORE" "STATUS"
```

Note that any line with the command prompt ">" showing indicates that I have typed the part that follows the command prompt into R. Throughout this book I have chosen not to type in the prompt when I myself type commands; however, when I copy what appears in R you will see the command prompt. If you copy that command, do **not** put in the command prompt sign!

Formatted: Font: Not Bold

1.2.4 Saving Data and Reading It Back In

Once you have created your own data set or imported a data set into R, you can save it. There are various ways of doing this, but the one which I prefer is to save your data as a .csv file. This is a comma-delimited file, and is saved as an Excel file. The easiest way of doing this is in R Commander. Go to DATA > ACTIVE DATA SET > EXPORT ACTIVE DATA SET, and you will see the dialogue box in Figure 1.12.

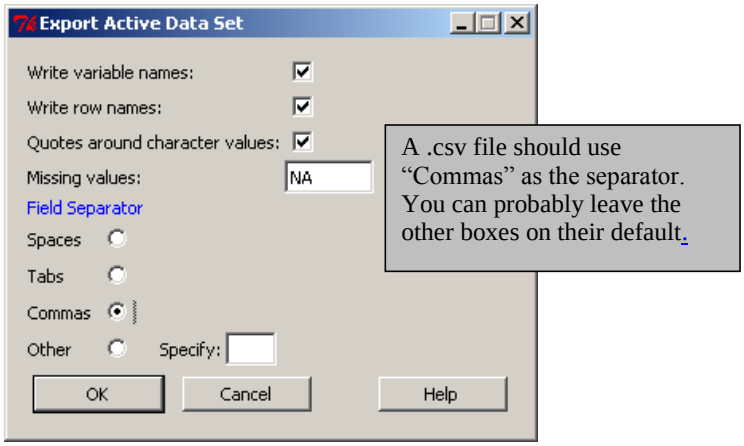


Figure 1.12 Saving data files in R Commander.

Once you click OK, you will see a dialogue box that lets you save your data in the usual way. Remember that the data you want to save should be the currently active data (you will see the name in the "File name" line, so if it's wrong you will know!). Now you can navigate to wherever you want to save the file. Using this method, R automatically appends .txt" to the file, but, if you want a .csv file, take out the .txt and replace it with .csv as you are saving it. When you want to open the file at a later date in R, you will repeat the process explained above for importing files. You can also save files through R Console, but it is a little more complicated. Here is the syntax for how to do it:

```
write.csv(dekeyser, file="dekeyser.csv", row.names=FALSE)
```

write.csv(x)	Writes a comma-delimited file in Excel format; by default will keep column names; it saves the data as a data frame (unless it is a matrix).
file="dekeyser.csv"	Names file; don't forget the quotation marks around it!
row.names=FALSE	If set to FALSE, names for rows are not expected. If you do want row names, just don't include this argument.

The object will be saved in R's working directory. You can find out where the working directory is by typing the command getwd() like this:

```
getwd()
[1] "C:/Documents and Settings/jenifer/My Documents"
```

You can see that my working directory is in the My Documents folder. If you'd like to reset this, you can do this by using the setwd() command, like this:

setwd("C:/Documents and Settings/jenifer/Desktop/R Files")

If the file you saved is in the working directory, you can open it with the read.csv(x) command. To make this file part of the objects on the R workspace, type:

dekeyser=read.csv("dekeyser.csv")

This will put the data set dekeyser on your workspace now as an object you can manipulate. You can verify which objects you have on your workspace by using the ls() command. In this case, you will leave the parentheses empty. I have a lot of objects in my workspace so I've only shown you the beginning of the list:

```
> ls()
  [1] "a.WH"                                [5] "AdultRegVerb"
  [2] "accuracy"                            [6] "amod"
  [3] "Additional.Expenses.March.2010"      [7] "AnovaModel.1"
  [4] "AdultIrregularVerb"                  [8] "AnovaModel.2"
```

1.2.5 Saving Graphics Files

Graphics are one of the strong points of R. You will want to be able to cut and paste graphics from the "**R Graphics Device.{** XE "R Graphics Device" **}**" The Graphics window has a pull-down menu to help you do this. Pull down the File menu, and you have the choice to SAVE AS a metafile, postscript, pdf, Png, bmp, or jpeg file. What I usually do, however, is take the COPY TO CLIPBOARD choice, and then choose AS A METAFILE. The file is then available on the clipboard and can be pasted and resized anywhere you like. These two ways of savings graphics, either by converting directly into a separate file or by copying on to the clipboard, are shown in Figure 1.13.

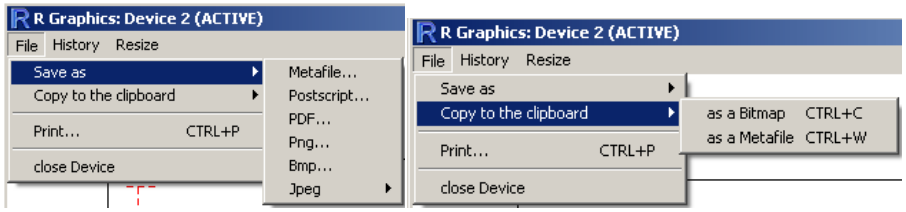


Figure 1.13 Choices for saving graphics or copying them to the clipboard.

1.2.6 Closing R and R Commander

When you close R and R Commander, they will ask you various questions about whether you want to save information. One question is "Save workspace image?" If you say yes, all of the objects and command lines you used in your session will be available to you when you open R up again. For example, in this session I had the object dekeyser open, and if I save the workspace that object will be automatically available when I open up R again, as will all of the commands I used, such as write.csv(dekeyser, file="dekeyser.csv", row.names=F). Once reloaded, I can access these commands by simply clicking on the up ↑ and down ↓ arrows as usual.

Thus one of the advantages of R is that by saving your workspace with your other data and files you will have easy access to all of your data files and commands for future reference. However, R does not automatically let you give the workspace a special name and save it in a

place different from the working directory if you just shut down your windows. Therefore, you should use the R Console menu to save your workspace before you close the windows. Choose FILE > SAVE WORKSPACE from the drop-down menu in R Console. You will be able to navigate to the folder of your choice and give the .RData file a unique name. When you start R again, you can load this workspace by using the drop-down menu in R Console, going to FILE and then LOAD WORKSPACE.

If you have saved a particular workspace and this comes up every time you start R but you want to start with a fresh workspace, you can find the .RData file in your working directory and simply delete it. The next time you open up R there will be no objects and no commands saved (you can verify this by using the command ls() to list all objects).

Another question is "Save script file?" If you say yes, all of your command lines will be saved in a file called .Rhistory wherever your working directory is. If you start R again by clicking on this icon, all of your previous commands will be available. This command is also available in the R Console by using the File menu and choosing SAVE HISTORY. Again, this feature can be quite useful for referencing your analysis in the future.

1.3 Application Activities for Practicing Entering Data into R

1. Create a new data set called Count. Assume it has two variables, Score and Group, where Score is numeric and Group is a character variable. Randomly enter the numbers 1–9 for the score (create 10 entries). For Group label the first five entries "C" for control group and the next five entries "T" for treatment group. When you are finished, type:

```
Count
```

in R Console to make sure your data frame looks good.

2. There is a .txt file that you will use later in this book called read.txt (download it on to your computer from the Routledge website). Import this file into R, and call it read. This data has variable names in the file and the numbers are separated by spaces, so in the dialogue box for importing choose the "Field Separator" called "Other" and type a space into the box that says to "Specify." Use the button along the top of the R Commander GUI that says "View data set" to make sure the data frame looks appropriate.

3. There are many SPSS files that you will need to import into R. Let's try importing the one called DeKeyser2000.sav (files labeled .sav are from SPSS; again, you will need to download this on to your computer from the Routledge website). Name the file dekeyser, and keep the other defaults for importing the same. After it is imported, either use the "View data set" button in R Commander or type the name of the file in the R Console to see the data frame.

1.4 Introduction to R's Workspace

In a previous section I mentioned the command ls(), which lists the names of objects that are in R's workspace. These are the data sets the user has imported into R, and the functions the user has defined. R Commander's "Data set" button will list the imported data sets, but not the functions that the user has defined (you'll understand better how you yourself might define a function when you work through the next chapter, "Understanding the R Environment").

If you want to clean this workspace up, you can remove data sets or functions through the rm() command:

rm(a.WH) # “a” was an object in the workspace shown at the end of a previous section

This can be useful if you have defined something you want to redefine or just want to clear things off the workspace.

1.4.1 Specifying Variables within a Data Set, and Attaching and Detaching Data Sets

As you are working with a specific data set in R, it can be highly useful to attach your data set to the R search path. By attaching your data set, you are able to change the way you call for arguments, and thus save yourself some time in typing. When you attach a data frame you can refer to variables by their names alone without having to call them up as components of the larger data frame. For example, in the data frame called dekeyser (which you imported in a previous application activity) there are three variables, as can be seen in this R printout:

```
> names(dekeyser)
[1] "Age"      "GJTScore" "Status"
> Age
Error: object 'Age' not found
> dekeyser$Age
 [1]  8 11  9 11 13  4  1 12  3 10  6 11  5  9  3 28 20 23 22
[20] 30 27 27 27 27 23 26 21 23 17 25 20 34 25 22 33 35 29 28
[39] 26 40 33 37 21 26 26 28 25 26 22 20 21 38 24 26 32 23 27
```

Note that, if I try to call up the variable Age from the DeKeyser data frame, it tells me it can’t be found. That is because, to call up a variable, I need to first specify the data set that the variable comes from, as in the command to print the variable Age with dekeyser$Age.

If, however, I attach the data frame first, I can legitimately call up just the variable Age.

```
> attach(dekeyser)
> Age
 [1]  8 11  9 11 13  4  1 12  3 10  6 11  5  9  3 28 20 23 22
[20] 30 27 27 27 27 23 26 21 23 17 25 20 34 25 22 33 35 29 28
[39] 26 40 33 37 21 26 26 28 25 26 22 20 21 38 24 26 32 23 27
```

The command search() will tell me which objects are attached most highly in R’s search path (it always starts with “.GlobalEnv,” so don’t try to remove that one).

```
> search()
 [1] ".GlobalEnv"        "dekeyser"
 [3] "package:survival"  "package:splines"
```

After you are done working with a particular data set, it is always a good idea to take it off the search path so that you can later attach other data sets.

detach(dekeyser)

You can also remove the data set by using the command rm(), but this will completely delete the data from R.

1.5 Missing Data

Whether you enter your own data or import it, many times there are pieces of the data missing. R will import an empty cell by filling in with the letters "NA" for "not available." If you enter data and are missing a piece, you can simply type in NA yourself.

R commands differ as to how they deal with NA data. Some commands do not have a problem with it, other commands will automatically deal with missing data, and yet other commands will not work if any data is missing.

You can quickly check if you have any missing data by using the command is.na(). This command will return a "FALSE" if the value is present and a "TRUE" if the value is missing.

```
is.na(forget)
         ID   SEX   AGE STATUS ENGUSE USYEARS RETURNAG TRIPTIME
[1,] FALSE FALSE FALSE  FALSE  FALSE    TRUE     TRUE    FALSE
[2,] FALSE FALSE FALSE  FALSE  FALSE    TRUE     TRUE    FALSE
```

The results return what are called "logical" answers in R. R has queried about whether each piece of data is missing, and is told that it is false that the data is missing. If you found the answer "TRUE," that would mean data was missing. Of course, if your data set is small enough, just glancing through it can give you the same type of information!

A way to perform listwise deletion and rid your data set of all cases which contain NAs is to use the na.action=na.exclude argument. This may be set as an option, like this:

```
options(na.action=na.exclude)
```

or may be used as an argument in a command, like this:

```
boxplot(forget$RLWTEST~lh.forgotten$STATUS, na.action=na.exclude)
```

In the case of the boxplot, however, this argument does not change anything, since the boxplot function is already set to exclude NAs.

1.6 Application Activities in Practicing Saving Data, Recognizing Missing Data, and Attaching and Detaching Data Sets

1. Create a new file called "ClassTest." Fill in the following data:

Pretest	Posttest	DelayedPosttest
22	87	91
4	56	68
75	77	75
36	59	64
25	42	76
48	NA	79
51	48	NA

You are entering in missing data (using the NA designation). After you have created the file, go ahead and check to see if you have any missing data (of course you do!).

2. Using the same data set you just created in activity 1, call up the variable Posttest. Do this first by using the syntax where you type the name of the data set as well as the name of the variable in, and then attach the data set and call up the variable by just its name.

3. Save the data set you created in activity 1 as a .csv file.

1.7 Getting Help with R

R is an extremely versatile tool, but if you are a novice user you will find that you will often run into glitches. A command will not work the way I have described, and you won't know why. So, to help you out, I'd like to introduce you to ways of getting more help in R.

First of all, if one of the steps I have outlined in the book doesn't work, first try going through the troubleshooting checklist below. I assume that your problem arises when using the R Console, not when using R Commander.

Troubleshooting checklist for novices to R:

- Have you typed the command *exactly* as shown in the book? Are the capitalized letters capital and the non-capitalized letters non-capital? Since R records what you do, you can go back and look at your command to check this.
- Have you used the appropriate punctuation? Have you closed the parentheses on the command? If you typed the command in Microsoft Word first, you might have a problem with smart quotes. An error message like this:

Error: unexpected input in "hist(norm.x, xlab=”"

shows there is a problem with the quote marks.

- Do you have any missing data (marked with NA)? If so, can the command you are trying to use deal with NAs? Read the help file to see what is said about missing data there. You may need to delete the row that contains missing data; try imputing the data (topics treated in Chapter 3) so that you will not have any NAs.
- What is the structure of your file? Is your file in the correct format for the command? Use the str() command to see whether your variables are numerical, character, or factors.

Next, there are help files available in R. If the command you want to use doesn't seem to be working correctly, you can call for further help. For example, if you wanted more information about how to set the working directory with the setwd() command, you would type:

```
help(setwd)
```

If you don't know the name of a command that you would like to use, you can use the help function to try to find the default command for that test in R. For example, if you want to use a t-test but don't know the command name, R will suggest you use the command:

```
help.search("ttest")
```

You can actually find the t-test command if you run a t-test in R Commander, and as a novice that is the way that I recommend going first. However, later in your career with R, you may want help with functions that are not in R Commander, and you can use this command to help search for help files.

Help files in R are extremely helpful, but they can be confusing at first. I was first thrown by the expression "foo." This means to substitute in whatever the name of your file is. For example, write.csv (foo, file="foo.csv") means to just put in the name of your file wherever

"foo" is. Thus, if my file is dekeyser, I would read the help files as telling me to use the command write.csv(dekeyser, file="dekeyser.csv").

There is more information in Appendix A on how to understand R's help files, but here are a couple of tips. Let's say you want to figure out how t-tests work, and you have discovered the command is t.test(). The last section, the Example section, may be the most useful for you at first. You can type:

example(t.test)

Some commands and results will appear in the R Console, and if there are any graphics they will pop up in the R Graphics device. Here is some output from the t-test:

```
t.test> ## Classical example: Student's sleep data
t.test> plot(extra ~ group, data = sleep)
Waiting to confirm page change...

t.test> ## Traditional interface
t.test> with(sleep, t.test(extra[group == 1], extra[group == 2]))

        Welch Two Sample t-test

data:  extra[group == 1] and extra[group == 2]
t = -1.8608, df = 17.776, p-value = 0.0794
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.3654832  0.2054832
sample estimates:
mean of x mean of y
     0.75      2.33
```

Remember, the number sign ("#") means commentary, so the beginning of this example tells you it is a classical example. The next line calls for a plot, and here is where the Graphics Device popped up. When I clicked on it, it outputted a boxplot. The fifth line down shows me one way of calling for a t-test, by using the with() command, where the data is called sleep. I see that the t-test on that line must have two arguments, but in this case it looks as though the data are all found in the same column called "extra," but are split into two groups, labeled simply as 1 and 2. After that the results for this test are printed out. Since you probably don't know the with() command yet, this example may not be clear, but the "Usage" and "Arguments" sections of the help file will tell you what types of arguments you need to put in the command. For t-tests, you will see in the "Usage" section that the syntax is:

t.test(x, . . .)

This means that you must have an argument "x" in the syntax. The "Arguments" section tells you that x is "a (non-empty) numeric vector of data values." In other words, you must have at least one vector (one column) of data as an argument. One vector would be fine for a one-way t-test, but most t-tests will include two vectors of data, and the "Arguments" section tells you that another argument "y" can be included, which is another numeric vector (also, the dots in the syntax show that other things, including those things which are listed in the Arguments section, can be inserted into the command).

The third entry in the "Arguments" section is shown below.

`alternative` a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`. You can specify just the initial letter.

This means that you can insert a command "alternative" into the syntax, like this:

```
t.test(x,y, alternative="two.sided")
```

The entry does note, however, that "two.sided" is the default, so it doesn't have to actually be inserted (as can be seen in the two-tailed t-test found in the example data earlier). However, if you want to have a one-tailed test, you can use the alternative = "greater" or alternative = "less" argument to call for this.

If you keep trying to understand the help files, eventually they will provide help! There are also some useful FAQs and manuals under R Console's menu choice Help.

One more place you can try going for help is the online R community. If you go to R's website (www.r-project.org) and click on "FAQs" on the left-hand menu, you can find answers to common questions. The "Search" link on the left-hand menu takes you to a list of places to search for archived answers to R questions. One I like is http://finzi.psych.upenn.edu/search.html. You actually can submit questions and get help with R, but be sure to follow the protocols and search the archives first so you won't be asking a question that has already been asked. I have found that, if you follow the rules for posting to the R help so that you submit a legitimate question, people there can get you answers quite quickly!

1.8 Using R as a Calculator

There are many excellent books which you might read to help you learn more about R. I have read several of them (Crawley, 2007; Dalgaard, 2002; Verzani, 2004), and what helped me most was following along on my computer, doing in R what they showed on the page. These books all started on the process of helping the reader to understand how R works by beginning to use R as a calculator. I will follow their example. In fact, if you should do nothing more than learn how to use R as a calculator, you'll be happy you did. I don't use a calculator anymore for my monthly budget; I use R, and I hope after this chapter is finished you'll see why! But my main point here is that, in trying to understand how R actually works, you really need to dig in and copy what I am doing. Chapter 1 helped you get R and R Commander set up on your computer. You now need to continue being active by replicating on your version of R what I do in this chapter. You can take this exercise a step further if you not only copy what I do, but if you intentionally make mistakes and see what happens when you do so. Working this way you will learn a lot about R and how it works.

R can add up numbers. Type the following sequence and see what happens:

```
67+35+99+10308
```

R will follow your input line with an output line that looks like this:

```
[1] 10509
```

This is the sum of the numbers you just added up. The symbols for other basic mathematical functions are a dash for a minus sign (-), a star for multiplication (*), and a slash for division (/). A decimal point is represented by a period (.). Use parentheses to perform mathematical functions in a preferred order. Thus

```
(240+50*10)/2 #240 plus 50 times 10 divided by 2
```

will be a different result from

```
240+(50*10)/2
```

because the parentheses group the numbers differently.

> Tip: You can scroll through previous commands by hitting the arrow up button. Once you have a previous command in the input line, you cannot use your mouse to move to different parts of the command in order to edit, but you can use the arrow forward or arrow back keys to edit your command. Try this with the parentheses for the two numbers above; bring up the previous command with the parentheses around (240+50*10) and delete the first parenthesis to put it in front of the 50 now. You'll want this skill for when you do the application activity. You can use the escape button (ESC) to get back to a blank prompt.

Deleted: e

Notice that I added a commentary after a hash mark in one of the lines above. This is a traditional way to add commentary after the command line in R. If you could copy and paste this command into R, the hash mark would tell R to ignore anything after it, and it would not try to compute anything. If you did not enter the hash mark, like this:

```
(240+50*10)/2 240 plus 50 times 10 divided by 2
```

and then entered this line into R, you would see the following error message:

```
Error: unexpected numeric constant in "(240+50*10)/2 240"
```

When working with R, you are sure to receive lots of error messages. It is useful to pay attention to these, as they can help you figure out what you did wrong.

One more useful set of functions is raising a number to a power and taking its square root. Take the square root by using the command sqrt(). The parentheses indicate that you need to put the number you are calculating into the parentheses after that command, like this:

```
sqrt(16)
[1] 4
```

Remember that, if you are using one of R's built-in functions, like the square root calculation, it will require that the arguments to the command be found in parentheses. You will get an error message if you forget to put the parentheses in. The error message will depend on what mistake you made. Look at the following examples:

```
sqrt32
Error: object "sqrt32" not found
```

Here, R thinks you are calling an object that has the name "sqrt32," but it doesn't have that object in its memory.

```
sqrt 32
Error: unexpected numeric constant in "sqrt 32"
```

Here the error message is different because I put a space between the command and the number, so now R doesn't think it's an object, but it can't evaluate the function without the parentheses.

```
sqrt(32
+
```

Here I put the first parenthesis but forgot the second one so R writes a plus sign to give me a chance to continue and finish my command. If I typed an ending parenthesis R would carry out the calculation. However, if you want to get out of a command without finishing the calculation, press the ESC button.

You might also want to raise a number to a power, and you can do this by adding the caret (^) after the base number and before the number you want to raise it to. Therefore, the number 10^5 would be represented in R as 10^5. Notice the result you get when you try to calculate 10^5.

```
10^5
[1] 1e+05
```

R doesn't think you need to see the number 100000. But if it's been a while since you've done math, remember that here the "e" means you take the number that's in front of it and move the decimal point to the right the number of spaces indicated by the number after the plus sign. In this case that number is simple (a 1 with 5 zeros after it), but in other cases it wouldn't be:

```
937546583^9
[1] 5.596747e+80
```

R as a Calculator

To use R as a calculator, enter numbers and numerical function symbols after the command prompt.

+ = add
- = subtract
* = multiply
/ = divide
sqrt() = square root function
^ = raise a number before the caret symbol to a power given after the symbol

Use parentheses to group your mathematical operations, if needed.

Formatted: Font: Italic

Formatted: Font: Italic

1.9 Application Activities with Using R as a Calculator

Perform the mathematical functions described below using R.

1. Add the numbers 88, 2689, 331, 389, and 2.

2. Subtract 89.32 from 25338.

3. Multiply 59 and 26. Divide by the square root of 24.

4. Let's pretend you make $2500 a month. Subtract out your rent ($800), money for food ($500), health care costs ($150), car payment ($136), car insurance ($55), car gas ($260), gym membership ($35), and entertainment expenses ($225). How much do you have left to save?

5. Add 275 and 38 together; then raise that to the 3rd power and divide everything by 6.

6. Divide 258367 by 268. Add 245 and 897 to that number. Multiply the result by 4 raised to the 20th power.

7. Whoops! You forgot to add in a couple of bills this month, so go back to your calculations in activity 4 and subtract your cell phone bill ($63.24), your cable bill ($35.10) and a new pair of glasses this month ($180). And your rent just went up to $850 a month, so change that. Now how much can you save?

8. Multiply 42 by 84, divide that by the sum of 90, 266, and 35, and then raise that whole result to the 3rd power.

1.10 Objects

So there you have it—you have seen how R can be your calculator! Actually the reason I like R better than a calculator is that I have all the numbers printed out in front of me so I can check my math, but I can also go back and change numbers or add more later, just as in the application activity. But we will now move on to looking at R as more than a calculator. Any command that we tell R to do can be put into an object. We name the object ourselves. The line below shows how I take all my budget calculations from the application activity and put them into an object I'll call Savings.March.2010:

```
Savings.March.2010<-2500-
(850+500+150+136+55+260+35+225+63.24+35.10+180)
```

Notice that I use the sequence of two symbols, the less than sign (<) and the dash (-), together called the assignment operator, to show that whatever is on the right side is going into the object on the left side of the symbols. You can also use an equals sign here (=), which is of course easier to type, but the assignment operator may be helpful because it shows which way the assignment is going (the part on the right is assigned into the object named on the left). Venables, Smith, and R Development Core Team (2005) note that the equals sign is an equivalent to the assignment operator in most cases, but there are some exceptions. You will see me use both the equals sign and the alternate assignment operator throughout the book.

Notice that, when you type the line that calculates my savings for the month of March into R, you won't see anything. R won't return any answer until you ask it to. When you type in the name of the object, then you get the results of the calculation:

```
Savings.March.2010
[1] 10.66
```

This type of object has only one entry. But let's create another object that has several entries. I can't just enter the name of an object that is not defined, though, so I will first make a couple more objects for my budget:

```
Salary<-2500
Fixed.Expenses<-c(850,500,150,136,55,260,35,225)
Bills<-c(63.24,35.10)
Additional.Expenses.March.2010<-180
```

Notice that, in every case where I'm entering more than one number, I've used a "c" before the parentheses. The "c" stands for "concatenation" and is a built-in, primitive function of R (for more information about primitive functions of R, type `library(help="base")`). This is a function that combines its arguments to make a **vector**, which is basically a column of data if you think of data set up in a spreadsheet. Now I'll create a formula to calculate my savings every month without having to type in my fixed numbers every time:

```
Savings=Salary-(sum(Fixed.Expenses) + sum(Bills) +
sum(Additional.Expenses.March.2010))
```

Notice that I had to use a function, `sum()`, that sums up the numbers. If I didn't do this, I would have gotten as an answer a collection of numbers, not just one number (try it yourself; write the equation without the command `sum()` and see what the object summary returns).

Now R has stored that number in the object called `Savings`. If I want to see it, I have to ask R to show me what's in the object, by just typing its name:

```
Savings
[1] 10.66
```

Obviously, this is what we calculated before, but the point is that now each month I could, for example, just redefine my "Bills" category and my "Additional Expenses" category and then run the equation again to see how much I can save that month.

At this point you can now understand how you yourself could create a vector of numbers. You would simply use the `c()` command and enter a series of numbers separated by commas. Vectors can also consist of character strings as well as numbers. To tell R that these are not numbers, you need to insert double or single quotation marks around the character strings, like this:

```
MyFamily<-c('Andrea', 'Mark', 'Annette', 'Caroline')
MyFamily
[1] "Andrea" "Mark" "Annette" "Caroline"
```

If you forget the quotation marks, R thinks you are looking for previously defined objects (like those I defined in my budget calculation above), and gives you an error message:

```
MyFamily<-c(Andrea,Mark,Annette,Caroline)
Error: object 'Andrea' not found
```

You cannot combine numbers and character strings in the same vector. If you put numbers and strings into the same object, R will coerce everything to be of the same type, which here

is a character (we can tell because the parentheses are around the numbers as well as the names).

```
MyFamily<-c('Andrea',23, 'Mark',15,'Annette',45,'Caroline',18)
MyFamily
[1] "Andrea" "23" "Mark" "15" "Annette" "45" "Caroline" "18"
```

In R you will create objects. A vector is one column of data, consisting of numbers or character strings. A vector can have only one type or the other, not both in the same vector. Create a vector by naming the vector on the left side of the equation and then specifying on the right the content:

```
FavoriteAmericanIdols<-c('Constantine', 'Clay Aiken', 'Kelly Clarkson')
FavoriteNumbers<-c(25,8,17,22)
```

Use the c() concatenation function to combine the elements of the vector. A vector may also consist of mathematical operations on previously defined objects:

```
LuckyNumber=sum(FavoriteNumbers)
LuckyNumber
[1] 72
```

Deleted: together

1.11 Application Activities in Creating Objects

1. Create a numeric vector of the following numbers and call it "Years": 1976, 2003, 2009, 1900. Look at the vector you just made. Did it rearrange the order of the numbers?

2. Create a character vector with your own family's names in it and call it "MyFamily." Look at the vector you just made. Are the names listed with a single quote mark, double quote mark, or no quote mark?

3. Create a numeric vector that lists your monthly income called "Salary." Create another numeric vector that lists your monthly expenses called "Expenses." Create one more vector called "Budget" that subtracts the sum of your monthly expenses from your salary. Show your vectors.

1.12 Types of Data in R

This section will look at different types, or classes, of data in R. This is not just an academic exercise; the class of data in R will often determine how graphics will be displayed in the plot() command, how generic functions such as summary() treat the data, and whether you can use specific types of commands which may require that the data be in a certain format in order to work. It is thus important to understand some of the types of data that you will meet most frequently when using R.

If you import data, you create an object that R labels a **data frame**. A data frame is a collection of columns and rows of data, and the data can be numeric or character (remember that characters in R are symbols that have no numerical value). It only has two dimensions, that of rows and columns. All of the columns must have the same length. Most of the data you will work with is in data frames, and you can think of it as a traditional table. You can see an example of a data frame in Table 1.1. To create the view in Table 1.1, I simply asked

R to give me just the first few rows of data by using the head() command with the mcguireSR data set:

```
head(mcguireSR) #gives first few rows of the data frame
```

This data set comes from work by McGuire (2009) on how fluency of English as a second language learners is affected by a focus on language chunking. If you want to follow along, import this .csv file (it is comma delimited) and call it mcguireSR.

An object with just one column of numbers (if you think of the numbers arranged in a spreadsheet) creates a **vector**, which was explained previously in the online documents "Understanding the R environment. Objects" and "Understanding the R environment. Types of data in R." Some commands for robust statistics that you will see later in the book require arguments that are vectors. If you take just one column out of a data frame and turn it into an object with a new name, this is a vector. For example, one column of post-test data is taken out of the imported data set called mcguireSR:

```
mcguireSR$Post.test
 [1] 140.5 134.0 143.3 156.3 162.0 141.3 134.2 160.0 134.2 137.2 168.9 128.5
[13] 145.9 147.5 127.7 174.8 165.5 151.8 179.4
```

I then put these numbers into an object I name MPT (for McGuire post-test):

```
MPT=mcguireSR$Post.test
is.vector(MPT)
[1] TRUE
class(MPT)
[1] "numeric"
```

A command which asks if an object is a vector verifies that it is. A more general command class() tells you what type of data you have, but if it is a vector it will be classified as numeric or character. You can see a vector of data in Table 1.1. For the vector in Table 1.1, I just typed the data from the post-test into a vector by hand:

```
mcguireVector=c(140.5, 134, 143.3, 156.3, 162.0, 141.3)
class(mcguireVector)
[1] "numeric"
```

Another object in R that you may need for doing robust statistics is a **list**. A list is just a collection of vectors, which may be of different lengths and different types (this is different from a data frame, where all of the columns must be the same length). You can see an example of the mcguireSR data turned into a list in Table 1.1. The syntax for massaging the data into a list is a little complicated (don't worry too much about it here; I'll give you another example when you need to turn your data into a list; it's more important to just understand now how a list differs from a data frame or a vector):

```
mcguirelist=list() #this creates an empty list
mcguirelist[[1]]<-c('Control', 'Control', 'Experimental', 'Experimental', 'Experimental',
'Experimental')
#the designation [[1]] says that this is the first column of data. I wanted this column to
#consist of characters, not numbers, so I had to put a quote around each word
mcguirelist[[2]]<-c(140.5,134.0,143.3,156.3,162.0,141.3)
```

```
mcguirelist[[3]]<-c(126.60,140.38,103.80, 135,60,133.66,146.10)
class(mcguirelist)
[1] "list"
```

Table 1.1 Illustrations of R Data Types: Data Frames, Vectors, and Lists

Data Frame

```
         Group Post.test Pre.test
1      Control     140.5   126.60
2      Control     134.0   140.38
3 Experimental     143.3   103.80
4 Experimental     156.3   135.60
5 Experimental     162.0   133.66
6 Experimental     141.3   146.10
```

Vector (Just the Post-Test Data)

```
[1] 140.5 134.0 143.3 156.3 162.0 141.3
```

List

```
[[1]]
[1] "Control"      "Control"      "Experimental" "Experimental" "Experimental"
[6] "Experimental"

[[2]]
[1] 140.5 134.0 143.3 156.3 162.0 141.3

[[3]]
[1] 126.60 140.38 103.80 135.00  60.00 133.66 146.10
```

There are a number of other types of data in R, namely matrices, arrays, and factors. A **matrix** is similar to a data frame, but all the data must be either numeric or character. An **array** can have more than two dimensions, and looks like a list of tables. It is often used with categorical data. These first two data types are not used much in this book, so I won't explain them further. More information can be found in Appendix A, however.

The last additional type of data is the **factor**. A factor is a vector that is a categorical variable, and it contains a vector of integers as well as a vector of character strings which characterize those integers. This is a very useful data type to use with statistical procedures, but in most cases, if you import your data and your categorical variables are strings, R will automatically classify them as factors, so no further attention to this is needed on your part. For example, the imported mcguireSR data set has an independent variable Group, which contains character strings. If this data is summarized, because it is a factor, R can count how many people belong to each group.

```
mcguireSR$Group
 [1] Control Control Experimental Experimental Experimental Experimental
summary(mcguireSR$Group)
 Control Experimental
 2 4
class(mcguireSR$Group)
[1] "factor"
```

You can find out what type of data you have by using the class() command. Here is a brief summary of the types of data in R:

data frame: a traditional table with columns and rows
vector: one column of a traditional table, where items are numeric or character strings
matrix: a table where all variables are numeric or all variables are character
list: a collection of vectors, which differs from a table because the vectors can have different lengths.
array: a collection of tables
factor: a vector that has character strings which can be counted

Deleted: dataframe

Deleted: . D

1.13 Application Activities with Types of Data

1. Import and open the beq data set (it is an SPSS .sav file). Find out what type of data this is (use the class() command).

2. There are four variables in this beq data frame. Find out their names by typing names(beq). The variable CatDominance in the beq data set is a categorical variable. Find out if it is indeed a factor (remember that you can pick this variable out of the set by writing the name of the data set followed by a dollar symbol ($) and the name of the variable).

3. How many levels are there in the CatDominance variable? How many people in each level? Use the summary() command.

4. Referring back to the vectors you created in the application activity "Understanding the R environment.ApplicationActivity _Creating objects," find out what class of data your object Years and also MyFamily are.

1.14 Functions in R

There are many built-in functions of R that we will use in future sections. Here I will start with a few simple and basic ones so that you can see how functions work in R. They will not necessarily be useful later, as there are more complex functions that will include the simple functions we look at here, but they will serve as a basic introduction to functions. We'll start with the function sum(). The arguments of sum() are numbers which are listed inside the parentheses, with each number followed by a comma.

```
sum(34,10,26)
[1] 70
```

The function mean() will return the average of any numbers included in its arguments.

```
mean(34,10,26)
[1] 34
```

If you have imported a data set, such as the data frame dekeyser, you can find the mean score of the entire numeric vector with this function as well. In other words, you can put any numeric object inside the mean() function, not just single numbers.

```
mean(dekeyser$GJTScore)
[1] 157.3860
```

If you put a non-numeric vector, however, such as the factor Status, in the dekeyser data frame, you will get an error message.

```
mean(dekeyser$Status)
Warning in mean.default(dekeyser$Status) :
 argument is not numeric or logical: returning NA
[1] NA
```

Several other short and simple functions could be used on a numeric vector:

```
> length(dekeyser$GJTScore)
[1] 57
> min(dekeyser$GJTScore)
[1] 76
> max(dekeyser$GJTScore)
[1] 199
> range(dekeyser$GJTScore)
[1]  76 199
> length(dekeyser$GJTScore) #gives N
[1] 57
> min(dekeyser$GJTScore)  #minimum score
[1] 76
> max(dekeyser$GJTScore)  #maximum
[1] 199
> range(dekeyser$GJTScore) #minimum and maximum score=range
[1]  76 199
> sd(dekeyser$GJTScore)  #standard deviation
[1] 29.60258
> var(dekeyser$GJTScore)  #variance (=sd^2)
[1] 876.3127
```

There are many built-in functions in R. Here is a summary of some very simple ones.

sum()	add arguments together
mean()	gives average of arguments
length()	returns N of vector
min()	minimum score
max()	maximum score
range()	gives minimum and maximum
sd()	standard deviation
var()	variance

Deleted:
Deleted:
Deleted:
Deleted:
Deleted:
Deleted:
Deleted:
Deleted:

1.15 Application Activities with Functions

1. Import the SPSS data set DeKeyser2000.sav. Call it dekeyser. Verify that the variance of the GJTScore is just the standard deviation squared using the var() and sd() functions.

2. Calculate the average age of the participants in the dekeyser data frame (use the names() command to help you get the right name for this column of data). Use the function mean().

3. What is the oldest age in the deKeyser data set? Use the function max().

4. Import the SPSS data set LarsonHallPartial.sav. Call it partial. How many participants were there in this study (count the number using the length() function)? What were the mean and standard deviation of their accuracy on words beginning in R/L?

1.16 Manipulating Variables (Advanced Topic)

At some point after you have begun working with R, it is likely that you will want to manipulate your variables and do such things as combine vectors or group participants differently than your original grouping. However, since we have not begun any real work with statistics yet, this section may not seem pertinent and may be more confusing than helpful. I recommend coming back to this section as needed when R is more familiar to you. I

will also note that, for me personally, manipulating variables is the hardest part of using R. I will often go to SPSS to do this part, because the ability to paste and copy numbers is a lot more intuitive than trying to figure out the syntax for manipulating numbers in R.

1.16.1 Combining or Recalculating Variables

For this example we will use a data set from Torres (2004). Torres surveyed ESL learners on their preference for native speaking teachers in various areas of language teaching. This file contains data from 34 questions about perception of native versus non-native speaking teachers. Let's say that I am interested in combining data from five separate questions about whether native speaking teachers are preferable for various areas of language study into one overall measure of student preference. I want to combine the five variables of Pronunciation, Grammar, Writing, Reading, and Culture, but then average the score so it will use the same 1–5 scale as the other questions.

To do this in R Commander, first make sure the data set you want to work with is currently active and shown in the "Data set" box in R Commander (you will need to import it from the SPSS file Torres.sav; call it torres). Combine variables by using DATA > MANAGE VARIABLES IN ACTIVE DATA SET > COMPUTE NEW VARIABLE. You will see a computation box like the one in Figure 1.14 come up.

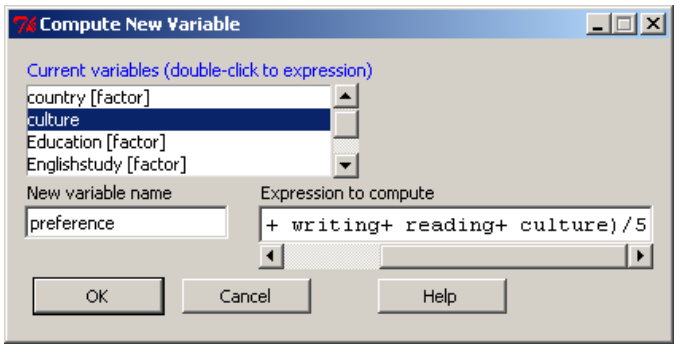
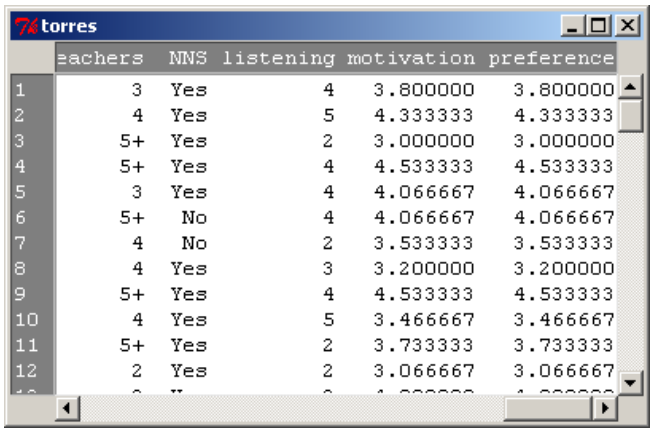


Figure 1.14 Computing a new variable in R Commander.

As you can see in Figure 1.14, I called the new variable preference. I moved the five variables that make up this new variable from the "Current variables" box into the "Expression to compute" box by double-clicking on them. After starting out with a parenthesis at the beginning, I added the variables one by one. After each variable I manually typed in a plus sign, and at the end of the entire string I typed in ")/5" to divide it all by 5. You can use the same symbols (such as "+," "*," "^") that were explained in the online document "Understanding the R environment.R as a calculator" to perform mathematical computations. Last of all you will want to make sure this worked right by opening up the "View data set" button on the R Commander interface and scrolling over to the end of the data set. Figure 1.15 shows how I have done this, and the column appears to be measured on a five-point scale.



	eachers	NNS	listening	motivation	preference
1	3	Yes	4	3.800000	3.800000
2	4	Yes	5	4.333333	4.333333
3	5+	Yes	2	3.000000	3.000000
4	5+	Yes	4	4.533333	4.533333
5	3	Yes	4	4.066667	4.066667
6	5+	No	4	4.066667	4.066667
7	4	No	2	3.533333	3.533333
8	4	Yes	3	3.200000	3.200000
9	5+	Yes	4	4.533333	4.533333
10	4	Yes	5	3.466667	3.466667
11	5+	Yes	2	3.733333	3.733333
12	2	Yes	2	3.066667	3.066667

Figure 1.15 Verifying that a new variable was correctly calculated in R Commander.

I would also ask for the range of scores to make sure I'm on the right track:

```
range(torres$preference)
[1] 2.266667 5.000000
```

Looks good! I'd also like to show you how commands can be executed on the R Console. As we go along I will first show you how to perform an action using R Commander, but then I will also explain the R code. At first you may not care about this, but as you become more experienced with R you might want to begin to move toward using more code, and so I want you to understand it. Note that the R code can be found in R Commander in the Script window. If you want to take the code that R Commander used for a certain command and tweak it a little, you can simply copy the code from the Script window to the console and paste it in.

Every time I give you R code, I will try to explain it in more detail. The box below first gives the command that was used to combine the variables. I then try to break down the command into its constituent parts and explain what each part is doing.

```
torres$preference <- with(torres, (pron+ grammar+ writing+ reading+ culture)/5)
```

torres$preference	This command creates and names the new variable preference.
<-	This symbol says to assign everything to the right of it into the expression at the left; you can also use an "equals" sign ("=").
with(torres, (expression))	with(data, expression, . . .) The with() command says to evaluate an R expression in an environment constructed from data.
(pron+grammar+ writing+ reading+ culture)/5	This is the arithmetic expression.

To Use Existing Variables to Calculate a New Variable

In R Commander, choose:

DATA > MANAGE VARIABLES IN ACTIVE DATA SET > COMPUTE NEW VARIABLE

In R, use the template:

torres$preference <- with(torres, (pron+ grammar+ writing+ reading+ culture)/5)

Formatted: Font: Italic
Formatted: Font: Italic
Formatted: Font: Italic
Formatted: Font: Italic
Deleted: ¶

1.16.2 Creating Categorical Groups

Sometimes you want to take the data and make categorical groups from it (you should be careful, though; making categorical groups when you have interval-level data is just throwing away information). To illustrate this process, let's look at data from DeKeyser (2000). DeKeyser administered a grammaticality judgment test to a variable of child and adult Hungarian L1 learners of English. DeKeyser divided the participants on the basis of whether they immigrated to the US before age 15 or after (this is his Status variable). But let's suppose we have a good theoretical reason to suspect that there should be four different groups, and we want to code the data accordingly. (I want to make it clear that I don't actually think this is a good idea for this particular data set; I'm just using it as an example.)

In R Commander, we can create new groups (also called recoding variables) by first making sure the data set we want is the active one in the "Data set" box in R Commander (if you have not already done so for a previous application activity, in order to follow along with me here you'll need to import the DeKeyser2000.sav SPSS file; name it dekeyser). Then pull down the DATA menu and choose MANAGE VARIABLES IN ACTIVE DATA SET > RECODE VARIABLES. A dialogue box will open (see Figure 1.16).

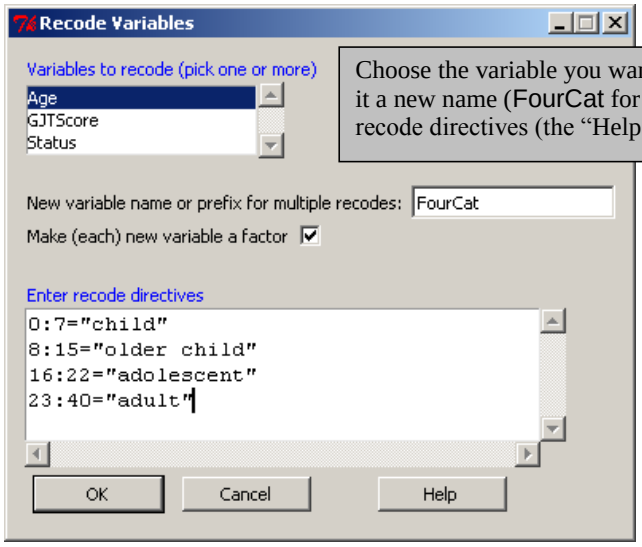


Figure 1.16 Recoding variables in R Commander.

After I pressed OK I looked at the data set and verified that this new variable had been created and seemed fine. The R code for this command is:

```
dekeyser$FourCat <- recode(dekeyser$Age,
 '0:7="child"; 8:15="older child"; 16:22="adolescent"; 23:40="adult"; ',
 as.factor.result=TRUE)
```

dekeyser$FourCat	This command creates and names the new variable FourCat.
<-	Assignment operator.
recode(data, expression, as.factor.result=TRUE)	The recode command gives instructions on how to restructure a numeric vector into one that is a factor. The expression part gives the recode specifications.
'0:7="child"; 8:15="older child"; 16:22="adolescent"; 23:40="adult"; '	This is the recode specification; note that the entire expression is enclosed in single quotes; each part of the recode directive is separated by a semi-colon; the expression 0:7="child" specifies that, if the number is 0 through 7, the label child should be attached in this factor.

If you want to change existing factor levels that are already categorical and named with non-numerical characters, you will need to put quotation marks around the existing names as well as the new names, as shown in the recoding done for the beqDom file (shown in Figure 1.17).

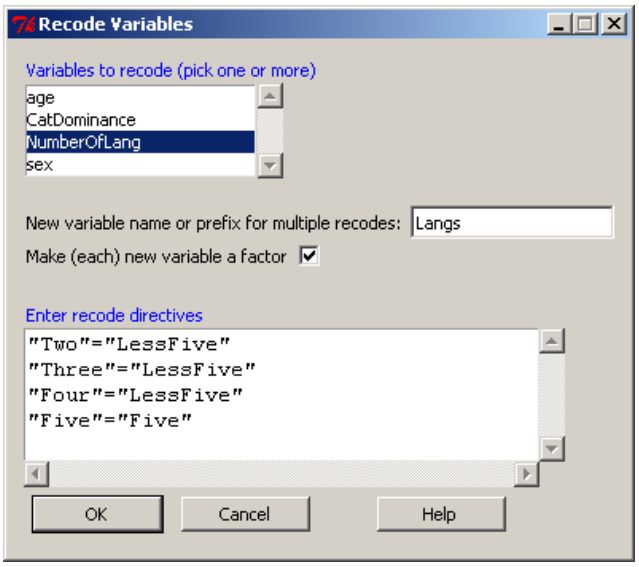


Figure 1.17 Recoding character variables in R Commander (factors with labels).

The R code for this command is:

```
beq$Langs <- recode(beq$NumberOfLang,
 '"Two"="LessFive"; "Three"="LessFive"; "Four"="LessFive"; "Five"="Five"; ',
   as.factor.result=TRUE)
levels(beq$Langs) #Use this to check that everything came out right
```

```
[1] "Five" "LessFive" #Looks good!
```

Creating Categorical Groups from Existing Data

In R Commander, choose:

DATA > MANAGE VARIABLES IN ACTIVE DATA SET > RECODE VARIABLES

In recode directives, enter the range of numbers to the left of the equals sign (separate range with a colon) and enter the name of the new category in quotes to the right of the equals sign. If you are recoding categories that already exist, put parentheses around character symbols to the left of the equals sign as well.

In R, use the template:

```
beq$Langs <- recode(beq$NumberOfLang,
'"Two"="LessFive"; "Three"="LessFive"; "Four"="LessFive";
"Five"="Five"; ',
   as.factor.result=TRUE)
```

Formatted: Font: Italic
Formatted: Font: Italic
Formatted: Font: Italic
Deleted:
Deleted:
Deleted: “
Deleted: "=“
Deleted: ";
Deleted: “
Deleted: "=“
Deleted: ";
Deleted: “
Deleted: "=“
Deleted: ";
Deleted: “
Deleted: "=“
Deleted: ";
Deleted: ‘
Formatted: Indent: First line: 0.16"
Deleted: ¶

1.16.3 Deleting Parts of a Data Set

Sometimes you may have a principled reason for excluding some part of the data set you have gathered. If you are getting rid of outliers, robust statistics (which I will discuss throughout this book) is probably the best way to deal with such a situation. However, there are other situations where you just need to exclude some part of the data set a priori. For example, in the case of the Obarow (2004) data set discussed in Chapter 11, “Factorial ANOVA,” some children who participated in the vocabulary test achieved very high scores on the pre-test. Children with such high scores would not be able to achieve many gains on a post-test, and one might then have a principled reason for cutting them out of the analysis (although you should tell your readers that you did this).

To cut out an entire column of data, make sure your data is the active data set and then use R Commander’s menu sequence DATA > MANAGE VARIABLES IN ACTIVE DATA SET > DELETE VARIABLES FROM DATA SET. To follow along with me here, import the Obarow data set (the SPSS file is called Obarow.Original.sav; name it obarow). I decide to cut the variable grade, so I choose it out of the list of variables. A prompt comes up to verify that I want to delete it, and I press OK.

The R code for this action is quite simple:

```
obarow$grade <- NULL
```

You just assign nothingness to the variable, and it ceases to exist.

There may also be times when you would like to exclude certain rows instead of delete entire variables. There is a menu command in R Commander to accomplish this: DATA > ACTIVE DATA SET > REMOVE ROW(S) FROM ACTIVE DATA SET. Here you would have a large number of choices as to what you would enter in the line labeled “Indices or quoted names of row(s) to remove” (see Figure 1.18). The “Help” button associated with this box takes you to the help

page titled “Extract or replace parts of a data frame.” A look at the examples on that page is quite interesting.

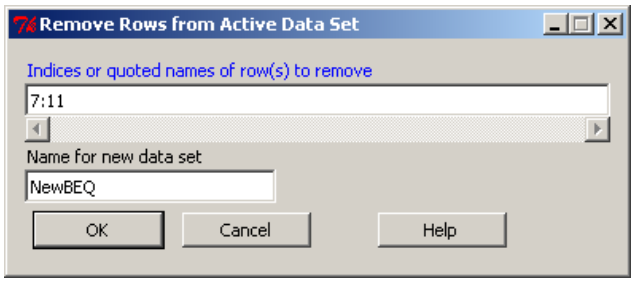


Figure 1.18 Removing a row from the active data set in R Commander.

If you want to remove one certain row, you can just put the number of that row in the box. Figure 1.18 shows a range of rows from 7 to 11 that will be excluded. I have given the data set a new name as well.

The R code for performing this action is simple as well, as long as you understand that in R you can specify rows and columns in brackets, with rows first and columns second. The R code for removing rows 7 through 11 from the obarow data set and creating a new one is:

```
NewObarow<-obarow[-c(7:11),]
```

R uses the minus sign (“-”) to show that it will subtract this subset of rows. The concatenation function, c(), is used whether or not there is more than one row or column being deleted. So the R command says to subtract out rows 7 through 11 (we know they are rows because they come before the comma) from all of the columns (we know this because nothing is specified; if you wanted to subtract out certain rows only from certain columns you would specify the columns after the comma.

Tip: Here’s a silly mnemonics device to help you remember which comes first, the row or the column:

Row, row, row your boat gently down the stream, toss your column overboard and listen to it scream. 

Row comes first and column comes second in my version of the nursery rhyme, so you can remember it works that way in the syntax too.

Formatted: Indent: First line: 0"

But what if you don’t know exactly which rows you want to delete? What if, instead, you want to remove any data that exceeds a certain number? To do this, you can use the subset() command. The following command puts a subset of the original data set into a file with a different name. In this command I get rid of any cases where the Pretest 1 score was over 18 points (out of 20 possible).

```
NewObarow<-subset(obarow, subset=pretest1<=18)
```

Notice that in the subset command I have used the comparison operators less than or equal to("<="). Other such operators in R include more than (">"), more than or equal to (">="), less than ("<"), equal to ("=="), or not equal to ("!=").

The subset() command can also be useful for rearranging data, not necessarily deleting it. This topic is covered in the next section.

Deleting Parts of a Data Set

In R Commander, for removing rows, choose:

DATA > ACTIVE DATA SET > REMOVE ROW(S) FROM ACTIVE DATA SET

For removing columns, choose:

DATA > MANAGE VARIABLES IN ACTIVE DATA SET > DELETE VARIABLES FROM DATA SET

In R, to remove columns, assign the column to the NULL operator:

```
obarow$grade <- NULL
```

To remove rows specify which rows to subtract out (and possibly rename your data):

```
NewObarow<-obarow[-c(7:11), ]
```

Formatted: Font: Italic
Formatted: Font: Italic
Formatted: Font: Italic
Formatted: Indent: First line: 0"
Deleted:
Deleted:
Formatted: Font: (Default) Times New Roman
Deleted: ¶ ¶

1.16.4 Getting Your Data in the Correct Form for Statistical Tests

Depending on how you have set up your data entry, there are two basic ways that you might have your data set up:

1. Data is split so that the results for each group for each variable are found in different columns. We'll call this the "**wide**" form (see Figure 1.19). This form of data already split by the categorical variables is often used for the robust statistics tests created by Wilcox (2005) that are used in this book.
2. All the data for one variable is in one column, and there is another column that codes the data as to which group it belongs to. Everitt and Dunn (2001) call this the "long" form because the columns will be longer in this case. This is the form used for ANOVA analysis (see Figure 1.20).

Here is an example of the data in the wide format. Let's say we are looking at the correlation between test scores of children and adults on regular and irregular verbs. We would have one column that represented the scores of the children on regular verbs, another column containing the scores of adults on the regular verbs, etc. In the wide format, we do not need any indexing (or categorical) variables, because the groups are already split by those variables (adult vs. child, regular vs. irregular verbs) into separate columns (see Figure 1.19; note that I just typed this data in by hand—it does not come from any data set. If you'd like to follow along with me, just type the initial data in Figure 1.19 in yourself and call the file verbs).

verbs

	ChildRegVerb	AdultRegVerb	ChildIrregVerb	AdultIrregV
1	14	13	14	15
2	13	15	15	15
3	15	15	11	15
4	15	13	15	14
5	13	8	14	15
6	8	13	14	15
7	13	13	14	13

Figure 1.19 Data in the "wide" format.

This data can be put into the long format (see Figure 1.20). In this case all of the scores are put into just one column, and there is another column (which is a factor) which indexes both the group variable and the verb regularity variable at the same time.

verbsLong

	Score	Group
1	15	AdultIrregV
2	15	AdultIrregV
3	15	AdultIrregV
4	14	AdultIrregV
5	15	AdultIrregV
6	15	AdultIrregV
7	13	AdultIrregV
8	13	AdultRegVerb
9	15	AdultRegVerb
10	15	AdultRegVerb
11	13	AdultRegVerb
12	8	AdultRegVerb
13	13	AdultRegVerb
14	13	AdultRegVerb
15	14	ChildIrregVerb
16	15	ChildIrregVerb
17	11	ChildIrregVerb
18	15	ChildIrregVerb
19	14	ChildIrregVerb
20	14	ChildIrregVerb
21	14	ChildIrregVerb
22	14	ChildRegVerb
23	13	ChildRegVerb
24	15	ChildRegVerb
25	15	ChildRegVerb
26	13	ChildRegVerb
27	8	ChildRegVerb
28	13	ChildRegVerb

Figure 1.20 Data in the "long" format.

With the help of R Commander, moving from one form to another is not too difficult. In R Commander I went to DATA > ACTIVE DATA SET > STACK VARIABLES IN ACTIVE DATA SET. There I picked all four variables by holding down the Ctrl button on my keyboard and right-clicking my mouse on each variable. I renamed the entire file verbsLong, renamed the

numeric variable Score and the factor variable that would be created Group. Figure 1.21 shows the "Stack Variables" dialogue box, and Figure 1.20 is the result from that command.

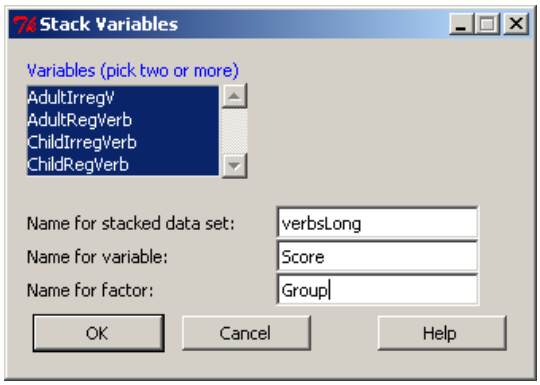


Figure 1.21 Going from wide form to long form in R Commander.

This process can of course be done using R code:

```
verbsLong <- stack(verbs[, c("AdultIrregV","AdultRegVerb ","ChildIrregVerb ",
"ChildRegVerb")])
names(verbsLong) <- c("Score", "Group")
```

verbsLong<-	Assign the result of what's on the right-hand side of the assignment operator to the object verbsLong.
stack(verbs[ ])	This command stacks separate vectors on top of each other and then creates an indexing column; here, it is told to work on the data set verbs.
[, c("AdultIrregV", "AdultRegVerb", "ChildIrregVerb", "ChildRegVerb")]	The data inside the brackets specifies that we want all of the rows from the data set verbs, and the four columns that correspond to the ones listed.
names(verbsLong)	The names on the right-hand side are assigned to the names dimension of the newly created verbsLong data frame.

For moving from a long format to a wide one, you can simply subset the original data set along the categorical variable or variables. In R Commander, choose DATA > ACTIVE DATA SET > SUBSET ACTIVE DATA SET. The tricky part comes in figuring out what to put in the box "Subset expression." This needs to be set up so that the level of the categorical variable is specified. Therefore, before you open the dialogue box to do this, make sure to find out what the levels are for your variable.

```
levels(VerbsLong$Group)
[1] "AdultRegVerb" "ChildRegVerb" "AdultIrregV" "ChildIrregVerb"
```

First, I click off "Include all variables" and select the variable of "Score" because I only want *one* column of data, not a column with Group and a column with Score. For the subset

expression, notice that I need to use double equals signs ("==") after the name of the variable ("Group") and that I need to spell and capitalize the name of the level exactly right, and put it in parentheses. Figure 1.22 shows the dialogue box and resulting data set.

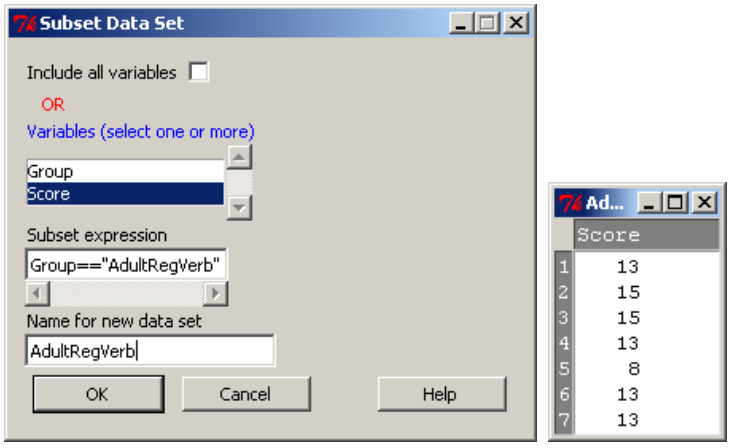


Figure 1.22 Subsetting a data set in the long form.

To continue with this process in R Commander, don't forget to change your active data set! If you try to subset the data set you just created (in my case, RegVerbAdults) your result will be null! Click in the "Data set" box along the top of the R Commander window to go back and choose the original data set. In the end I will have four different data sets that I will want to put together into one wide data set. Here the merge() command will not work, because the rows are numbered according to their original position in the long form of the data, and so there will be many NAs inserted (see Figure 1.19 to refer to what the wide form should look like). What is needed is to coerce the data together into a data frame and correctly label the columns. R code will do this:

```
Verbs<-cbind.data.frame(ChildRegVerb,AdultRegVerb,ChildIrregularVerb,
AdultIrregularVerb)
```

Verbs<-	Assign the result of what's on the right-hand side of the assignment operator to the object Verbs.
cbind.data.frame()	cbind stands for "column bind," so this ties together columns and makes them into a data frame.
ChildRegVerb,AdultRegVerb, ChildIrregularVerb, AdultIrregularVerb	The names of the four columns I want to bind together

```
dimnames(Verbs)[[2]]=c("ChildRegVerb","AdultRegVerb","ChildIrregVerb","AdultIrreg
Verb")
```

dimnames(Verbs)	Set the dimension names of an object, in this case the data frame Verbs.
[[2]]	Specifies that the dimension should be the columns (remember, rows come first, and then columns).
c("ChildRegVerb","AdultRegVerb "	The concatenation command c()

"ChildIrregVerb", "AdultIrregVerb")	contains the names I want appended to the columns. Note these must have quotes around them!

If you have followed my computations, you can verify for yourself at this time that the data set Verbs looks just as it started out in Figure 1.19. One major problem with this process is that, to make a data frame, the vectors you are binding together must be exactly the same length. If they are not, you will get an error message that tells you the vectors have differing numbers of rows. In most cases, you actually don't need to bind the subsetted vectors together—as a single vector they will perform in the statistical command the way you intended.

To subset using R Commander, use the subset() command. Here is an example of one I used with this data.

```
ChildRegVerb <- subset(VerbsLong, subset=Group=="ChildRegVerb",
select=c(Score))
```

ChildRegVerb<-	Assign the result of what's on the right-hand side of the assignment operator to the object ChildRegVerb.
subset()	Take a subset of the first argument.
VerbsLong	The name of the data set I want to subset
subset=Group=="ChildRegVerb"	Name the level of the variable you want to subset out. Be careful to use the double equals signs and the quotes around the name of the level; otherwise you'll get an error message.
select=c(Score)	Include this if you don't want to take all of the columns from the original data set. Here I only wanted the column with the Score.

In summary, manipulating data from the wide form to the long or vice versa is not impossible in R, but it is a process requiring a number of steps. Certainly if you cut and paste data this may be a much easier process in another program that works with spreadsheets, such as Excel or SPSS, depending on how large your data set is.

1.17 Application Activities for Manipulating Variables

1.17.1 Combining or Recalculating Variables

1. Use the torres data set. From this data set, create a new variable called OralPreferences by combining the variables of speaking and listening preference for a native speaker teacher. Don't forget to average the two scores so the result stays on a 1–5 scale. What is the range of scores for this new variable? Find out by using the function range() on the new variable.

2. Use the data set mcguireSR (this is a .csv file, comma delimited, if you did not import it earlier when using the online document "Understanding the R environment.Manipulating Variables_Advanced Topic"). Create a variable called gainscore by subtracting the pre-test from the post-test scores. What is the mean score for this variable?

3. Use the data set partial (this is an SPSS file, called LarsonHall.partial.sav; import it now, if you have not imported it before). The variable of R_L_accuracy is not measured in its raw

form as a score out of 7. Convert the scores to percentages and call the new variable PercentAccuracy. What is the maximum percentage achieved?

1.17.2 Creating Categorical Groups

1. Using the same beq data set as was used in the section "Combining or recalculating variables" in the online document "Understanding the R environment.Manipulating Variables_Advanced Topic" (or import it if you have not previously), recode the variable NumberOfLang so there are only two levels: those with two or three languages (call them "minorglots") and those with four or more languages (call them "majorglots"). Call your new variable Glots. Use the summary() command to see how many participants are in each of your new categories (you might want to reimport this file (BEQ.Dominance.sav) so that all of the cases will be there, since if you were following along with me in the text you might have deleted some cases!).

2. Use the mcguireSR data set (import as a text file; it is comma-delimited). Currently the participants are divided into whether they were in the experimental or control group. But let's say you decided you wanted to divide them into slow and fast speakers, according to their fluency scores on the pre-test (again, I don't think this is a good idea, but it's just for practice in manipulating variables!). Divide the speakers by calling those lower than the mean score "slow" and those at the mean or higher "fast." Name your new variable rate. Verify your variable has two levels by typing the name of the new variable.

3. Use the torres data set (import SPSS file Torres.sav), and the variable labeled beginner. This indicates whether each person prefers a NS teacher for a beginning-level language learner (where 5 = strongly prefer, 3 = neither prefer nor dislike and 1 = strongly dislike). Categorize participants into those who have strong opinions (both 5s and 1s), moderate opinions (2s and 4s), and neutral (label them as "strong," "moderate," and "neutral"). Call this new variable TypeOfPreference. Which type of participant predominates? Use the summary() command to see how many participants are in each category.

1.17.3 Deleting Parts of a Data Set

1. Working with rows and columns. First, let's do some of the examples with data that are found in R itself. These examples use a data set that comes built into R called swiss which has data on Swiss fertility and socioeconomic indicators for various cities (use the command library(help="datasets") to see a list of all data sets that are included in R). This is a rather large data frame, so let's first cut it down to make it more manageable. Type:

```
sw<-swiss[1:5, 1:4]
sw #look at your data now
```

Now we will work with the data set sw.

Perform the following commands, and explain what happens for each one, following the example in (a):

a. sw[1:3] #shows first 3 columns, all rows
b. sw[1:3,] #
c. sw[,1:3]
d. sw[4:5, 1:3]
e. sw[[1]]
f. sw[[5]]

g. sw["C",]

2. In activity 1 you created a new data frame called sw, with four variables. Remove the variable Examination.

3. Use the data set ChickWeight (this is a built-in R data set). This data looks at the weight versus age of chicks on different diets. How many data points are there? Find out using the length() command with any variable in the data set (don't just use the name of the data set itself, or it will only be 4!). Create a new variable called ChickDiet that excludes chicks that received Diet #4. How many participants are in the ChickDiet data set?

1.17.4 Getting Data in the Correct Form for Statistical Tests

1. The dekeyser data set is currently in the long form (you could look at it by using the command head(dekeyser) to see just the first six rows). Subset it into the wide form so there are only two columns of data, one for participants "Under 15" and one for those "Over 15." The resulting vectors are of different lengths, so do not try to combine them. How many entries are in each of the two subsetted vectors (you can use the length() command, or the Messages section of R Commander will tell you how many rows the new subset contains)?

2. Import the Obarow.Story1.sav SPSS file (call it OStory). This file is in the long form. Subset it into four columns of data, each one with results on the first gain score (gnsc1.1) for each of the four possible conditions (± pictures, ± music) listed in the Treatment variable. How many entries are in each of the four subsetted vectors (you can use the length() command, or the Messages section of R Commander will tell you how many rows the new subset contains)?

3. Open the .csv text file called Effort. This is an invented data set comparing outcomes of the same participant in four different conditions. It is set up in the wide form. Change it to the long form so there are only two columns of data, and call it EffortLong; call one column Score and the other Condition. What is the average score over the entire group (this is not really something you'd be interested in, but it's just a way to check you've set up the file correctly)?

4. Open the .csv text file called Arabic. This is an invented data set that looks at the results of ten participants on four phonemic contrasts in Arabic, and also records the gender of each participant. It is set up in the wide form. Change it to the long form so there are only two columns of data that are indexed by the type of contrast that was used, and call it ArabicLong. Call one column Contrast (as in the phonemic contrast) and the other Score. What is the average score over the entire group (this is not really something you'd be interested in, but it's just a way to check you've set up the file correctly)?

1.18 Random Number Generation

Random number generation may be useful when conducting experiments. When you are starting to do an experiment, you may want to randomize participants, sentences, test items, etc. You can generate random numbers in R by simply typing .Random.seed in the R console:

```
Random.seed
```

```
  [1]         403         352   647596969   505200444   240961364  -405956513
  [7]  1383978883  -757042607 -1537940647  1335279341 -1415915721  -553350035
 [13]  1206833492  1544732470    61259225 -1901511605  1941367673   144512002
 [19]  -133901986   704475269  1373605742   946416810 -1594397076  1611783465
 [25]  2099204935  1399887620   387066228  1563494368  1212353276  1355307236
 [31]  1928977319   598844249  1762831767  1393242963  1627814881  -706537856
```

Random single integers can be drawn from this distribution by simply going across the row: 4, 0, 3, 3, 5, 2, etc. Random numbers between 0 and 99 can be drawn by going across the row in pairs: 40, 33, 52, 64, etc.