

# Handout: Introduction to **R**

Dave Armstrong  
University of Wisconsin – Milwaukee  
Department of Political Science

e: [armstrod@uwm.edu](mailto:armstrod@uwm.edu)

w: [www.quantoid.net/ICPSR.php](http://www.quantoid.net/ICPSR.php)

## Contents

<b>1</b>	<b>Basics: What is R and how Does it Work?</b>	<b>2</b>
1.1	Using R . . . . .	4
<b>2</b>	<b>Assigning Output to Objects</b>	<b>4</b>
<b>3</b>	<b>Vectors and Matrices</b>	<b>5</b>
3.1	Matrix Math . . . . .	7
<b>4</b>	<b>Reading in your Data</b>	<b>8</b>
4.1	SPSS . . . . .	9
4.1.1	Data Types in <b>R</b> . . . . .	10
4.2	Stata . . . . .	10
<b>5</b>	<b>Accessing Data</b>	<b>11</b>
5.1	Attaching . . . . .	12
5.2	Recoding and Adding New Variables . . . . .	12
5.3	Missing Data . . . . .	15
5.3.1	How <b>R</b> Deals with Missing Data . . . . .	17
5.4	Getting Help . . . . .	17

Rather than slides, I am going to work through one large handout. This series of workshops should introduce you to many of the most commonly-used features of **R**. Throughout the workshops, I will provide some reading and exercises from John Fox and Sanford Weisberg's (2011) *An R and S-Plus Companion to Applied Regression*. This is a great book for beginners as it starts from no **R** knowledge and moves through linear models. This is where I started when I learned **R**.

There are a few other books that you may want to take a look at:

- *Modern Applied Statistics with S* (?) is probably one of the most recommended books for **R**. This is more technical and comprehensive than the Fox's book. As such, it is a good reference, but is probably not the best place to start for the absolute beginner.
- *The R Book* (?) is, again, a nice reference. It is 922 pages of relatively broad, not especially deep **R** knowledge. It will tell you how to run a bunch of models in **R**.
- *R Graphics* (?) provides a nice overview of all of the various graphics systems provided by **R** (there are two or three of them).
- *Lattice* (?) is narrower than Murrell's book as it only covers the Lattice graphics system, but does so comprehensively.
- *R for Stata Users* (?) is designed to get Stata users over to **R**.
- *Data Analysis Using Regression and Multilevel/Hierarchical Models* (?) has lots of **R** code and is a nice introduction to LMs and GLMs as well as hierarchical models.
- *Data Manipulation with R* (?) helps tackle what many find to be the most challenging tasks in **R** – data management.
- *Linear Models with R* (?) and *Extending the Linear Model with R* (?) are both good mixes of theoretical discussion and application with **R**.

There are many more books about **R**, some more statistical in nature and some more substantive, but these will work for our purposes.

## 1 Basics: What is **R** and how Does it Work?

**R** is an object-oriented statistical programming environment. It remains largely command-line driven.<sup>1</sup> **R** is open-source (i.e., free) and downloadable from <http://www.cran.r-project.org>. Click the link for **R Binaries** on the right and then choose the link that is appropriate for your operating system. Then, click on the link for **base** and then the link for **R-#.#. #-os.exe** (At the writing of this document, the newest Windows (32

---

<sup>1</sup>There are a couple of attempts at generating point-and-click GUIs for **R**, but these are almost necessarily limited in scope and tend to be geared toward undergraduate research methods students. Some examples are RCommander and SciViews.

bit) version is `R-2.13.1-win32.exe`). You will be asked to download a file. Once it is done, double-click on the resulting file and that will guide you through the installation process. There are some decisions to be made, but if you're unsure, following the defaults is generally not a bad idea. In Windows, you have to choose between MDI mode (Multiple Document Interface) where graphs and help files open in their own windows or SDI mode where graphs and help files open as sub-windows in the **R** window.

Like Stata, **R** has an active user-developer community. This is attractive as the types of models and situations **R** can deal with is always expanding. Unlike Stata, in **R**, you have to load the packages you need before you're able to access the commands within those packages. All openly distributed packages are available from the Comprehensive **R** Archive Network, though some of them come with the Base version of R. To see what packages you have available, type `library()`. When you download **R**, you will have to install many of the packages we use over the next few days. You can do this by typing `install.packages(name)` where `name` is the name of the package to be downloaded in double-quotes. For example - `install.packages('gmodels')`. Then, you'll be asked to choose a CRAN mirror (CRAN is the Comprehensive R Archive Network). You may pick any one, but generally people pick one that's geographically close to them.

Open **R** on your computer. As you can see, there are not many buttons here and very few menus in the **R** window. The easiest way to interact with **R** is to type in commands at the caret (`>`). Though this way is easy, it is less conducive to saving work. Just as in Stata, you can interact with the program by using an **R**-enabled text editor (akin to the do file editor). There are a few choices here. Emacs with its Emacs Speaks Statistics plug-in is a relatively common choice.<sup>2</sup> One of the benefits of this editor is that it is free. One potential down-side is that the up front costs are a bit higher than some are willing to bear. There is another editor called Tinn that has an **R** plug-in. This is a reasonably good editor and is free, but is only for Windows.<sup>3</sup> On Windows, I use WinEdt - which is a general purpose text editor that also has an **R** plug-in available. This is not free(it is \$40 (US) for educational purposes), but it fit my needs (it is also a front-end for  $\LaTeX$ ) so I use it. If you download and install WinEdt from <http://www.winedt.com/>, then install the **R** package `RWinEdt` by typing the following in **R**: `install.packages('RWinEdt')`. Follow the instructions, then once it is installed, if you type `library(RWinEdt)` in **R**, it will pop up an **R**-enabled instance of WinEdt. There should be some **R** buttons in the toolbar at the top of the page. You should notice toolbar buttons `Hist`, `Paste`, `Source` and `Script` - all with **R** icons. Now, we're ready to start working in **R**. If you're working on a Mac, you can use TextWrangler, though I use TextMate. Again, this is a proprietary piece of software, but has worked very nicely as front-end for both **R** and  $\LaTeX$ . You can learn more about Textmate from <http://macromates.com/>. There is a nice page about various editor options and their platform availability here [http://www.sciviews.org/\\_rgui/projects/Editors.html](http://www.sciviews.org/_rgui/projects/Editors.html).

---

<sup>2</sup>Information on this program can be found at: <http://stat.ethz.ch/ESS/>.

<sup>3</sup>Information on this editor and functionality can be found here: <http://www.sciviews.org/Tinn-R/>.

## 1.1 Using R

R is a case-sensitive environment, so be careful how you name and access objects in the space.

There are a few tips that don't really belong anywhere, but are nonetheless important, so I'll just mention them here and you can refer back when they become relevant.

- You can return to the command you previously entered by hitting the “up arrow” (similar to Page Up in Stata).
- You can find out what directory R is in by typing `getwd()`.
- You can set the working directory of R by typing `setwd(path)` where `path` is the full path to the directory you want to use. The directories must be separated by forward slashes / and the entire string must be in quotes (either double or single). For example: `setwd("C:/documents and settings/dave/desktop")`
- To see the values in any object, just type that object's name into the command window and hit enter.

## 2 Assigning Output to Objects

R can be used as a big calculator. By typing `2+2` into R, you will get the following output:

```
> 2+2
[1] 4
```

After my input of `2+2`, R has provided the output of 4, the evaluation of that mathematical expression. R just prints this output to the console. Doing it this way, the output is not saved *per se*. Notice, that unlike Stata, you do not have to ask R to “display” anything in Stata, you would have to type `display 2+2` to get the same result.

Often times, we want to save the output so we can look at it later. The assignment character in R is `<-` (the less-than sign directly followed by the minus sign). You may hear me say “X gets 10,” in R, this would translate to

```
> X <- 10
> X
[1] 10
```

Since R is “object-oriented,” you can assign the result of any operation to an object. For instance, you could say:

```
> X <- 4+4
> X
[1] 8
```

Now the object X contains the evaluation of the expression `4+4` or 8. We see the contents of X simply by typing its name at the command prompt and hitting enter. In the above command, we're assigning the output (or result) of the command `4+4` to X.

### 3 Vectors and Matrices

We can make a vector of numbers by combining a bunch of numbers together using the `c()` function. This collects the numbers together in a single object.

```
> Y <- c(1,2,3,4)
> Y
```

```
[1] 1 2 3 4
```

We can also do math on vectors. For example, we can add a scalar and it will add the scalar to each element of the vector:

```
> Y +3
```

```
[1] 4 5 6 7
```

It is easy to make, manipulate and perform mathematical operations on matrices in **R**. Unlike Stata, where matrix routines are split between the native matrix operations and Mata, (almost) all of **R**'s matrix routines are available in the base package. You can create a matrix in **R** with the `matrix()` command as follows:

```
> mat1 <- matrix(c(1,2,3,4), ncol=2)
> mat1
```

```
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Matrices are stored in **R** in “column-major order” unless specified otherwise. This means that **R** will fill up the first column, then the second, then the third etc... until all of the data have been used. If you want, you can change this to “row-major order” as follows:

```
> mat1 <- matrix(c(1,3,2,4), ncol=2, byrow=T)
> mat1
```

```
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

You can access and use the elements of the matrix by doing the following `matrix[r,c]`, where `matrix` is the name of the matrix, `r` represents the row(s) desired and `c` represents the columns desired. For instance, you could take the upper-left element from matrix `mat1` as follows:

```
> mat1[1,1]
```

```
[1] 1
```

You could take the first row of matrix `mat1` in a couple of different ways:

```
> mat1[1,1:2]
```

```
[1] 1 3
```

```
> mat1[1, ]
```

```
[1] 1 3
```

We can either explicitly give the minimum and maximum column numbers as in the first method above or we can give them implicitly by leaving the column designation blank, which will return all columns.

We can also make a matrix by “column-binding” two vectors together:

```
> vec1 <- c(1,2)
```

```
> vec2 <- c(3,4)
```

```
> cbind(vec1, vec2)
```

```
      vec1 vec2
[1,]    1    3
[2,]    2    4
```

Or by “row-binding” two vectors together:

```
> vec3 <- c(1,3)
```

```
> vec4 <- c(2,4)
```

```
> rbind(vec3, vec4)
```

```
      [,1] [,2]
vec3    1    3
vec4    2    4
```

The row and column names of a matrix can be accessed and changed with the `rownames()` and `colnames()` commands:

```
> mat3 <- rbind(vec3, vec4)
```

```
> rownames(mat3)
```

```
[1] "vec3" "vec4"
```

```
> colnames(mat3)
```

```
NULL
```

```
> rownames(mat3) <- c("row1", "row2")
```

```
> colnames(mat3) <- c("col1", "col2")
```

```
> mat3
```

```
      col1 col2
row1    1    3
row2    2    4
```

### 3.1 Matrix Math

To multiply the matrix by a scalar you can do the following:

```
> mat1 * 3

      [,1] [,2]
[1,]    3    9
[2,]    6   12
```

The matrix multiplication command is `%*%`, as in:

```
> mat2 <- matrix(c(5,6,7,8), ncol=2)
> mat2
```

```
      [,1] [,2]
[1,]    5    7
[2,]    6    8
```

```
> mat1
```

```
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
> mat1 %*% mat2
```

```
      [,1] [,2]
[1,]   23   31
[2,]   34   46
```

A couple of other matrix operations may be handy, especially for those in linear models classes:

- Transpose - to take the transpose of a matrix `mat`, simply do `t(mat)`.

```
> mat1
```

```
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
> t(mat1)
```

```
      [,1] [,2]
[1,]    1    2
[2,]    3    4
```

- Inverse - to take the inverse of a square matrix, you can use the `solve()` command:

```

> mat3 <- mat1 %*% t(mat1)
> mat3

      [,1] [,2]
[1,]   10   14
[2,]   14   20

> solve(mat3)

      [,1] [,2]
[1,]  5.0 -3.5
[2,] -3.5  2.5

```

## 4 Reading in your Data

Before we move on to more complicated operations and more intricacies of dealing with data, the one thing everyone wants to know is - “How do I get my data into **R**?” As it turns out, the answer is - “quite easily.” There are a number of routines that can read in many different data formats. The two most common in social science are probably SPSS and Stata, though there are certainly others. To enable these commands, you will need to load the `foreign` library in the following way:

```
> library(foreign)
```

**R** will generally produce no output if it works.<sup>4</sup> If there is output, it will perhaps be telling you that it can’t find the library, but this is rare, especially with `foreign` because it is distributed with the base installation of **R**.

To see what commands are available in the `foreign` package, you can type `help(package='foreign')`. That will produce a file, much of which is reproduced below:

<code>data.restore</code>	Read an S3 Binary or data.dump File
<code>lookup.xport</code>	Lookup Information on a SAS XPORT Format Library
<code>read.arff</code>	Read Data from ARFF Files
<code>read.dbf</code>	Read a DBF File
<code>read.dta</code>	Read Stata Binary Files
<code>read.epiinfo</code>	Read Epi Info Data Files
<code>read.mtp</code>	Read a Minitab Portable Worksheet
<code>read.octave</code>	Read Octave Text Data Files
<code>read.spss</code>	Read an SPSS Data File
<code>read.ssd</code>	Obtain a Data Frame from a SAS Permanent Dataset, via <code>read.xport</code>

---

<sup>4</sup>The possible exception here is if the library was built under a different version of **R** than the version you’re using. Often, these warnings are not critical and all of the commands will work as intended. If there is a major conflict that results in the commands not working, you will likely get an error message rather than a warning.

<code>read.systat</code>	Obtain a Data Frame from a Systat File
<code>read.xport</code>	Read a SAS XPORT Format Library
<code>write.arff</code>	Write Data into ARFF Files
<code>write.dbf</code>	Write a DBF File
<code>write.dta</code>	Write Files in Stata Binary Format
<code>write.foreign</code>	Write Text Files and Code to Read Them

The dataset we'll be using here has three variables - `x1`, (a numeric variable), `x2` (a labeled numeric variable [0=none, 1=some]) and `x3` a string variable ("no" and "yes"). I've called this dataset `r_example.sav` (SPSS) and `r_example.dta` (Stata).

**R** has lots of different data structures available (e.g., arrays, lists, ect...). The one that we are going to be concerned with right now is the *data frame*; the **R** terminology for a dataset. A data frame can have different types of variables in it (i.e., character and numeric). It is rectangular (i.e., all rows have the same number of columns and all columns have the same number of rows. There are some more distinctions that make the data frame special, but we'll talk about those later.

## 4.1 SPSS

The command in **R** for reading in spss data files into **R** is, `read.spss`. You can see the options for this command by typing `help(read.spss)`. The first argument is the name of the dataset. If the dataset is in **R**'s current working directory, then only the file name is needed. If the files is not in **R**'s working directory, then we have to put the full path to the dataset. Either way, the full path or the file name, both have to be in either double or single quotes. There are two other options that we really care about here: `to.data.frame` and `use.value.labels`. These are both *logical* arguments, which means that they take one of two values (T)RUE or (F)ALSE. Generally, we want these to be TRUE because we want the resulting object to be a data frame (as opposed to a list, which is the default) and we want to use the value labels for variables that were used in SPSS. We can put the output of this in an object called `spss.dat`.

```
> spss.dat <- read.spss("r_example.sav",
+   to.data.frame=T,
+   use.value.labels=T)
```

You may get a couple of warnings here (more likely on Windows than the Mac). If so, they are alerting us that there were string variables, (remember, `x3` was a character string variable). We will see the consequences of this in a second.

To see what the data frame looks like, you simply type the name of the object at the command prompt and hit enter:

```
> spss.dat

  x1  x2  x3
1   1 none yes
2   2 none no
```

3	3	some	no
4	4	none	yes
5	3	none	no
6	4	none	yes
7	1	some	yes
8	2	some	yes
9	5	some	no
10	6	none	no

### 4.1.1 Data Types in R

This is a convenient time to talk about different types of data in **R**. There are basically three different types of variables - numeric variables, factors and character strings.

- Numeric variables would be something like GDP/capita, age or income (in \$). Generally, these variables do not contain labels because they have many unique values. Dummy variables are also numeric with values 0 and 1. **R** will only do mathematical operations on numeric variables (e.g., mean, variance, etc...).
- Factors are variables like social class or party for which you voted. When you think about how to include variables in a model, factors are variables that you would include by making a set of category dummy variables. Factors in **R** look like numeric variables with value labels in either Stata or SPSS. That is to say that there is a numbering scheme where each unique label value gets a unique number (all non-labeled values are coded as missing). Unlike in those other programs, **R** will not let you perform mathematical operations on factors.
- Character strings are simply text. There is no numbering scheme with corresponding labels, the value in each cell is simply that cell's text, not a number with a corresponding label like in a factor.

When **R** reads in data from SPSS, it converts any string variables (like `x3`) to factors.

## 4.2 Stata

The basic operations here are pretty similar when reading in Stata datasets. The only difference is there is a different command - `read.dta`. You can see what the optional arguments are for the function by typing `help(read.dta)`. There are a couple of differences here. There is no `to.data.frame` argument because this command only produces a data frame (i.e., there is no option to produce some other data structure, like a list). There is also no `use.value.labels` command. In `read.dta`, the corresponding command is `convert.factors`. This is also a logical function (taking value either `TRUE` or `FALSE`). When the argument is `TRUE` (the default), any variable with value labels is converted to a factor. When it is `FALSE`, all variables are read in as numeric. Both options are useful in the right context, so don't forget about this.

When **R** reads in a factor (either from Stata or SPSS), the behavior is not exactly what you might want. Rather than preserving the numbers behind the value labels, it

converts them to consecutive integers starting with 1. So, this will be suboptimal if you need not only the labels, but the original numbers. We'll talk about how to get around this a bit later.

The only other real difference here is that character strings in a Stata dataset remain character strings when read into **R** (unlike in SPSS where they get converted to factors).

```
> stata.dat <- read.dta("r_example.dta",
+                       convert.factors=T)
> stata.dat

  x1  x2 x3
1   1 none yes
2   2 none no
3   3 some no
4   4 none yes
5   3 none no
6   4 none yes
7   1 some yes
8   2 some yes
9   5 some no
10  6 none no
```

There are obviously many other types of data you can read in and the set of commands required are quite similar to the ones I looked at above. If you have trouble with reading in a different data type, please ask either me or one of the computer consultants and we can get it figured out.

## 5 Accessing Data

Once the data are read in, you need to do things with it. We saw above that you can look at the data by simply by typing the name of the data frame at the command prompt and hitting enter. This works well for small datasets, but may not work all that well for large datasets.

If we want to see just one variable (let's say **x2**), we can do this in a couple of different ways. The most common is to use the **\$** to access an element of the data frame:

```
> stata.dat$x2

[1] none none some none none none some some some none
Levels: none some
```

Notice, that this shows us the values of this variable and the **Levels** - which are the unique values of this factor. A factor is the only type of variable that has a **Levels** attribute. So, if you see a variable that has this **Levels** statement, then you know it's a factor. You could also use the **[[ 'varname' ]]** construct to get a variable from a dataset:

```
> stata.dat[["x2"]]

[1] none none some none none none some some some none
Levels: none some
```

## 5.1 Attaching

Another way to make the data accessible is to **attach** it. You can do this by typing `attach(stata.dta)`. What this does is it places the data set in **R**'s *search path*. You might ask - what is the search path? Whenever you load a library, **R** adds that library to its search path. What this means is that it will now look in that library for commands that you issue. When you read in a dataset, it exist in the **R** workspace (i.e., it is an object you can use), but it is not in the search path. That is to say, if you just type the variable name in **R**, you will not get the “right” output. The reason is that **R** doesn't know that it is supposed to look there for the file. By attaching the dataset, **R** now knows that to look in that data frame (along with all of the packages attached) for the variable name. Here's an example:

```
x2
Error: object "x2" not found

> attach(stata.dat)
> x2

[1] none none some none none none some some some none
Levels: none some

> detach(stata.dat)
```

It might seem that attaching your dataset is the right thing to do and many people proceed this way. One of the strengths of **R** is that you can have lots of datasets open and in the workspace at the same time. However, if you have ten datasets and they are all attached, if there are any overlaps in the name (e.g., two of the datasets contain the variable `age`, then when you type `age` at the command prompt and hit enter, **R** will be taking the `age` variable from the first dataset on the search path containing that variable name. Further, and this seems to be a relatively recent development, variables can be masked by functions. For example, if you have a dataset with a variable called “class”, **R** won't know that you want the variable “class” rather than the function `class()`. For this reason, I will caution you against attaching any data and I will almost always use the `$`. You may encounter `attach()` so I wanted you to know what it is and why you might not want to use it.

## 5.2 Recoding and Adding New Variables

To demonstrate a couple of the features of **R**, we will add a variable to the dataset. Let's add a dummy variable that has zero for the first five cases and one for the last five cases. Unlike SPSS and Stata, there's not a particularly good spreadsheet-type data editor in **R**. For us, it is easier to make an object that looks the way we want, and then append that object to the dataset. If this is the strategy we adopt, first we need to make the object. What we want is a string of numbers (five zeros and five ones). To do this, we need to use **R**'s concatenate function, `c()`. I'll show this to you, then we'll discuss.

```
> x4 <- c(0,0,0,0,0,1,1,1,1,1)
> x4
```

```
[1] 0 0 0 0 0 1 1 1 1 1
```

What this did is make *one* object, called `x4` that is a string of numbers as above. Specifically, this is a vector with a length of ten (that is, it has ten entries). Now, we need to assign a new variable in the dataset the values of `x4`. We can do this as follows:

```
> stata.dat$x4 <- x4
> stata.dat
```

```
   x1  x2  x3 x4
1   1  none yes  0
2   2  none no   0
3   3  some no   0
4   4  none yes  0
5   3  none no   0
6   4  none yes  1
7   1  some yes  1
8   2  some yes  1
9   5  some no   1
10  6  none no   1
```

Recoding and making new variables that are functions of existing variables are two relatively common operations as well. These are relatively easily done in **R**, though perhaps not as easily as in Stata and SPSS. First, generating new variables. As we saw above, we can generate a new variable simply by giving the new variable object in the dataset some values. We can also do this when creating transformations of existing variables. For example:

```
> stata.dat$log_x1 <- log(stata.dat$x1)
> stata.dat
```

```
   x1  x2  x3 x4  log_x1
1   1  none yes  0 0.0000000
2   2  none no   0 0.6931472
3   3  some no   0 1.0986123
4   4  none yes  0 1.3862944
5   3  none no   0 1.0986123
6   4  none yes  1 1.3862944
7   1  some yes  1 0.0000000
8   2  some yes  1 0.6931472
9   5  some no   1 1.6094379
10  6  none no   1 1.7917595
```

In the first command above, I generated the new variable (`log_x1`) as the log of the variable `x1`. Now, both of variables exist in the dataset `stata.dat`.

Recoding variables is a bit more cumbersome. There are commands in the `car` library (written by John Fox) that make these operations more user-friendly. To make those

commands accessible, we first have to load the library with: `library(car)`. Then, we can see what the command structure looks like by looking at `help(recode)`. Let's now say that we want to make a new variable where values of one and 2 on `x1` are coded as 1 and values 3-6 are coded 2. We could do this with the `recode` command as follows:

```
> library(car)
> recode(stata.dat$x1, "c(1,2)=1; c(3,4,5,6)=2")

[1] 1 1 2 2 2 2 1 1 2 2
```

Here, the recodes amount to a vector of values and then the new value that is to be assigned to each of the existing values. The old/new combinations are each separated by a semi-colon and the entire recoding statement is put in double-quotes. Since I have not assigned the recode to an object, it simply prints the recode on the screen. It gives me a chance to, "try before I buy". If I'm happy with the output, I can now assign that recode to a new object.

```
> stata.dat$recoded_x1 <- recode(stata.dat$x1,
+                               "c(1,2)=1; c(3,4,5,6)=2")
> stata.dat
```

	x1	x2	x3	x4	log_x1	recoded_x1
1	1	none	yes	0	0.0000000	1
2	2	none	no	0	0.6931472	1
3	3	some	no	0	1.0986123	2
4	4	none	yes	0	1.3862944	2
5	3	none	no	0	1.0986123	2
6	4	none	yes	1	1.3862944	2
7	1	some	yes	1	0.0000000	1
8	2	some	yes	1	0.6931472	1
9	5	some	no	1	1.6094379	2
10	6	none	no	1	1.7917595	2

You can also recode entire ranges of values as well. Let's imagine that we want to recode `log_x1` such that anything greater than zero and less than 1.5 is a 1 and that anything greater than or equal to 1.5 is a 2. We could do that as follows:

```
> recode(stata.dat$log_x1, "0=0; 0:1.5=1; 1.5:hi = 2")

[1] 0 1 1 1 1 1 0 1 2 2
```

```
> cbind(stata.dat$log_x1, recode(stata.dat$log_x1,
+                               "0=0; 0:1.5=1; 1.5:hi = 2"))
```

	[,1]	[,2]
[1,]	0.0000000	0
[2,]	0.6931472	1
[3,]	1.0986123	1

```
[4,] 1.3862944 1
[5,] 1.0986123 1
[6,] 1.3862944 1
[7,] 0.0000000 0
[8,] 0.6931472 1
[9,] 1.6094379 2
[10,] 1.7917595 2
```

### 5.3 Missing Data

In **R**, missing data are indicated with `NA` (similar to the `.`, or `.a`, `.b`, etc..., in Stata). The dataset `r_example_miss.dta`, looks like this in Stata:

```
. list

      +-----+
      | x1      x2      x3 |
      |-----|
  1. | 1   none   yes |
  2. | 2   none   no  |
  3. | .   some   no  |
  4. | 4     .   yes |
  5. | 3   none   no  |
      |-----|
  6. | 4   none     . |
  7. | 1   some   yes |
  8. | 2   some   yes |
  9. | 5   some   no  |
 10. | 6   none   no  |
      +-----+
```

Notice that it looks like values are missing on all three variables. Let's read the data into **R** and see what happens.

```
> stata2.dat <- read.dta("r_example_miss.dta",
+       convert.factors=T)
> stata2.dat

  x1  x2  x3
1  1 none yes
2  2 none no
3 NA some no
4  4 <NA> yes
5  3 none no
6  4 none .
7  1 some yes
8  2 some yes
```

```
9 5 some no
10 6 none no
```

Notice that the missing element in the `x1` is `NA`, this is how missing data looks in numeric variables. In factors, it looks like `<NA>`. Notice that the `.` has not been converted to missing. **R** doesn't know that the period character in a string variable is supposed to be missing. If we convert it to a factor, however, it would not show up as missing:

```
> as.factor(stata2.dat$x3)

[1] yes no no yes no . yes yes no no
Levels: . no yes
```

It will only show up as missing if we explicitly tell **R** it is missing. We can do this one of two ways. First, we could make a new factor suggesting that the only valid values are 'yes' and 'no'.

```
> stata2.dat$x3fac <- factor(stata2.dat$x3,
+                            levels=c("no", "yes"))
> stata2.dat
```

```
   x1  x2  x3 x3fac
1   1  none yes  yes
2   2  none no   no
3  NA some no   no
4   4 <NA> yes  yes
5   3  none no   no
6   4  none .   <NA>
7   1 some yes  yes
8   2 some yes  yes
9   5 some no   no
10  6 none no   no
```

or, we could tell **R** that every instance of `'.'` in the variable `x3` should be changed to `NA`.

```
> stata2.dat$x3[which(stata2.dat$x3 == ".")] <- NA
> stata2.dat
```

```
   x1  x2  x3 x3fac
1   1  none yes  yes
2   2  none no   no
3  NA some no   no
4   4 <NA> yes  yes
5   3  none no   no
6   4  none <NA> <NA>
7   1 some yes  yes
8   2 some yes  yes
9   5 some no   no
10  6 none no   no
```

### 5.3.1 How R Deals with Missing Data

There are a few different methods for dealing with missing values, though they produce the same statistical result, they have different post-estimation behavior. These are specified through the `na.action` argument to modeling commands and you can see how these work by using the help functions: `?na.action`. In lots of the things we do, we will have to give the argument `na.rm=TRUE` to remove the missing data from the calculation. When we're taking the mean if a single observation has missing data, the mean returned for that variable will be `NA`, unless `na.rm=TRUE` is specified. More on this later.

## 5.4 Getting Help

There are a bunch of different places you can get help.

- R's internal help facilities work for both libraries and commands. If you know that package you want to look at, you can type: `help(package=foreign)` [if you want help for the `foreign` package]. Or, if you want help for a particular command, you can type `help(read.dta)`, for example. If you don't know what command you're looking for, you can type `??` and then a term you want to search for – `??regression`.
- The R Site Search tool is also helpful as it search through past listserv archives as well as help for various packages. You can find this page at <http://search.r-project.org/nmz.html> and you might also check out its parent, <http://search.r-project.org/>, for lots of helpful tools relating to finding help for **R**.
- As suggested above, there is a mailing list, similar to the Statalist, though beware here. This is not a particularly friendly place for newbies with newbie questions. You can see all of the various mailing lists here: <http://www.r-project.org/mail.html> and you will definitely want to look at the posting guide before sending anything here <http://www.r-project.org/posting-guide.html>. Failure to do so often results in the less-than-helpful RTFM response.
- There are a growing number of **R** books that cover both general and specific topics, it might be that you could find help for specific questions here.

Often times googling “R” and a few words relating to your question will return something interesting, but given it's name, searching this way can become a bit cumbersome. You can also rely on me and your peers for help.