Chapter 1

Data Analysis Commands

1.1 Command Syntax

Once R is installed, you only need to know a few basic elements to get started. It's important to remember that R, like any spoken language, has rules for proper syntax. Unlike English, however, the rules for intelligible R are small in number and quite precise (see Section **??**).

1.1.1 Getting Started

1. To start R under Linux or Unix, type R at the terminal prompt or M-x R under ESS.
2. The R prompt is `>`.
3. Type commands and hit enter to execute. (No additional characters, such as semicolons or commas, are necessary at the end of lines.)
4. To quit from R, type `q()` and press enter.
5. The `#` character makes R ignore the rest of the line, and is used in this document to comment R code.
6. We highly recommend that you make a separate working directory or folder for each project.
7. Each R session has a workspace, or working memory, to store the *objects* that you create or input. These objects may be:
   (a) *values*, which include numerical, integer, character, and logical values;
   (b) *data structures* made up of variables (vectors), matrices, and data frames; or
   (c) *functions* that perform the desired tasks on user-specified values or data structures.

After starting R, you may at any time use Zelig's built-in help function to access on-line help for any command. To see help for all Zelig commands, type `help.zelig(command)`, which will take you to the help page for all Zelig commands. For help with a specific Zelig or R command substitute the name of the command for the generic `command`. For example, type `help.zelig(logit)` to view help for the logit model.

1.1.2 Details

.

Zelig uses the syntax of R, which has several essential elements:

1. R is case sensitive. `Zelig`, the package or library, is not the same as `zelig`, the command.

2. R functions accept user-defined arguments: while some arguments are required, other optional arguments modify the function's default behavior. Enclose arguments in parentheses and separate multiple arguments with commas. For example, `print(x)` or `print(x, digits = 2)` prints the contents of the object `x` using the default number of digits or rounds to two digits to the right of the decimal point, respectively. You may nest commands as long as each has its own set of parentheses: `log(sqrt(5))` takes the square root of 5 and then takes the natural log.

3. The `<-` operator takes the output of the function on the right and saves them in the named object on the left. For example, `z.out <- zelig(...)` stores the output from `zelig()` as the object `z.out` in your working memory. You may use `z.out` as an argument in other functions, view the output by typing `z.out` at the R prompt, or save `z.out` to a file using the procedures described in Section **??**.

4. You may name your objects anything, within a few constraints:

   - You may only use letters (in upper or lower case) and periods to punctuate your variable names.
   - You may *not* use any special characters (aside from the period) or spaces to punctuate your variable names.
   - Names cannot begin with numbers. For example, R will not let you save an object as `1997.election` but will let you save `election.1997`.

5. Use the `names()` command to see the contents of R objects, and the `$` operator to extract elements from R objects. For example:

```
# Run least squares regression and save the output in working memory:
> z.out <- zelig(y ~ x1 + x2, model = "ls", data = mydata)
# See what's in the R object:
> names(z.out)
```

```
 [1] "coefficients"  "residuals"  "effects"  "rank"
# Extract and display the coefficients in z.out:
> z.out$coefficients
```

6. All objects have a class designation which tells R how to treat it in subsequent commands. An object's class is generated by the function or mathematical operation that created it.

7. To see a list of all objects in your current workspace, type: `ls()`. You can remove an object permanently from memory by typing `remove(goo)` (which deletes the object `goo`), or remove all the objects with `remove(list = ls())`.

8. To run commands in a batch, use a text editor (such as the Windows R script editor or emacs) to compose your R commands, and save the file with a `.R` file extension in your working directory. To run the file, type `source("Code.R")` at the R prompt.

If you encounter a syntax error, check your spelling, case, parentheses, and commas. These are the most common syntax errors, and are easy to detect and correct with a little practice. If you encounter a syntax error in batch mode, R will tell you the line on which the syntax error occurred.

1.2 Data Sets

1.2.1 Data Structures

Zelig uses only three of R's many data structures:

1. A **variable** is a one-dimensional vector of length n.

2. A **data frame** is a rectangular matrix with n rows and k columns. Each column represents a variable and each row an observation. Each variable may have a different class. (See Section **??** for a list of classes.) You may refer to specific variables from a data frame using, for example, `data$variable`.

3. A **list** is a combination of different data structures. For example, `z.out` contains both `coefficients` (a vector) and `data` (a data frame). Use `names()` to view the elements available within a list, and the `$` operator to refer to an element in a list.

For a more comprehensive introduction, including ways to manipulate these data structures, please refer to Chapter **??**.

1.2.2 Loading Data

Datasets in Zelig are stored in "data frames." In this section, we explain the standard ways to load data from disk into memory, how to handle special cases, and how to verify that the data you loaded is what you think it is.

Standard Ways to Load Data

Make sure that the data file is saved in your working directory. You can check to see what your working directory is by starting R, and typing `getwd()`. If you wish to use a different directory as your starting directory, use `setwd("dirpath")`, where `"dirpath"` is the full directory path of the directory you would like to use as your working directory.

After setting your working directory, load data using one of the following methods:

1. If your dataset is in a **tab- or space-delimited .txt file**, use `read.table("mydata.txt")`

2. If your dataset is **a comma separated table**, use `read.csv("mydata.csv")`.

3. To import **SPSS, Stata, and other data files**, use the foreign package, which automatically preserves field characteristics for each variable. Thus, variables classed as dates in Stata are automatically translated into values in the date class for R. For example:

```
> library(foreign)                              # Load the foreign package.
> stata.data <- read.dta("mydata.dta")          # For Stata data.
> spss.data <- read.spss("mydata.sav", to.data.frame = TRUE) # For SPSS.
```

4. To load data in R format, use `load("mydata.RData")`.

5. For sample data sets included with R packages such as Zelig, you may use the `data()` command, which is a shortcut for loading data from the sample data directories. Because the locations of these directories vary by installation, it is extremely difficult to locate sample data sets and use one of the three preceding methods; `data()` searches all of the currently used packages and loads sample data automatically. For example:

```
> library(Zelig)                                # Loads the Zelig library.
> data(turnout)                                 # Loads the turnout data.
```

Special Cases When Loading Data

These procedures apply to any of the above `read` commands:

1. If your file uses the **first row to identify variable names**, you should use the option `header = TRUE` to import those field names. For example,

   ```
   > read.csv("mydata.csv", header = TRUE)
   ```

   will read the words in the first row as the variable names and the subsequent rows (each with the same number of values as the first) as observations for each of those variables. If you have additional characters on the last line of the file or fewer values in one of the rows, you need to edit the file before attempting to read the data.

2. The R missing value code is `NA`. If this value is in your data, R will recognize your missing values as such. If you have instead used a place-holder value (such as -9) to represent missing data, you need to tell R this on loading the data:

   ```
   > read.table("mydata.tab", header = TRUE, na.strings = "-9")
   ```

   Note: You must enclose your place holder values in quotes.

3. Unlike Windows, the file extension in R does not determine the default method for dealing with the file. For example, if your data is tab-delimited, but saved as a `.sav` file, `read.table("mydata.sav")` will load your data into R.

Verifying You Loaded The Data Correctly

Whichever method you use, try the `names()`, `dim()`, and `summary()` commands to verify that the data was properly loaded. For example,

```
> data <- read.csv("mydata.csv", header = TRUE)             # Read the data.
> dim(data)                    # Displays the dimensions of the data frame
[1] 16000  8                    #  in rows then columns.
> data[1:10,]                  # Display rows 1-10 and all columns.
> names(data)                  # Check the variable names.
[1] "V1" "V2" "V3"             # These values indicate that the variables
                               #  weren't named, and took default values.
> names(data) <- c("income", "educate", "year")     # Assign variable names.
> summary(data)                # Returning a summary for each variable.
```

In this case, the `summary()` command will return the maximum, minimum, mean, median, first and third quartiles, as well as the number of missing values for each variable.

1.2.3 Saving Data

Use `save()` to write data or any object to a file in your working directory. For example,

```
> save(mydata, file = "mydata.RData")     # Saves `mydata' to `mydata.RData'
                                          #  in your working directory.
> save.image()                            # Saves your entire workspace to
                                          #  the default `.RData' file.
```

R will also prompt you to save your workspace when you use the `q()` command to quit. When you start R again, it will load the previously saved workspace. Restarting R will not, however, load previously used packages. You must remember to load Zelig at the beginning of every R session.

Alternatively, you can recall individually saved objects from .RData files using the `load()` command. For example,

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

loads the objects saved in the `mydata.RData` file. You may save a data frame, a data frame and associated functions, or other R objects to file.

1.3 Variables

1.3.1 Classes of Variables

R variables come in several types. Certain Zelig models require dependent variables of a certain class of variable. (These are documented under the manual pages for each model.) Use `class(variable)` to determine the class of a variable or `class(data$variable)` for a variable within a data frame.

Types of Variables

For all types of variable (vectors), you may use the `c()` command to "concatenate" elements into a vector, the `:` operator to generate a sequence of integer values, the `seq()` command to generate a sequence of non-integer values, or the `rep()` function to repeat a value to a specified length. In addition, you may use the `<-` operator to save variables (or any other objects) to the workspace. For example:

```
> logic <- c(TRUE, FALSE, TRUE, TRUE, TRUE) # Creates `logic' (5 T/F values).
> var1 <- 10:20                             # All integers between 10 and 20.
> var2 <- seq(from = 5, to = 10, by = 0.5)  # Sequence from 5 to 10 by
                                            #  intervals of 0.5.
> var3 <- rep(NA, length = 20)              # 20 `NA' values.
> var4 <- c(rep(1, 15), rep(0, 15))         # 15 `1's followed by 15 `0's.
```

For the `seq()` command, you may alternatively specify `length` instead of `by` to create a variable with a specific number (denoted by the `length` argument) of evenly spaced elements.

1. **Numeric** variables are real numbers and the default variable class for most dataset values. You can perform any type of math or logical operation on numeric values. If `var1` and `var2` are numeric variables, we can compute

   ```
   > var3 <- log(var2) - 2*var1      # Create `var3' using math operations.
   ```

   `Inf` (infinity), `-Inf` (negative infinity), `NA` (missing value), and `NaN` (not a number) are special numeric values on which most math operations will fail. (Logical operations will work, however.) Use `as.numeric()` to transform variables into numeric variables. Integers are a special class of numeric variable.

2. **Logical** variables contain values of either `TRUE` or `FALSE`. R supports the following logical operators: `==`, exactly equals; `>`, greater than; `<`, less than; `>=`, greater than or equals; `<=`, less than or equals; and `!=`, not equals. The `=` symbol is *not* a logical operator. Refer to Section **??** for more detail on logical operators. If `var1` and `var2` both have n observations, commands such as

   ```
   > var3 <- var1 < var2
   > var3 <- var1 == var2
   ```

   create n `TRUE`/`FALSE` observations such that the ith observation in `var3` evaluates whether the logical statement is true for the ith value of `var1` with respect to the ith value of `var2`. Logical variables should usually be converted to integer values prior to analysis; use the `as.integer()` command.

3. **Character** variables are sets of text strings. Note that text strings are always enclosed in quotes to denote that the string is a value, not an object in the workspace or an argument for a function (neither of which take quotes). Variables of class character are not normally used in data analysis, but used as descriptive fields. If a character variable is used in a statistical operation, it must first be transformed into a factored variable.

4. **Factor** variables may contain values consisting of either integers or character strings. Use `factor()` or `as.factor()` to convert character or integer variables into factor variables. Factor variables separate unique values into levels. These levels may either be ordered or unordered. In practice, this means that including a factor variable among the explanatory variables is equivalent to creating dummy variables for each level. In addition, some models (ordinal logit, ordinal probit, and multinomial logit), require that the dependent variable be a factor variable.

1.3.2 Recoding Variables

Researchers spend a significant amount of time cleaning and recoding data prior to beginning their analyses. R has several procedures to facilitate the process.

Extracting, Replacing, and Generating New Variables

While it is not difficult to recode variables, the process is prone to human error. Thus, we recommend that before altering the data, you save your existing data frame using the procedures described in Section **??**, that you only recode one variable at a time, and that you recode the variable outside the data frame and then return it to the data frame.

To extract the variable you wish to recode, type:

```
> var <- data$var1              # Copies `var1' from `data', creating `var'.
```

Do *not* sort the extracted variable or delete observations from it. If you do, the ith observation in `var` will no longer match the ith observation in `data`.

To replace the variable or generate a new variable in the data frame, type:

```
> data$var1 <- var                # Replace `var1' in `data' with `var'.
> data$new.var <- var             # Generate `new.var' in `data' using `var'.
```

To remove a variable from a data frame (rather than replacing one variable with another):

```
> data$var1 <- NULL
```

Logical Operators

R has an intuitive method for recoding variables, which relies on logical operators that return statements of `TRUE` and `FALSE`. A mathematical operator (such as `==`, `!=`, `>`, `>=` `<`, and `<=`) takes two objects of equal dimensions (scalars, vectors of the same length, matrices with the same number of rows and columns, or similarly dimensioned arrays) and compares every element in the first object to its counterpart in the second object.

- `==`: checks that one variable "exactly equals" another in a list-wise manner. For example:

  ```
  > x <- c(1, 2, 3, 4, 5)               # Creates the object `x'.
  > y <- c(2, 3, 3, 5, 1)               # Creates the object `y'.
  > x == y                              # Only the 3rd `x' exactly equals
    [1] FALSE FALSE  TRUE FALSE FALSE   #  its counterpart in `y'.
  ```

  (The `=` symbol is *not* a logical operator.)

- `!=`: checks that one variable does not equal the other in a list-wise manner. Continuing the example:

  ```
  > x != y
    [1]  TRUE  TRUE FALSE  TRUE  TRUE
  ```

- `>` (`>=`): checks whether each element in the left-hand object is greater than (or equal to) every element in the right-hand object. Continuing the example from above:

  ```
  > x > y                                 # Only the 5th `x' is greater
    [1] FALSE FALSE FALSE FALSE  TRUE     #  than its counterpart in `y'.
  > x >= y                                # The 3rd `x' is equal to the
    [1] FALSE FALSE  TRUE FALSE  TRUE     #  3rd `y' and becomes TRUE.
  ```

- `<` (`<=`): checks whether each element in the left-hand object is less than (or equal to) every object in the right-hand object. Continuing the example from above:

```
> x < y                             # The elements 1, 2, and 4 of `x' are
[1]  TRUE  TRUE FALSE  TRUE FALSE #  less than their counterparts in `y'.
> x <= y                            # The 3rd `x' is equal to the 3rd `y'
[1]  TRUE  TRUE  TRUE  TRUE FALSE #  and becomes TRUE.
```

For two vectors of five elements, the mathematical operators compare the first element in `x` to the first element in `y`, the second to the second and so forth. Thus, a mathematical comparison of `x` and `y` returns a vector of five `TRUE`/`FALSE` statements. Similarly, for two matrices with 3 rows and 20 columns each, the mathematical operators will return a 3×20 matrix of logical values.

There are additional logical operators which allow you to combine and compare logical statements:

- `&`: is the logical equivalent of "and", and evaluates one array of logical statements against another in a list-wise manner, returning a `TRUE` only if both are true in the same location. For example:

```
> a <- matrix(c(1:12), nrow = 3, ncol = 4)      # Creates a matrix `a'.
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> b <- matrix(c(12:1), nrow = 3, ncol = 4)      # Creates a matrix `b'.
> b
     [,1] [,2] [,3] [,4]
[1,]   12    9    6    3
[2,]   11    8    5    2
[3,]   10    7    4    1
> v1 <- a > 3                   # Creates the matrix `v1' (T/F values).
> v2 <- b > 3                   # Creates the matrix `v2' (T/F values).
> v1 & v2                       # Checks if the (i,j) value in `v1' and
      [,1] [,2] [,3]  [,4]      #  `v2' are both TRUE.  Because columns
[1,] FALSE TRUE TRUE FALSE      #  2-4 of `v1' are TRUE, and columns 1-3
[2,] FALSE TRUE TRUE FALSE      #  of `var2' are TRUE, columns 2-3 are
[3,] FALSE TRUE TRUE FALSE      #  TRUE here.
> (a > 3) & (b > 3)             # The same, in one step.
```

  For more complex comparisons, parentheses may be necessary to delimit logical statements.

- `|`: is the logical equivalent of "or", and evaluates in a list-wise manner whether either of the values are `TRUE`. Continuing the example from above:

```
> (a < 3) | (b < 3)                # (1,1) and (2,1) in `a' are less
      [,1]  [,2]  [,3]  [,4]       #  than 3, and (2,4) and (3,4) in
[1,]  TRUE FALSE FALSE FALSE       #  `b' are less than 3; | returns
[2,]  TRUE FALSE FALSE  TRUE       #  a matrix with `TRUE' in (1,1),
[3,] FALSE FALSE FALSE  TRUE       #  (2,1), (2,4), and (3,4).
```

The `&&` (if and only if) and `||` (either or) operators are used to control the command flow within functions. The `&&` operator returns a `TRUE` only if every element in the comparison statement is true; the `||` operator returns a `TRUE` if any of the elements are true. Unlike the `&` and `|` operators, which return arrays of logical values, the `&&` and `||` operators return only one logical statement irrespective of the dimensions of the objects under consideration. Hence, `&&` and `||` are logical operators which are *not* appropriate for recoding variables.

Coding and Recoding Variables

R uses vectors of logical statements to indicate how a variable should be coded or recoded. For example, to create a new variable `var3` equal to 1 if `var1` $<$ `var2` and 0 otherwise:

```
> var3 <- var1 < var2              # Creates a vector of n T/F observations.
> var3 <- as.integer(var3)         # Replaces the T/F values in `var3' with
                                   #  1's for TRUE and 0's for FALSE.
> var3 <- as.integer(var1 < var2)  # Combine the two steps above into one.
```

In addition to generating a vector of dummy variables, you can also refer to specific values using logical operators defined in Section **??**. For example:

```
> v1 <- var1 == 5                  # Creates a vector of T/F statements.
> var1[v1] <- 4                    # For every TRUE in `v1', replaces the
                                   #  value in `var1' with a 4.
> var1[var1 == 5] <- 4             # The same, in one step.
```

The index (inside the square brackets) can be created with reference to other variables. For example,

```
> var1[var2 == var3] <- 1
```

replaces the ith value in `var1` with a 1 when the ith value in `var2` equals the ith value in `var3`. If you use `=` in place of `==`, however, you will replace all the values in `var1` with 1's because `=` is another way to assign variables. Thus, the statement `var2 = var3` is of course true.

Finally, you may also replace any (character, numerical, or logical) values with special values (most commonly, `NA`).

```
> var1[var1 == "don't know"] <- NA   # Replaces all "don't know"'s with NA's.
```

After recoding the `var1` replace the old `data$var1` with the recoded `var1`: `data$var1 <- var1`. You may combine the recoding and replacement procedures into one step. For example:

```
> data$var1[data$var1 =< 0] <- -1
```

Alternatively, rather than recoding just specific values in variables, you may calculate new variables from existing variables. For example,

```
> var3 <- var1 + 2 * var2
> var3 <- log(var1)
```

After generating the new variables, use the assignment mechanism `<-` to insert the new variable into the data frame.

In addition to generating vectors of dummy variables, you may transform a vector into a matrix of dummy indicator variables. For example, see Section **??** to transform a vector of k unique values (with n observations in the complete vector) into a $n \times k$ matrix.

Missing Data

To deal with missing values in some of your variables:

1. You may generate multiply imputed datasets using Amelia (or other programs).

2. You may omit missing values. Zelig models automatically apply list-wise deletion, so no action is required to run a model. To obtain the total number of observations or produce other summary statistics using the analytic dataset, you may manually omit incomplete observations. To do so, first create a data frame containing only the variables in your analysis. For example:

   ```
   > new.data <- cbind(data$dep.var, data$var1, data$var2, data$var3)
   ```

   The `cbind()` command "column binds" variables into a data frame. (A similar command `rbind()` "row binds" observations with the same number of variables into a data frame.) To omit missing values from this new data frame:

   ```
   > new.data <- na.omit(new.data)
   ```

   If you perform `na.omit()` on the full data frame, you risk deleting observations that are fully observed in your experimental variables, but missing values in other variables. Creating a new data frame containing only your experimental variables usually increases the number of observations retained after `na.omit()`.