Transforming and Restructuring Data

Jamie DeCoster

Department of Psychology
University of Alabama
348 Gordon Palmer Hall
Box 870348
Tuscaloosa, AL 35487-0348

Phone: (205) 348-4431
Fax: (205) 348-8648

May 14, 2001

These notes were prepared with the support of a grant from the Dutch Science Foundation. I would like to thank Heather Claypool and Lynda Mae for comments made on earlier versions of these notes. If you wish to cite the contents of this document, the APA reference for them would be

DeCoster, J. (2001). *Transforming and Restructuring Data.* Retrieved <month, day, and year you downloaded this file> from http://www.stat-help.com/notes.html

For future versions of these notes or help with data analysis visit
`http://www.stat-help.com`

Contents

1 Introduction **1**

2 Transformations: Calculating New Values from Existing Variables **6**

3 Normalizing Data **10**

4 Working with Conditionals (if statements) **15**

5 Working with Arrays and Loops **20**

6 Restructuring Data: Changing the Unit of Analysis **28**

Chapter 1

Introduction

1.1 Overview

- Often times the initial form of your data is not the way you want it for analysis. The reasons for this could be many. For example,
  - A researcher might choose to have data entered in a format that is easy for typists (to reduce data-entry errors) but which differs from the form needed for analysis.
  - An experiment may have been administered by a computer program that is forced to record the data on a trial-by-trial basis when the participant is the desired unit of analysis.
  - The residuals of an ANOVA might be observed to have a severe skew. This is problematic because ANOVAs assume that the residuals have a normal distribution. Correcting this often involves transforming the response variable.
  - A particular way of looking at the data is not apparent until after analysis has already begun and the data have been loaded into the statistics program in a format incompatible with the new analysis.
- These notes attempt to explain the circumstances under which you would manipulate your data and provide a number of tools and techniques to make manipulation easier and more efficient. Three tools that are particularly important are conditional statements, loops, and arrays.
  - *Conditional statements*, explained in chapter 4, allow you to apply categorical transformations. This includes both transformations of a categorical variable as well as applying different transformations to a numeric variable based on a categorical distinction.
  - *Loops* and *arrays*, explained in chapter 5, provide you with a means of performing large numbers of similar transformations using a relatively small section of written code.
- The great majority of people performing statistical analysis do so using either SPSS or SAS. These notes will therefore always follow the introduction of a particular method of data manipulation with specific instructions on how to implement it in both of these software packages. In the main body of each chapter we will use pseudocode (generic programming statements not specifically applicable to either program).

1.2 Data and Data Sets

- The information that you collect from an experiment, survey, or archival source is referred to as your *data*. Most generally, data can be defined as list of numerical and/or categorical values possessing meaningful relationships.

- For analysts to do anything with a group of data they must first translate it into a *data set*. A data set is a representation of data, defining a set of "variables" that are measured on a set of "cases."
    - A *variable* is simply a feature of an object that can categorized or measured by a number. A variable takes on different *values* to reflect the particular nature of the object being observed. The values that a variable takes will vary when measurements are made on different objects at different times. A data set will typically contain measurements on several different variables.
    - Each time that we record information about an object we create a *case*. Like variables, a data set will typically contain multiple cases. The cases should all be derived from observations of the same type of object with each case representing a different example of that type. Cases are also sometimes referred to as *observations*. The object "type" that defines your cases is called your *unit of analysis*. Sometimes the unit of analysis in a data set will be very small and specific, such as the individual responses on a questionnaire. Sometimes it will be very large, such as companies or nations.

  When describing a data set you should always provide definitions for your variables and the unit of analysis. You typically would not list the specific cases, although you might describe their general characteristics.

- Many different data sets can be constructed from the same data. Different data sets could contain different variables and possibly even different cases.

- For example, a researcher gives a survey to four different people (John, Vicki, James, and Heather) asking them how they felt about dogs, cats, and birds. The survey showed that
    - John likes dogs, but is neutral towards cats and birds.
    - Vicki dislikes dogs, but likes cats and birds.
    - James is neutral towards dogs, but dislikes cats and birds.
    - Heather dislikes dogs, likes cats, and is neutral towards birds.

  From this data the researcher could construct the data set presented in table 1.1. When displaying a data set in tabular format we generally put each case in a separate row and each variable in a separate column. The entry in a given cell of the table represents the value of the variable in that column for the case in that row.

Table 1.1: Pet Data Set 1

Case	**Person**	**Pet**	**Rating**
1	John	Dog	1
2	John	Cat	0
3	John	Bird	0
4	Vicki	Dog	−1
5	Vicki	Cat	1
6	Vicki	Bird	1
7	James	Dog	0
8	James	Cat	−1
9	James	Bird	−1
10	Heather	Dog	−1
11	Heather	Cat	1
12	Heather	bird	0

The unit of analysis for this data set is a person's evaluation about a pet. It has three variables: **person**, representing whose evaluation it is, **pet**, representing the animal being evaluated, and **rating**, coding whether the person has a positive, negative, or neutral evaluation.

While this is an accurate representation of the data, it might be easier to examine if the responses from the same person could be seen on the same line. The researcher might therefore restructure the data set as in table 1.2.

Table 1.2: Pet Data Set 2

Case	**Person**	**Dog**	**Cat**	**Bird**
1	John	1	0	0
2	Vicki	−1	1	1
3	James	0	−1	−1
4	Heather	−1	1	0

The unit of analysis for this data set is an individual. This time there are four variables: **person**, indicating who is providing the evaluation, **dog**, representing the person's evaluation of dogs, **cat**, representing the person's evaluation of cats, and **bird**, representing the person's evaluation of birds.

Looking at the data this way it's pretty clear that some people appear to like pets in general more than others. The researcher might therefore decide that it would be useful to add a new variable to indicate the person's average pet rating. The data set that would result appears in table 1.3.

Table 1.3: Pet Data Set 3

Case	**Person**	**Dog**	**Cat**	**Bird**	**Average**
1	John	1	0	0	.33
2	Vicki	−1	1	1	.33
3	James	0	−1	−1	−.66
4	Heather	−1	1	0	0

The unit of analysis for this data set is again the individual. It includes all of the variables found in data set 2 as well as a new variable, **average**, representing the mean rating of all three pets.

All three of these data sets are accurate representations of the original data but contain different variables and have different units of analysis. The important thing when building your data set is to make sure that you maintain the relationships that were originally present in the data. The exact structure that your data sets should have depends on what sort of analyses you wish to perform. Analyses that are easy using one form of your data could be very difficult using another.

1.3 Data Manipulation

- *Data manipulation* is the procedure of creating a new data set from an existing data set. In almost every study you will need to alter your initial data set in some way before you can begin analysis.
- The different ways that you can change your data set can be grouped into two general categories.
  1. Changes that involve calculating new variables as a function of one or more old variables in your data set are called *transformations*. The new data set will typically have all of the original variables, with the addition of one or more new variables. Sometimes a transformation will simply involve changing the values of an existing variable. After performing a transformation the cases of the new data set will be exactly the same as those of the old data set.
  2. If you alter your data set in such a way that you end up changing the unit of analysis you are performing *data restructuring*. The new data set will typically use entirely new variables, with maybe a small number that are the same as in the original data set. Additionally, your new data

set will be composed of entirely new cases. Restructuring a data set is typically a more more difficult and involved procedure than simply transforming variables.

- The first thing you should always do when thinking about manipulating your data is to write down exactly what you would want your final data set to look like. You should describe the unit of analysis for your cases, as well as define all of your variables. This step will make it much easier for you to determine what transformation and restructuring steps you will need to take.

1.4 Data Manipulation in SPSS

- There are two basic ways that you can work with SPSS. Most users typically open up an SPSS data file in the data editor, and then select items from the menus to manipulate the data or to perform statistical analyses. This is referred to as *interactive mode*, because your relationship with the program is very much like a personal interaction, with the program providing a response each time you make a selection. If you request a transformation the new data set is immediately updated. When you select an analysis the results immediately appear in the output window.

- It is also possible to work with SPSS in *syntax mode*, where the user types code in a syntax window. Once the full program is written it is then submitted to SPSS to get the results. Working with syntax is more difficult than working with the menus because you must learn how to write the programming code to produce the manipulations and analyses you want. However, certain procedures and analyses are only available through the use of syntax. For example, loops and arrays (see chapter 5) are only available when working with syntax. You can also save the programs you write in syntax. This can be very useful if you expect to perform the same or similar analyses multiple times. If you would like more general information about writing SPSS syntax you should examine the *SPSS Base Syntax Reference Guide*.

- Whether you should work in interactive or syntax mode depends on several things. Interactive mode is easier and generally quicker if you only need to perform a small number of simple transformations on your data. You should therefore probably work interactively unless you have a specific reason to use syntax. Some reasons to choose syntax would be:

  - You need to use options or procedures that are not available using interactive mode.
  - You expect that you will perform the same manipulations or analyses on several different data sets and want to save a copy of the program code so that it can easily be re-run.
  - You need to perform a large number of conditional transformations, such that you would be benefited by using arrays and loops.
  - You need to change the unit of analysis of your data set.
  - You are performing a very complicated set of manipulations or analysis, such that it would be useful to document all of the steps leading to your results.

- Whenever you work in interactive mode, SPSS actually writes down syntax code reflecting the menu choices you select in a "journal file." The name of this file can be found (or changed) by selecting **Edit** → **Options** and selecting the **General** tab. If you ever want to see or use this code you can edit the journal file in a syntax window.

- SPSS also provides an easy way to see the code corresponding to a particular menu function. Most selections include a **Paste** button which will open up a syntax window containing the code for the function, including the details required for any specific options that you have chosen.

1.5 Data Manipulation in SAS

- Like SPSS, SAS can be used in either an interactive mode or a syntax mode. However, the interactive mode in SAS (called *SAS Insight*) is actually difficult to find and difficult to use. Most people who use

SAS type code into the program window and then run the final program to get the results. In these notes we will therefore only discuss how to perform data manipulation in SAS using programming code.

- SAS imposes some structure on the way that you may write your programs. Specifically, SAS requires that you keep the code relating to loading and manipulating your data separate from the code that performs statistical analysis. Therefore, each time you create a new data set, SAS divides the following code into a *Data Step* and a *Procedure Step*. It assumes that the Data Step ends as soon as you write your first **proc** statement.

  All data manipulation must be performed in the Data Step. SAS will give you an error if you try to change any of your variables after you have already asked for an analytical procedure. If you want to manipulate your data after performing a procedure this must be done in a new data set. You can copy the variables from one data set to another using the **set** statement, as in the following example.

```
data old;
      ⋮
(the variable C is created here in a procedure so you can't transform it in
this data set)
      ⋮
data new;
  set old;
  ctrans = c + 6;
run;
```

  If you would like more general information about writing SAS code you should examine the *SAS Language Guide*.

Chapter 2

Transformations: Calculating New Values from Existing Variables

2.1 Conceptual Overview

- One of the simplest types of data manipulation is to calculate a new value as a direct function of existing variables in your data set. Most commonly the calculated value will be assigned to a new variable in your data set, although it can also be used to replace values on an existing variable.

- The basic requirement to use the procedures presented in this chapter is that the same transformation function applies to every case in the existing data set, and that the calculation can be made using some combination of the variables present within a single case. For example, if you would need to obtain information from case 2 to calculate the new value for case 1, you would need to use more complicated procedures.

  There is a way to get around this if you are working with summary statistics of your variables. For example, you might be interested in standardizing a variable. A standardized value z is defined as

  $$z_i = \frac{x_i - \bar{x}}{s_x}, \tag{2.1}$$

  where x is the original variable, $\bar{x}$ is the mean of the original variable, and s_x is the standard deviation of the original variable. Calculating the mean and the standard deviation of course requires using information from all of your cases. However, you could calculate the mean and standard deviation in a separate step. All you would then need to do for each case would be to subtract a constant from the original variable and then divide by a constant. This way you no longer need to access information from other cases to perform your transformation, so you can work with the simple methods presented in this chapter.

- The first step to applying a transformation is to write down an expression that defines your new value as a function of existing variables in your data set. This equation can be as simple or as complicated as you want to make it. Sometimes a transformation calculates a new value from a single existing variable. Some examples might be

  $$y = \sqrt{x}, \tag{2.2}$$

  $$b = \ln a, \tag{2.3}$$

  or

  $$yquad = 4y^2 + 3y + 6. \tag{2.4}$$

  Other transformations calculate a new value using several existing variables. Some examples might be

  $$d = \frac{a + b + c}{3}, \tag{2.5}$$

$$invsum = \frac{1}{x} + \frac{1}{y} + \frac{1}{z}, \tag{2.6}$$

or

$$varmax = \max(v1, v2, v3). \tag{2.7}$$

It is perfectly acceptable to assign the result of your transformation to a variable that is used in the calculation of the new value. The original value is used in the calculation the new value, but is replaced immediately afterwards. For example, if the value of x in one case was 6, applying the transformation

$$x = x - 3 \tag{2.8}$$

would change the value of x for that case to be 3. You should be careful when performing such transformations because the new value will completely replace the old value. It is generally better to use new variables to hold the results of your transformations. However, you might want to replace existing variables to apply a transformation on a variable but analyze it using some pre-existing code (which makes reference to the original variable name), or you may simply want to limit the number of variables in your data set.

- After you have defined your transformation you must apply it to your data set. The exact way you go about doing this will vary depending on what software you are using. In general it is a fairly easy procedure, involving little more than describing the transformation expression to your software and then indicating what variable you want to hold the new value.

2.2 Using SPSS

- When using SPSS interactively, new values can be obtained by selecting **Transform** → **Compute**. This opens up a dialog box with blank fields for **Target Variable** and **Numeric expression**, as well as lists of the variables in your data set and the functions available to SPSS. You will also see buttons for all the numbers and simple arithmetic symbols. The first thing you need to do here is type in the mathematical function in the **Numeric Expression** field. Double clicking on a entry in the list of variables of the list of functions will put the corresponding entry into the **Numeric Expression** field. Similarly, clicking one of the buttons enters the corresponding number or symbol into your expression. Both the lists and buttons can be useful but they are not necessary. It is often faster to just type in the expression using the keyboard.

  The next thing you need to do is indicate what variable you wish to hold the calculated value in the **Target Variable** field. If you enter the name of a variable that already exists in the data set, the new values will replace those that already exist in that variable. If you enter a name that doesn't correspond to any existing variable a new variable will be created with that name to hold the calculated values.

  Once you have filled in both the **Numeric Expression** and **Target Variable** fields you click the **OK** button to tell SPSS to perform the transformation.

  At the bottom of the of the dialog button you will notice a button labeled **If**. This allows you to only apply the transformation to a limited set of cases that satisfy some criterion that you establish. The use of this will be discussed in more detail in chapter 4.

- When using SPSS syntax, transformations may be performed using the **compute** statement. The format is

  compute *target variable* = *expression*.

  where *target variable* is the name of the variable to which you want the new values assigned. If the name is the same as an existing variable in your data set, the new values will replace the existing values of that variable. If the name does not match any existing variables, SPSS will create a new variable to hold the computed values. *expression* is the mathematical function of existing variables that defines

the transformation you want. The mathematical functions that you may use are the same as when in interactive mode, and are listed on pages 44-60 of the *SPSS Base Syntax Reference Guide*.

For example, if you wanted to create a new variable *xprime* equal to the base 10 logarithm of an existing variable x you would issue the statements

compute xprime = lg10(x).
execute.

In syntax mode, SPSS waits until it receives an **execute** statement before performing any transformations or analyses. It is therefore a good idea to always include this statement at the end of your programs.

2.3 Using SAS

- Performing transformations in SAS is very similar to performing transformations in SPSS syntax mode. To create or modify a variable you may issue an assignment command

  target variable = *expression*;

  where *target variable* is the name of the variable to which you want the new values assigned. If the name is the same as an existing variable in your data set, the new values will replace the existing values of that variable. If the name does not match any existing variables, SAS will create a new variable to hold the computed values. *expression* is the mathematical function of existing variables that defines the transformation you want. The mathematical functions that you may use are listed on pages 521-523 of the *SAS Language Guide*.

  For example, if you wanted to create a new variable *acos* equal to the cosine of an existing variable a you would issue the statements

  acos = cos(a);
  run;

  SAS waits until it receives a **run** statement before performing any transformations or analyses. It is therefore a good idea to always include this statement at the end of your programs.

- Remember that any changes to your data set must be made in the data step of your program. If you need to transform variables created by a procedure you will need to create a new data set in which to perform your transformations.

2.4 Missing Values

- Missing values, by default represented by periods in both SPSS and SAS, indicate that your data set does not contain a value for a particular variable on a particular case. When performing transformations there are basically two reasons you might get a missing value.
  - The transformation produced an erroneous value for the particular case. For example, the transformation you proposed might have called for dividing by zero or for taking the square root of a negative number.
  - One of the variables in the transformation function had a missing value.
- If you ever find that the result of a transformation produces nothing but missing values, it probably means that there was an error in your transformation expression.

- Sometimes you might wish to calculate a summary statistic ignoring missing values. For example, you might want the average of a very large set of variables where it doesn't matter much if the average fails include a few variables that have missing values. Both SPSS and SAS have functions that calculate summary statistics this way.
    - SPSS functions that ignore cases with missing values are **sum, mean, sd, variance, cfvar, min,** and **max**.
    - SAS functions that ignore cases with missing values are **css, cv, kurtosis, max, mean, min, range, skewness, std, stderr, sum, uss,** and **var**.

Chapter 3

Normalizing Data

3.1 Conceptual Overview

- One of the most common reasons people perform a transformation is in an attempt to give a variable a *normal distribution*. Figure 3.1 shows an example of a normal distribution.

  –Insert Figure 3.1 here–

  Figure 3.1: Example of a normal distribution.

  The *relative frequency* can be thought of as either the probability of observing an observation of the given value, or as how many of that value you might expect if you observed a large number of observations on that variable.

- The main reason that you would want a variable to have a normal distribution is that both ANOVA and regression make the assumption that the response variable is distributed normally at each level of the explanatory variable. If your response variable tends to follow some other type of distribution you may falsely conclude that your explanatory variable predicts the response when it actually does not.

  Be sure to note that the response should be normally distributed *at each level of the explanatory variable*. The overall distribution (combining responses across levels of the explanatory variable) does not need to have a normal distribution. In fact, if your explanatory variable successfully predicts your response you would expect that the overall distribution would *not* be normal (as you would be combining distributions possessing different means).

  As a result, we usually examine the normality of the *residuals* for a given model rather than the normality of the response variable itself. For each observation i the residual e_i may be calculated as

  $$e_i = y_i - \hat{y}_i, \tag{3.1}$$

  where y_i is the actual value of the response variable observed in the data and $\hat{y}_i$ is the model's predicted value for that case. Most statistical software packages have options to allow you to easily obtain the residuals for a given regression or ANOVA model (see sections 3.3 and 3.4).

- There are two ways that you can graphically examine the normality of your residuals. First, you can directly examine the distribution of your residuals (such as with a histogram) and compare it to the normal distribution shown in figure 3.1. Alternatively you can examine a *normal probability plot* or a *Q-Q plot*. While slightly different, both plot the values of your distribution against corresponding values from a normal distribution. Normal distributions will produce straight lines in these plots, while non-normal distributions will show significant non-linear deviations.

  There are also several statistical tests of normality. For each, a significant test statistic indicates that your data is not normally distributed. You should be careful, however, when interpreting these

statistics because they are strongly dependent on your sample size. If you have a very large sample these statistics will often be significant even if the distribution is acceptably normal.

- Both ANOVA and regression are relatively robust to violations of normality, so you don't need worry about transforming your response variable if it is slightly irregular. If it is distinctly non-normal, however, you will need to find an appropriate transformation.
- There are generally three things you would want to try if you observe that the residuals of your model have a significantly non-normal distribution.
  1. You can look for an additional variable to include in your model to explain the deviations. If there is an important explanatory variable omitted by your model it might cause your data to appear non-normal. Since we are interested in the normality of the residuals, putting the variable in the model will remove it's influence on normality.
  2. You can check to see if you have one or more outliers that are causing your distribution to appear non-normal.
  3. You can perform a transformation on your response variable. Section 3.2 shows how a number of transformations can take specific patterns of non-normal data and give them normal distributions.

3.2 Commonly Used Transformations

- The transformations discussed in this section are all very standard functions and may be applied using the methods discussed in chapter 2. Each transformation will be presented using the format

$$y' = f(y) \tag{3.2}$$

  where y is the value of the original variable and y' is the value of the transformed variable that you would use in your analyses. These transformations should be applied to your response variable, after which the analysis should be rerun using the new transformed variable.
- Once you have decided that you will need to transform your data you need to decide exactly what transformation you will perform. Below we will see several common patterns of non-normality your data may take, and what transformation you should use for each.
- If your residuals have a moderate right skew, as in figure 3.2, you should apply a square-root transformation to your data.

–Insert Figure 3.2 here–

Figure 3.2: Example of data requiring a square-root transformation.

  This transformation is based on the formula

$$y' = \sqrt{y}. \tag{3.3}$$

  If the pattern of your residuals looks like figure 3.2 but with a more severe skew (such that the residuals still appear skewed after performing a square-root transformation) you can try using a cube-root ($\sqrt[3]{y}$) or fourth-root ($\sqrt[4]{y}$) transformation. Note that neither SPSS nor SAS have specific keywords for these functions. They still can be obtained, however, by raising y to the $\frac{1}{3}$ or $\frac{1}{4}$ power, respectively.
- If your residuals have a severe right skew, as in figure 3.3, you should apply a logarithmic transformation to your data.

  This transformation is based on the formula

$$y' = ln(y). \tag{3.4}$$

–Insert Figure 3.3 here–

Figure 3.3: Example of data requiring a logarithmic transformation.

While formula 3.4 makes use of the natural logarithm, a base 10 logarithm would normalize the data just as well. Logarithmic transformations are commonly used to normalize reaction-time data (Newell & Rosenbloom, 1981).

- If your data has an even more severe right skew, as in figure 3.4, you should perform an inverse transformation.

–Insert Figure 3.4 here–

Figure 3.4: Example of data requiring an inverse transformation.

This transformation is based on the formula

$$y' = \frac{1}{y}. \tag{3.5}$$

- If your residuals have a moderate left skew, as in figure 3.5, you should perform a square transformation.

–Insert Figure 3.5 here–

Figure 3.5: Example of data requiring a square transformation.

This transformation is based on the equation

$$y' = y^2. \tag{3.6}$$

If the pattern of your residuals looks like figure 3.5 but with a more severe skew (such that the data still appear skewed after performing a square transformation) you can try raising y to the third or fourth power.

- Finally, if your residuals have a severe left skew as in figure 4.6, you should perform an exponential transformation.

  This transformation is based on the formula

$$y' = \exp(y). \tag{3.7}$$

- If none of these transformations will normalize your data and you cannot find an appropriate explanatory variable to account for the deviations, you may be in a state where it will be impossible to obtain a normal distribution. Some patterns of data simply cannot be mathematically transformed into normal distributions, such as those that possess two or more modes. In such cases you should probably not use standard ANOVA or regression procedures.

3.3 Using SPSS

- Most of the advanced statistical modeling procedures (including all GLM and regression models) have a **Save** button which causes the procedure to create new variables in the data set. You can use this to have SPSS save the residuals from your model so that you can examine their normality. After selecting **Save** simply click on the type of residuals you wish to examine and a new variable containing them

–Insert Figure 3.6 here–

Figure 3.6: Example of data requiring an exponential transformation.

will be added once the model is run. You will probably be most interested in obtaining either the unstandardized or standardized residuals. The first are the actual residual values while the second subtracts off the mean and divides by the standard deviation of the residuals. This latter option can sometimes be useful for identifying outliers (often defined as cases that have standardized residuals with magnitudes of 3 or greater).

To obtain residuals when using SPSS syntax, you can add a **/save =** option to the model procedure. Following the equal sign you add a code indicating what type of residuals you would like, **resid** for unstandardized residuals, **zresid** for standardized residuals, **sresid** for studentized residuals, or **dresid** for deleted residuals. For example, the following code would be used to obtain all four types of residuals from a GLM.

```
GLM
  rating02 by cond
  /method = sstype(3)
  /intercept=include
  /save = resid zresid sresid dresid
  /criteria = alpha(.05)
  /design.
```

- Once you have obtained the residuals it is a fairly simple matter to plot their distribution. Using SPSS interactive mode you can simply select **Graphs** → **Histogram**, select the appropriate variable, and then click **Ok**. If you are using SPSS syntax you would use the code

  ```
  graph
    /histogram = res_1.
  ```

  assuming that your residuals are contained in the variable **res_1**.

- Normality plots and tests can be obtained using the selection **Statistics** → **Summarize** → **Explore**. You first select the variables you wish to examine from the list on the left side, and then click the arrow button to move them into the **Dependent List**. You then click the **Plots** button and check the box next to **Normality plots and tests**. After that you click the **Continue** button to return to the variable selection screen, and then click **Ok** to get the results.

  If you use SPSS syntax these plots and tests may be obtained using the code

  ```
  examine
    variables=res_1
    /plot npplot.
  ```

  assuming that your residuals are contained in the variable **res_1**.

- The transformations listed in section 3.2 may be obtained in SPSS using the following expressions.
  - **sqrt(y)** for a square root transformation. Cubed and fourth root transformations may be obtained using the expressions **y**(1/3)** and **y**(1/4)**, respectively.
  - **lg10(y)** for a base 10 logarithm or **ln(y)** for a natural logarithm.
  - **1/y** for an inverse transformation.
  - **y**2** for a square transformation. Cubed and fourth power transformations may be obtained using the expressions **y**3** and **y**4**, respectively.
  - **exp(y)** for an exponential transformation.

3.4 Using SAS

- You can obtain the residuals from either **proc glm** or from **proc reg** by adding an **output** line to your model statement as in the example below.

```
data dset1;
     ⋮
(statements reading in data)
     ⋮
proc glm;
   model response=treatmnt;
   output out=dset2 r=resids;
```

  The **output** statement in this example would create a new data set called **dset2** that would contain all of the variables from **dset1** as well as a new variable called **resids** containing the unstandardized residuals. By changing the word following **out=** you can change the name of the new data set. By changing word following **r=** you can change the name of the variable that will hold the residuals. If you would prefer to receive the studentized residuals instead of standardized residuals you can substitute **student=** for **r=** in the **output** statement.

- To obtain histograms, normal probability plots, and normality tests for your residuals you can use the following code

```
proc univariate plot;
   var resid;
```

  where your residuals are assumed to be stored in the variable **resid**.

- The transformations listed in section 3.2 may be obtained in SAS using the following expressions.
  - **sqrt(y)** for a square root transformation. Cubed and fourth root transformations may be obtained using the expressions **y**(1/3)** and **y**(1/4)**, respectively.
  - **log10(y)** for a base 10 logarithm or **log(y)** for a natural logarithm.
  - **1/y** for an inverse transformation.
  - **y**2** for a square transformation. Cubed and fourth power transformations may be obtained using the expressions **y**3** and **y**4**, respectively.
  - **exp(y)** for an exponential transformation.

Chapter 4

Working with Conditionals (if statements)

4.1 Conceptual Overview

- Conditionals allow you to apply a transformation to a select set of cases. There are three basic situations that call for the use of a conditional.

  1. You want to perform a categorical transformation. For example, you might want to create a new categorical variable with two groups from an existing categorical variable with three groups, combining two of the groups together.
  2. You want to create categorical values based on the values of an existing numeric variable. For example, you may want to create a categorical variable that indicates whether a particular case was above or below the mean on a numeric variable.
  3. You wish a numerical transformation to vary with a categorical distinction. For example, say you have responses on a numeric variable for cases in three different groups. For each case you might want to subtract off the mean value of its group. Different cases would need different transformations depending on which group they are in.

- To create a conditional transformation you need to define a *criterion expression* and an *outcome operation*. The criterion is logical expression that determines the cases that you want included in the transformation. This can be either an equality, such as

$$\text{sex} = \text{“male”}, \tag{4.1}$$

  or an inequality, such as

$$\text{age} < 40, \tag{4.2}$$

  defined in terms of existing variables in your data set. The outcome operation defines the data manipulation you wish to be applied to cases that meet the criterion. This is typically a transformation expression as discussed in chapter 2, but can be a more complex set of procedures as well.

- It is possible to design more complicated criteria by using *boolean operators*. There are three major boolean operators.

  1. *AND* allows you to specify two different expressions, both of which much be satisfied for the criterion to be met. For example, in a data set dealing with medical outcomes the criterion

$$(\text{age} > 25) \text{ AND } (\text{sex} = \text{“male”}) \tag{4.3}$$

     would only be met if the patient in question was a person who was both male and over 25. A conditional transformation using this criterion would not be applied to women of any age or to any person aged 25 and under.

2. *OR* allows you to specify two expressions where the satisfaction of either allows the criterion to be met. For example, in the same data set the criterion

$$(\text{health} = \text{“poor”}) \text{ OR } (\text{age} > 65) \tag{4.4}$$

would be met if the patient had poor health, was over 65, or both. A conditional transformation using this criterion would be applied to all cases except those where the patient was both in good health and under the age of 65.

3. *NOT* allows you to establish a criterion based on some expression being not true. For example, in the medical data set the criterion

$$\text{NOT}(\text{condition} = \text{“cancer”}) \tag{4.5}$$

would be met if a patient had any condition other than cancer. A conditional transformation using this criterion would be applied to all cases except those where the patient had cancer.

If you find yourself working with a complicated boolean expression you should use make liberal use of parenthesis to ensure that your program interprets the expression in the way that you want. For example, the expression (A=1 OR B=1 AND C=1) is not the same as (A=1 OR (B=1 AND C=1)).

- In some cases a conditional can include an *alternative operation*, indicating procedures that are be applied to cases that fail to satisfy the criterion expression. A transformation is applied to every case using such statements. The outcome operation is performed if the case meets the criterion, while the alternative operation is performed if it does not. Like the outcome operation, the alternative operation is typically a transformation expression, but it can also be a more complex set of procedures.

- Conditionals are sometimes referred to as *if-then-else* or simply *if* statements because of the structure they typically take in computer programs. The structure typically follows the form

  if (*criterion expression*) then (*outcome operation*) else (*alternative operation*).

  This form is similar to that used by both SPSS and SAS, so we will use it in the generic examples (i.e., not dealing specifically with either piece of software) in these notes.

- Often times you will need a set of related conditional statements to perform the transformation you desire. For example, you might issue the following statements to create a categorical variable *agecat* from a numerical variable *age*.

  $$\begin{array}{ll} \text{if (age} < 20) & \text{then (agecat} = 1) \\ \text{if (age} \geq 20) \text{ AND (age} < 35) & \text{then (agecat} = 2) \\ \text{if (age} \geq 35) \text{ AND (age} < 50) & \text{then (agecat} = 3) \\ \text{if (age} \geq 50) \text{ AND (age} < 65) & \text{then (agecat} = 4) \\ \text{if (age} \geq 65) & \text{then (agecat} = 5) \end{array}$$

  This transformation only makes sense if all of these statements are used together. If you are missing a single one then your new variable *agecat* would be incomplete (and probably not useful for analysis).

- Sometimes you may want to create a single categorical variable representing the combination of two (or more) other categorical variables. For example, in an ANOVA you might have a variable a representing a factor with 2 levels and a variable b representing another factor with 3 levels. If you wanted to perform simple comparisons between all of these means you would need to create a variable with 6 levels representing the various combinations of a and b. If you have a and b coded numerically (e.g., cases in level 1 of the first factor have a value of 1 for a, cases in level 2 of the first factor have a value of 2 for a, etc.) and both variables have less than 10 levels there is a handy shortcut for creating composite variables. You simply perform the transformation

  $$x = (10 * a) + b. \tag{4.6}$$

In this example x would take on the values of 11, 12, 13, 21, 22, and 23. The first digit would represent the level of the first factor while the second digit represents the level of the second factor. There are two benefits to using this transformation. First, it can be completed using a single line of code, where assigning the groups manually would likely take at least 6. Second, you can directly see the correspondence between the variable x and the variables a and b by looking at the digits of x. If we had recoded x such that it had values between 1 and 6 (which would work just as well for analysis) we would likely need to reference a table to know the relationships between x and the original factors a and b.

If you ever want to create a composite of three or more variables you follow the same general principle, basically multiplying each variable by decreasing powers of 10, such as

$$x = (100 * a) + (10 * b) + c \tag{4.7}$$

or

$$x = (1000 * a) + (100 * b) + (10 * c) + d. \tag{4.8}$$

4.2 Using SPSS

- You may perform conditional transformations in SPSS interactive mode using either the selection **Transform** $\rightarrow$ **Compute** or the selection **Transform** $\rightarrow$ **Recode**. With **Compute** you can specify a single transformation that is applied to a subset set of cases defined by a conditional expression. Section 2.2 discussed using **Compute** to perform transformations. To make a given transformation conditional all you need to do is to click the **If** button, click the button next to "Include if case satisfies condition," enter your criterion expression in the box, and then click **Continue**. If you like you can include boolean operators (AND, OR, NOT) your criterion. Back on the first screen you will now see your criterion expression next to the **If** button. When the transformation is completed (by clicking **Ok**) it will only be applied to those cases that meet the criterion expression.

- **Recode** is somewhat more complicated than **Compute** but is better at performing certain types of transformations. Specifically, you should use **Recode** rather than **Compute** when you are transforming a categorical variable, or when you want to create a categorical variable based on the values of a numerical variable.

  Once you select **Recode** you will be asked whether you want to recode **Into same variables** or **Into different variables**. The first selection causes your transformation to replace an existing variable, while the second selection will create a new variable to hold the results of the transformation. If you choose to recode **Into same variables** SPSS will produce a dialog box allowing you to select the variables to be transformed. If you choose to recode **Into different variables** you will see a similar variable selection box, but after selecting the variables to be transformed you must then indicate the names of the variables that will hold the results of the transformation. To do this you must click on each variable in the **Input Variable -> Output Variable** box, enter the name of the variable that will hold the transformation in the **Output Variable** box, and then press the **Change** button.

  When the target variables are selected you click the **Old and New Values** button to describe the transformation. Here you define what values the new variables should take based on the values of the old variables. After entering a value or set of values on the left side of the dialog box you enter the corresponding value of the new variable on the upper-right side of the dialog box and press the **Add** button. You can add as many old-new value pairs as you like. Once you have defined all of your transformations you press the **Continue** button to go back to the first screen.

  If you want to restrict the recode to a set of cases based on a criterion expression you may click the **If** button and enter that expression. Once you are finished you click the **Ok** button to have SPSS apply the transformation.

- There are two functions in SPSS syntax that allow you to perform conditional transformations. For simple tasks you can use the **if** statement. The format of the **if** statement is

if (*criterion expression*) *variable* = *value*.

The *value* can either be a constant (such as a number, letter, or word) or another variable. For example, all of the following would be valid uses of the **if** statement.

if (company = 'Ford') compnum=12.
if (age > 20 AND sex = 1) group=2.
if (change = 0) newrate=oldrate.

- If your outcome operation requires several statements or you wish your conditional to include an alternative operation you will need to use a combination of **do if** and **end if** statements. The format for a **do if - end if** combination without an alternative operate is

do if *criterion expression*.
-statements of your outcome operation here-
end if.

You may include as many statements between the **do if** and the **end if** statement as you like. These statements will only be performed if the case matches the criterion expression. If it does not they will all be skipped. If you want to include an alternative operation the format is

do if *criterion expression*.
  -statements of your outcome operation here-
else.
  -statements of your alternative operation here-
end if.

Here the statements of the outcome operation will be performed if a given case matches the criterion, while the statements of the alternative operation will be performed if it does not.

4.3 Using SAS

- All conditional transformations in SAS are performed using the **if** statement. If you want to perform a simple conditional transformation the syntax is

if (*conditional expression*) **then** *variable* = *value*;

The *value* can either be a constant (such as a number, letter, or word) or another variable. For example, all of the following would be valid uses of the **if** statement.

if (company = 'Ford') then compnum=12;
if (age > 20 AND sex = 1) then group=2;
if (change = 0) then newrate=oldrate;

- If you wish to use a conditional including an alternative operation you would use the following syntax.

if (*conditional expression*) **then** *variable1* = *value1*;
else *variable2* = *value2*;

While we use the terms *variable1* and *variable2* to indicate that these two may be different variables, but they do not have to be. For example, both of the following are valid uses of the **if** statement.

if (major = 'Psychology') then psym = 1;
  else psym = -1;

if (school = 'Graduate School') then grad = 1;
  else undgrad = 1;

- If you wish either the outcome operation or the alternative operation to be more complicated you can replace variable assignment in the above examples with a **do** statement. For example,

```
if answer=10 then do;
   correct = 1;
   wrong = 0;
end;
else do;
   correct = 0;
   wrong = 1;
end;
```

  For more information on how to use the **do** statement see the *SAS Language Guide*.

Chapter 5

Working with Arrays and Loops

5.1 Conceptual Overview

- Arrays and loops are two tools drawn from computer programming that can be very useful when manipulating data. Their primary use is to perform a large number of similar computations using a relatively small program. Some of the more complicated types of data manipulation can only reasonably be done using arrays and loops.
- An *array* is a set of variables that are linked together because they represent similar things. The purpose of the array is to provide a single name that can be used to access any of the entire set of variables.
- A *loop* is used to tell the computer to perform a set of procedures a specified number of times. Often times we need to perform the same transformation on a large number of variables. By using a loop we only need to define the transformation once, and can tell it to do the same thing to all the variables using a loop.
- If you are familiar with these terms from a computer programming course you are a step ahead. Arrays and loops are used in data manipulation in more or less the same way that they are used in standard computer programming.

5.2 Specifics of Using Arrays

- Before you can use an array you first need to define it. This typically involves specifying the name of the array and listing what variables are associated with the array. Variables referenced by an array are called *elements* of that array.
- Arrays are used in transformation statements just like variables. However, the array itself isn't able to hold values. Instead, the array acts as a middle-man between your statement and the variables it references. The variables included in an array are given a specific order. Whenever you use an array you always provide it with an *index*. The index tells the array exactly which one of its elements you want to work with. You do not need to know what the exact name of the variable is - you just need to know its location in the array. References to items within an array are typically made using the format

  $$aname\ (index),$$

  where *aname* is the name of the array and *index* is the numerical position of the desired element.
- As an example let's create an array for a group of variables designed to keep track of the contents of an actual office cabinet. The name of the array will be C. We will assume that there are four different

shelves inside the cabinet and that only one thing will be kept on a shelf at a time. We create four different variables, called $s1, s2, s3$, and $s4$ to hold the name of whatever item is currently on each of the four shelves. We will assume that the variables are ordered in the array in the same way that the shelves are ordered in the cabinet (so that $s1$ is the first element, $s2$ is the second element, etc.). To start with let us assume that the four shelves hold paper, computer disks, envelopes, and pencils, respectively. Table 5.1 summarizes how the array structures correspond to the real life objects in this example.

Table 5.1: Structures in cabinet example

Structure Name	**Structure Type**	**Real-life Object**
C	Array	Cabinet
$s1, s2, s3, s4$	Variables	Shelves inside cabinet
"paper", "disks", "envelopes", "pencils"	Values of variables	Objects on the shelves

If we performed the transformation

$$x = C(1), \tag{5.1}$$

we would find that the value of x would be "paper." Similarly, if we performed the transformation

$$C(3) = \text{"staples"}, \tag{5.2}$$

we would find that the value of $s3$ had changed to "staples." As you can see, the array mediates the interaction between you and the variables. Remember that arrays themselves are not variables. Rather, they provide you with another way for you to access your variables.

- Right now you might find yourself asking "What's the point?" On the surface there doesn't seem to be any advantage to referring to a variable as $C(1)$ rather than as $s1$. The advantage is that an array index can be any mathematical expression that resolves to an integer, as long as it is less than or equal to the number of variables in the array. This means that the specific element accessed by an array can be dependent on other variables. For example, we could write a transformation statement like

$$item = C(curshelf). \tag{5.3}$$

In this case the specific element called by C would depend on the value of $curshelf$. If $curshelf$ took on different values then repeating this statement would perform different transformations.

The index does not even have to be a single variable. Indices can also be more complicated expressions such as in the following examples.

$$\begin{gathered} itemnum = invar(t * 3 + 1) \\ status = statarr(a * 4 + b) \\ nextitem(i) = item(i + 1) \end{gathered} \tag{5.4}$$

The index can be any mathematical expression as long as it evaluates to a positive integer and that integer is less than or equal to the number of elements in the array.

- The use of arrays is not limited to transformations. You can use an array reference anywhere that you could use the variables contained inside the array. This includes the conditional expressions discussed in chapter 4.

- Variables that are declared to be part of an array can still be accessed and changed through normal methods. The array provides an additional way to read or change the variable without replacing the normal transformation methods. So even after declaring that the variable $s2$ was an element of the C array in the cabinet example we could issue a statement in our program like

$$s2 = \text{"pens"}. \tag{5.5}$$

It is even possible to use an array reference and the normal variable reference in the same statement, such as the following conditional transformation.

$$\text{if } (C(1) = \text{“pens”}) \text{ then } s2 = \text{“paper”} \tag{5.6}$$

5.3 Specifics of Using Loops

- A typical loop declaration looks something like the following.

  loop *lvar* = *lowlimit* to *uplimit*
  -some other programming statements-
  end loop

  The **loop** statement indicates the beginning of the loop. The way a loop works is that it starts out by assigning the loop variable *lvar* the value of *lowlimit*, the “lower limit” of the loop. The program then executes the following lines until it reaches the **end loop** statement. At this point the value of the loop variable is increased by 1 and compared to the value of *uplimit*. If it is less or equal to *uplimit*, the “upper limit” of the loop, the program goes back to first line following the **loop** statement, repeating all the statements inside of the loop. If it is greater than *uplimit* the program proceeds to the line following the bf end loop statement, leaving the loop. Any statements in between the **loop** and **end loop** statements are therefore performed a total of (*uplimit* - *lowlimit* + 1) times. Each of these repetitions is called a *cycle* of the loop.

- The following program is one of the simplest examples of a loop.

  ```
  count = 0
  loop t = 1 to 5
    count = count + 10
  end loop
  ```

  The first line simply initializes the variable *count* to the value of zero. The second line defines the conditions of the loop. The loop variable is named t and starts with a value of 1. The loop will continue to cycle until the value of t is greater than 5. This causes the program to perform a total of 5 cycles. Since value of the variable *count* is increased by 10 during each cycle, at the end of the program it has a value of 50.

  Just in case you were wondering, the first statement initializing **count** is actually necessary for the sum to be calculated. Most programming languages, including both SPSS and SAS syntax, start off variables with missing values. Adding anything to a missing value produces a missing value, so we must explicitly start the variable *count* at zero to be able to obtain our sum.

- Although the lower and upper limits have been constants in our examples so far, they can also be numeric variables. For example, the loop below uses different limits depending on the value of the variable **group**.

  ```
  if (group = 'A') then members = 12
  if (group = 'B') then members = 14
  if (group = 'C') then members = 10
  grouppay = 0
  loop t = 1 to members
    grouppay = grouppay + 25
  end loop
  ```

- Remember that the value of the loop variable is changed in each cycle. This can sometimes be used as part of your program. For example, the following program can be used to calculate the sum of the squares of the numbers between 5 and 10.

```
sqsum = 0
loop i = 5 to 10
  sqsum = sqsum + (i * i)
end loop
```

This loop will cycle a total of 6 times. The value of the variable *sqsum* will be the sum of 25, 36, 49, 64, 81, and 100, the squares between 5 and 10.

- There is very little that is special about the loop variable other than the fact that it gets initialized and then incremented by the loop. You can even change its value using normal transformation statements. However, you need to be careful anytime you manually make changes to the loop variable because you can accidentally set up a situation where the loop variable never exceeds the upper limit. In such a situation, called an *infinite loop*, the computer program will get stuck inside the loop.
- It is possible to have a loop inside of another loop. This is called *nesting* two loops. In this case the loop on the inside is repeated during each cycle of the outer loop. Consider the following example.

```
sum = 0
loop lvar1 = 1 to 3
  loop lvar2 = 5 to 6
    sum = sum + lvar1*lvar2
  end loop
end loop
```

In this program the *lvar2* loop (with two cycles) is executed during each of the three cycles of the *lvar1* loop. As a result, the summation in line 4 is performed a total of six times, adding up the numbers 5, 6, 10, 12, 15, and 18.

An **end loop** statement (or its equivalent in whatever programming language you use) always ends the innermost loop that is still running. In line 5, neither the *lvar1* loop nor the *lvar2* have been completed, so the **end loop** statement is used to end the *lvar2* loop which is inside the *lvar1* loop. By line 6 the *lvar2* loop is completed, so the **end loop** statement is used to end the *lvar1* loop.

Nested loops must use different names for their loop variables.

5.4 The Power of Combining Arrays and Loops

- While you can work with arrays and loops alone, they were truly designed to be used together. A combination of arrays and loops can save you incredible amounts of time when performing certain types of repetitive transformations.
- Consider the characteristics of arrays and loops. An array lets you reference a set of related variables using a single name and an index. The index can be a variable or a mathematical expression involving one or more variables. A loop repeatedly performs a set of commands, incrementing a loop variable after each cycle. What would happen if a statement inside of a loop referenced an array using the loop variable as the index? During each cycle the loop variable increases by 1, so during each cycle the array would refer to a different variable. If you correctly design the upper and lower limits of your loop, you could use a loop to perform a transformation on every variable composing an array.
- For an example, let's say that you conducted a reaction-time study where research participants observed strings of letters on the screen and judged whether they composed a real word or not. In your study you had a total of 200 trials in several experimental conditions. You want to analyze your data with an ANOVA to see if your the reaction time varies by condition, but you find that the data has a right skew (which is common). To use ANOVA you will need to transform the data so that it has a normal distribution (see chapter 3) which involves taking the logarithm of the response time to each trial.

  In terms of your data set, what you need is a set of 200 variables whose values are equal to the logarithms of the 200 response time variables. Without using arrays or loops you would need to write

200 individual transformation statements to create each log variable from the corresponding response time variable. Using arrays and loops, however, we can do the same work with the following simple program. The program assumes that the original response time variables are *rt001* to *rt200* and the desired log variables will be *lrt001* to *lrt200*.

```
array Rtarray = rt001 to rt200
array Larray = lrt001 to lrt200
loop item = 1 to 200
  Larray(item) = log(Rtarray(item))
end loop
```

The first two statements set up a pair of arrays, one to represent the original response time variables and one to represent the transformed variables. The third statement creates a loop with 200 cycles. Each cycle of the loop corresponds to a trial in the experiment. The fourth line actually performs the desired transformation. During each cycle it takes one variable from *Larray* and sets it equal to the log of the corresponding variable in *Rtarray*. The fifth line simply ends the loop. By the time this program completes it will have created 200 new variables holding the log values that you desire.

- In addition to greatly reducing the number of programming lines, there are other advantages to performing transformations using arrays and loops. If you need to make a change to the transformation you only need to change a single statement. If you write separate transformations for each variable you must change every single statement anytime you want to change the specifics of the transformation. It is also much easier to read programs that use loops than programs with large numbers of transformation statements. The loops naturally group together transformations that are all of the same type, while with a list you must examine each individual transformation to find out what it does.

5.5 Using SPSS

- SPSS refers to arrays as *vectors*, but they otherwise operate just as described in this chapter.
- If the variables inside a vector have already been declared (such as by using a **numeric** or **string** statement) you declare a vector using the following syntax.

  vector *Vname* = *variable list*.

  The vector is given the name *Vname* and is used to reference a set of variables defined by the variable list. The list must be a sequentially ordered set of variables (such as $x1$, $x2$, $x3$, $x4$ or $item08$, $item09$, $item10$) and must be declared using the syntax "$varX1$ **to** $varX2$", where $X1$ is the number of first variable in the vector and $X2$ is the number of the last. For example, the following are all acceptable vector declarations.

  ```
  vector V = v1 to v8.
  vector Myvector = entry01 to entry64.
  vector Grade = grade1 to grade12.
  vector Income = in1992 to in2000.
  ```

  Note that the index used by a vector only takes into account the number of elements in the vector - not the names of the variables. To reference the variable **in1993** in the **Income** above you would use the phrase **income(2)**, *not* **income(1993)**.
- If the variables in the array have not already been declared you can do so as part the **vector** statement. For more information on this see page 904 of the *SPSS Base Syntax Reference Guide*.
- Some notes on naming vectors and variables:

- In these notes we will follow the convention of starting vector names with capital letters and variable names with lower-case letters. This is not required, but it helps prevent confusion as to which names refer to vectors and which refer to variables.
- Often times you will want to give your variable names the same stem as the vector name to make them easier to identify. However, this is not required by SPSS.
- All variables in SPSS must have eight or fewer characters, including the number at the end. So **myvarib01** would be unacceptable as a variable name.
- When you are numbering variables you can decide to use leading zeros (such as **var001 to var300**) or not (such as **var1 to var300**). If you use leading zeros in your declaration then SPSS will use leading zeros in all the variable names. If your declaration does not use leading zeros then your variables won't have them either. In general it is a good idea use leading zeros because variables are often sorted in alphabetical order. If you do not use leading zeros then your list of variables will appear out of order at such times.

- To reference a variable inside of a vector you type the vector name followed by the index inside a set of parentheses. For example, the following are all valid uses of vectors.

```
compute V(6) = 9.
compute yval = Myvector(xval).
if (Grade(current)=Grade(current-1)) change=0.
if (year<1996) Income(year-1991)=Income(year-1991)-1500.
```

- To specify a loop in SPSS you use the following syntax.

loop *loopvar* = *n* **to** *m*.
-some other programming statements-
end loop.

Here *loopvar* is the name of your loop variable, n is the lower limit of the loop and m is the upper limit of the loop.

- The following is an example of an SPSS program that uses a loop to calculate a sum.

```
compute x = 0.
loop #t = 1 to 8.
+    compute x = x+#t.
end loop.
```

In this example the loop variable t is denoted as a *scratch variable* by putting a number sign (#) in front of it. When something is denoted as a scratch variable in SPSS it is not saved in the final data set. Typically we are not interested in storing the values of our loop variables so it is common practice to denote them as scratch variables. For more information on scratch variables see page 32 of the *SPSS Base Syntax Reference Guide*.

You will also notice the plus sign (+) placed before the compute statement in line 3. SPSS needs you to start all new commands in the first column of each line. Here we wish to indent the command to indicate that it is part of the loop. We therefore put the plus symbol in the first column which tells SPSS that the actual command starts later on the line.

- The following is an example of an SPSS program that uses vectors and a loop to perform a number of transformations.

```
numeric lrt001 to lrt200.
vector Rtarray = rt001 to rt200.
vector Larray = lrt001 to lrt200.
loop #item = 1 to 200.
+    compute Larray(#item) = lg10(Rtarray(#item)).
end loop.
```

Notice that line 1 declares the variables **lrt001** to **lrt200** that will be used to hold the transformed values. SPSS requires that variables must be declared before they can be assigned to a vector. We didn't declare the variables **rt001** to **rt200** because they must already exist in the data set for us to be able to perform the transformations.

5.6 Using SAS

- To declare an array in SAS you use the syntax

  array *Aname*[*size*] *variable list***;**

  where *Aname* is the name of the array, *size* is the number of variables contained in the array, and the variable list specifies exactly what variables are contained in the array. If you want you can put an asterisk (*) in place of *size* and SAS will count the number of variables you list to determine the size of the array. This can be particularly useful if you think you might change the number of variables in the array at a later point in time. If your variables are sequentially ordered (such as $x1$, $x2$, $x3$, $x4$ or $item08$, $item09$, $item10$) you can list them using the syntax "$varX1 - varX2$", where $X1$ is the number of first variable in the array and $X2$ is the number of the last. If they are not then you simply write out the names of the variables included in the array. The order of variables inside the array is the same order as presented in the variable list. The following are all acceptable array declarations.

  array Greek[4] alpha beta gamma delta;
  array Myarray[*] entry01-entry64;
  array Grade[12] grade1-grade12;
  array Income[*] in1992-in2000;

  Note that the index used by a vector only takes into account the number of elements in the vector - not the names of the variables. To reference the variable **in1993** in the **Income** above you would use the phrase **income(2)**, *not* **income(1993)**.

- If you want to declare an array of string variables you must place a dollar sign ($) before the variable list, as in the following example.

  array Names[*] $ name1-name5;

- Some notes on naming arrays and variables:

  - In these notes we will follow the convention of starting array names with capital letters and variable names with lower-case letters. This is not required, but it helps prevent confusion as to which names refer to arrays and which refer to variables.
  - Often times you will want to give your variable names the same stem as the array name to make them easier to identify. However, this is not required by SAS.
  - All variables in SAS must have eight or fewer characters, including the number at the end. So **myvarib01** would be unacceptable as a variable name.
  - When you are numbering variables you can decide to use leading zeros (such as **var001-var300**) or not (such as **var1-var300**). If you use leading zeros in your declaration then SAS will use leading zeros in all the variable names. If your declaration does not use leading zeros then your variables won't have them either. In general it is a good idea to use leading zeros because variables are often sorted in alphabetical order. If you do not use leading zeros then your list of variables will appear out of order at such times.

- To reference a variable inside of an array you type the array name followed by the index inside a set of parentheses. For example, the following are all valid uses of arrays.

```
Greek[3] = 4;
yval = Myarray[xval];
if (Grade[current]=Grade[current-1]) then change=0.
if (year<1996) then Income[year-1991]=Income[year-1991]-1500.
```

- To specify a loop in SAS you use a pair of **do** and **end** statements. The syntax is as follows:

  do *loopvar* = *n* **to** *m*;
  -some other programming statements-
  end;

  Here *loopvar* is the name of your loop variable, n is the lower limit of the loop and m is the upper limit of the loop.

- The following is an example of a SAS program that uses a loop to calculate a sum.

```
x = 0;
do t = 1 to 8;
   x = x + t;
end;
```

- The following is an example of a SAS program that uses arrays and a loop to perform a number of transformations.

```
array Rtarray[*] rt001-rt200;
array Larray[*] lrt001-lrt200;
do item = 1 to 200;
   Larray[item] = log(Rtarray[item]);
end;
```

Chapter 6

Restructuring Data: Changing the Unit of Analysis

6.1 Conceptual Overview

- Sometimes a single transformation or a series of transformations will not be enough to make the changes that you want. The goal of your data manipulation may require more fundamental changes, requiring a full restructuring of your data set.
- Restructuring data involves changing the unit of analysis of your data set, followed by the creation of variables that are meaningful to the new unit. There are two basic ways that you can change the unit of analysis.
    - You can move to a smaller unit of analysis. Here you take each case in your original data set and break it down into several different cases for your new data set.
    - You can move to a larger unit of analysis. Here you build each case for your new data set by combining the information from several cases in your original data set.

6.2 Converting to a Smaller Unit of Analysis

- Moving to a smaller unit of analysis is relatively simple. You only need to use information from a single case in the original data set to create each case in the new data set. Additionally, you can typically keep a reasonable portion of the variables from the original data set without changes.
- In the language used by the regression and ANOVA, this is referred to as converting data from multivariate format to univariate format.
- To help illustrate the process of moving to a smaller unit of analysis, consider a data set containing information about the distribution of employees in companies. The cases of the data set represent different companies. Variables within the data set refer to the size of the production, distribution, and sales departments, as well as a region code indicating the geographical location of the company. Table 6.1 shows a possible excerpt from this data set. For this illustration let us assume that you want to convert this data set to one where each case represents a department within a company, rather than a full company. We will also assume that we want the new data set to contain all the information present in the original.
- The first thing you need to do is precisely specify the unit of analysis for the new data set. You must decide how you are going to derive the cases of the new data set from the cases of the old data set, as well as how many cases in the new data set will be derived from each case in the original data set. Usually each case in the original will correspond to the same number of cases in the new data set,

Table 6.1: Employee Data Set 1

Case	Company	Region	# in Production	# in Distribution	# in Sales
1	Janice's Horse Feed	1	25	10	10
2	Shoes R Us	2	100	35	20
3	International House of Pots	2	10	0	10
4	Fried Fish of Faradole	1	150	80	25

although this is not strictly necessary. Each of the cases in the new data set must, however, have the same unit of analysis.

In our example we have decided that each case in the new data set should represent a single department from a company. The original data set contained information about three departments for each company, so each case in the original data set will be used to create three cases in the new data set.

- Next you need to define the variables for the new data set. There are two different classes of variables that you might create when moving to a smaller unit of analysis.
  - Variables whose values only vary *between* the observations of the original data set. Cases in the new data set that are drawn from the same observation in the original data set will all have the same value on these variables. Constructing these variables is typically easy because there will be usually be a variable in the original data set whose values you can directly copy to your new variable.
  - Variables whose values vary *within* (as well as between) observations of the original data set. Cases in the new data set that are derived from the same observation in the original data set will *not* necessarily have the same values on these variables. While the values of these variables will still be drawn from the original data set, different cases in the new data set will draw their values from different variables in the original data set.

  In the employee example we would want our new data set to include variables indicating the company name and the region code, both examples of the first class of variables. We will be able to take the values for our new variables directly from the variables **company** and **region** in the original data set. We would also want variables holding the division name and the size of the division, examples of the second class. The value for the division size will be drawn from either the **# in Production**, **# in Distribution**, or the **# in Sales** column, depending on the specific division represented by the new case.

- Once you have defined both the unit of analysis and the variables of the new data set you should know what the final data set should look like. Figure 6.2 shows the final version of the employee data set, including all of the new variables we discussed.

- The last thing you will need to do is program your statistical software to restructure your data set. Specific examples of programs can be found in sections 6.4 and 6.5.

6.3 Converting to a Larger Unit of Analysis

- This is usually more difficult than moving to a smaller unit of analysis, mainly because you need to combine the information from several cases of the original data set to create each case in the new data set.

- In the language used by regression and ANOVA, this is referred to as converting data from univariate format to multivariate format.

Table 6.2: Employee Data Set 2

Case	Company	Division	Region	# in Division
1	Janice's Horse Feed	Production	1	25
2	Janice's Horse Feed	Distribution	1	10
3	Janice's Horse Feed	Sales	1	10
4	Shoes R Us	Production	2	100
5	Shoes R Us	Distribution	2	35
6	Shoes R Us	Sales	2	20
7	International House of Pots	Production	2	10
8	International House of Pots	Distribution	2	0
9	International House of Pots	Sales	2	10
10	Fried Fish of Faradole	Production	1	150
11	Fried Fish of Faradole	Distribution	1	80
12	Fried Fish of Faradole	Sales	1	25

- To help illustrate the process of moving to a larger unit of analysis, consider a data set holding the responses to a public opinion survey. In this data set each case represents an individual's response to a particular question on a 10-item survey. The data set contains variables relating to the question and the individual, as well as the response. Table 6.3 shows a possible excerpt from this data set. For this

Table 6.3: Survey Data Set 1

Case	Subject	Condition	Question	Response
1	1	*A*	1	6
2	1	*A*	2	1
⋮	1	⋮	⋮	⋮
10	1	*A*	10	4
11	2	*B*	1	2
12	2	*B*	2	3
⋮	2	⋮	⋮	⋮
20	2	*B*	10	5
21	3	*A*	1	7
22	3	*A*	2	7
⋮	3	⋮	⋮	⋮
30	3	*A*	10	1

  illustration let us assume that you want to convert this data set into one where each case represents an individual, rather than a specific response. We will also assume that we want the new data set to contain all the information present in the original.

- The first thing you need to do is specify the unit of analysis for the new data set. This will be some superordinate grouping of the observations in the old data set, something that sets of observations will have in common. You will typically combine the same number of cases in the old data set to create each case in the new data set, although this is not strictly necessary. In the survey example we decided that each case in the new data set should represent a respondent. There are 10 cases in the original data set for each respondent: one for each item on the questionnaire. This means that we will be using the information from 10 observations in the old data set to create each observation in the new data set.

- Next you will need to create the variables in the new data set. Each variable in the old data set whose information you wish to keep will become one or more variables in the new data set, depending on how the old variable is distributed in the original data set.
    - If the values of the old variable only vary *between* the new superordinate grouping (such that old observations contributing to the same new case always have the same value on the variable) you will only need a single variable in the new data set to hold the information from the old variable. You can obtain the value of this new variable from the value of the old variable in any of the original observations being combined. It doesn't matter which you choose because the old observations being combined all have the same value.
    - If the values of the old variable vary *within* (as well as between) the new superordinate grouping (such that old observations contributing to the same new case can have different values on the variable) you will need a set of variables, one for each observation being combined, to hold the information from the old variable. You must then go through each of the original observations and assign its value on the old variable to one of the new variables.

  In the survey example we would want our new data set to include variables representing the subject number and the condition, which should only need one variable in the new data set. We would also want to keep the information about the responses to the questions. This will take 10 different variables, one for each question item. We will assign the responses to particular variables based on the question number so we will not need to keep the information in the old **Question** variable.

  Sometimes you wish to construct a variable for your new data set that is a numerical summary of a variable that varied within the new superordinate grouping in the old data set. For example, we might want to include a variable in the new data set representing a subject's average response. The easiest way to construct this variable would be to first create the 10 variables holding the item responses and then afterwards calculate the summary statistic.

- Once you have defined both the unit of analysis and the variables of the new data set you should know what the final data set should look like. Figure 6.4 shows the final version of the survey data set, including all of the new variables we discussed.

Table 6.4: Survey Data Set 2

Case	**Subject**	**Condition**	**Q1**	**Q2**	$\cdots$	**Q10**	**Mean Response**
1	1	*A*	6	1	$\cdots$	4	3.9
2	2	*B*	2	3	$\cdots$	5	2.9
3	3	*A*	7	7	$\cdots$	1	5.6

- The last thing you will need to do is program your statistical software to restructure your data set. Specific examples of programs can be found in sections 6.4 and 6.5.

6.4 Using SPSS

- Restructuring data sets can only be performed using SPSS syntax. The procedures involved simply require too many specifications to be implemented in a menu. Additionally, it can only be performed within an *input program*. This is a section of an SPSS syntax file that creates a data set. This means that you can only restructure data when you are first reading it in. You cannot restructure an existing data set (one that is saved as a ".sav" file or that is already instantiated in the Data Editor). If you need to restructure something in these formats you will need to first save the data set as a text file and then reload it using an input program, during which you can perform the necessary restructuring. More information about working with an input program can be found on pages 484-487 of the *SPSS Base Syntax Reference Guide*.

- This section provides examples demonstrating how to restructure data sets in SPSS. The first example shows how to move to a smaller unit of analysis using the employee data, while the second example shows how to move to a larger unit of analysis using the survey data. For each example we will go through two programs that can be used to restructure the data set, one performing the transformation without loops and vectors and one performing the transformation using loops and vectors. It is easier to understand what is going on with the first program but it is generally more efficient to use more advanced programming techniques, especially if you have a large number of variables in your data set.

- Before trying to restructure a data set you must be aware of two commands from SPSS syntax that are important to their operation.

    - **leave**. This command is used to tell SPSS not to re-initialize a variable during the input program. By default SPSS resets every variable to missing each time it reads in a new record, as well as each time it encounters an **end case** statement (described below). Variables listed in a **leave** statement only change if specifically reassigned by the programmer. If you want to retain the value of a variable across several records or cases you will need to include it in a **leave** statement.

    - **end case**. This command tells SPSS to create a case in the data set using the current variable values. If you do not explicitly use an **end case** command SPSS creates the case at the end of the input program, causing it to create one case from each record it reads. By explicitly stating where SPSS should create a case, however, you can instead have it create several cases from a single record or have it read several records before it creates a case.

- The program in figure 6.1 reads in the employee data in the form shown in table 6.1 and converts it to the data set shown in table 6.2.

  The very first line starts the input program. Lines 2-3 define a set of scratch variables to read in the original data. Lines 4-5 define the four variables that we will want to include in our final data set holding the company name, department name, region number, and the division size. Line 6 is used to read in the data records.

  The second set of commands (following the first blank line) do the actual work of constructing the cases for the new data set. Lines 1-5 are used to create a case containing information about the production division. Line 1 assigns the company name, line 2 assigns the region code, line 3 records the name of the division and line 4 records the size of the division. Line 5 tells SPSS to create a new case using the current values of the variables. In a similar way, lines 6-10 are used to create a case containing the information from the distribution division and lines 11-15 are used to create a case containing the information from the sales division. Three new cases are therefore created each time an observation from the original data set is read. The last line in this section indicates the end of the input program.

  At this point the data set now looks just like the data in table 6.2. The last part of the program simply contains the data that was used in this example. In your own programs you may decide to read your records from an external file, in which case you would omit these statements.

- The program in figure 6.2 is basically the same thing as the program in figure 6.1 but uses more advanced programming techniques to simplify the code.

  The very first line starts the input program. Lines 2-3 define the loop variable, a set of scratch variables to hold the size of the divisions, and an array to reference the division sizes. Lines 4-8 define a set of variables that will hold the names of the departments and a vector to refer to those variables. Lines 9-10 define the variables that we will want in the final version of the new data set. Line 11 is used to read in the data records.

  Line 1 in the second set of commands (following the first blank line) tells SPSS to retain the values of **company** and **region** through the processing of the entire record. Normally these variables would be re-initialized each time an **end case** statement was executed. Instead of reading these values in with one set of variables and then assigning them to new variables as we did in figure 6.1, this time we simply read **company** and **region** directly from the original data set and then simply hold their values as we create the cases for the new data set.

```
input program.
numeric #oldreg #dsize1 to #dsize3(F3.0).
string #oldcomp (A5).
numeric region size (F3.0).
string company div (A5).
data list free / #oldcomp #oldreg #dsize1 to #dsize3.

compute company=#oldcomp.
compute region=#oldreg.
compute div='product'.
compute size=#dsize1.
end case.
compute company=#oldcomp.
compute region=#oldreg.
compute div='distrib'.
compute size=#dsize2.
end case.
compute company=#oldcomp.
compute region=#oldreg.
compute div='sales'.
compute size=#dsize3.
end case.
end input program.

begin data
JHF 1 25 10 10
SRU 2 100 35 20
IHOP 2 10 0 10
FFF 1 150 80 25
end data.
execute.
```

Figure 6.1: Sample SPSS program to convert data to a smaller unit of analysis.

The loop in lines 2-6 has three cycles, one for each of the company divisions. Line 2 defines the loop. Line 3 assigns the name of the division to the variable **div** while line 4 assigns the size of the division to the variable **size**. It is important that the order of the variables in the **Dname** vector corresponds to the order in the **Dsize** vector. Otherwise, we might end up with the departments being paired with the wrong sizes. The **end case** statement in line 5 outputs a new case at the end of each cycle of the loop. Line 6 closes the loop, and line 7 ends the input program. The last part of the program simply contains the data that was used in this example.

- The program in figure 6.3 reads in the employee data in the form shown in table 6.3 and converts it to the data set shown in table 6.4.

  The very first line starts the input program. Lines 2-3 initialize a set of variables that will be used to read data in from the original data set. These are defined as scratch variables so that they won't be included in the final data set. Lines 4-5 define the variables we will include in the new data set. Line 6 is used to read in the data records.

  The remainder of the input program is used to perform the conversion. Line 1 in the second set of commands (following the first blank line) tells SPSS to retain the values of the 10 variables holding subjects' responses as it reads in the records of the original data set. We will want to hold onto these values as we read in all of the records from the original data set until we create the case in the new data set. Normally these variables would be re-initialized each time a new record was read.

```
input program.
numeric #t #dsize1 to #dsize3 (F3.0).
vector Dsize=#dsize1 to #dsize3.
string #dname1 to #dname3(A5).
compute #dname1='product'.
compute #dname2='distrib'.
compute #dname3='sales'.
vector Dname=#dname1 to #dname3.
numeric region size (F3.0).
string company div (A5).
data list free / company region #dsize1 to #dsize3.

leave company region.
loop #t=1 to 3.
+    compute div=Dname(#t).
+    compute size=Dsize(#t).
+    end case.
end loop.
end input program.

begin data
JHF 1 25 10 10
SRU 2 100 35 20
IHOP 2 10 0 10
FFF 1 150 80 25
end data.
execute.
```

Figure 6.2: Revised SPSS program to convert data to a smaller unit of analysis.

A total of ten records will be read in from the original data set to create each case in the new data set. The ten **if** statements in lines 2-11 are used to assign the value of **#resp** to one of the variables **q01** to **q10**, depending on which question is currently being processed. Lines 12-15 complete the processing of each new observation. Line 12 is used to determine when all of the records pertaining to the same new observation have been read. The items in the original data set are sorted by question number, so we know that an observation is complete once the record for question number 10 has been read. At this point we want to do two things. Line 13 calculates the average of the responses to the 10 questions. Line 14 tells SPSS to add the case to the new data set.

At this point the data set now looks just like the example in table 6.4. The last part of the program simply contains the data that was used in this example. In your own programs you may decide to read your records from an external file, in which case you would omit these statements.

There are two characteristics your data set *must* have to restructure it using this type of program. First, the records of the original data set must all be ordered such that those that will be used to construct the same case in the new data set are all adjacent. Second, there must be a way to determine when the last record for a particular case has been read. In this example we used the value of the variable **#qnum**. If the same number of records are always used for each case you might just count them. Other times you may need to add a variable to your data set that specifically indicates which is the last record for each case. For example, you could have a variable that has the value of 0 for every record except the last one for each case, where it would have the value of 1.

- The program in figure 6.4 is basically the same thing as the program in figure 6.3 but uses more advanced programming techniques to simplify the code.

```
input program.
numeric #oldsub #qnum #resp (F3.0).
string #oldcond (A1).
numeric subject q01 to q10 (F3.0) ave(F3.1).
string cond (A1).
data list free / #oldsub #oldcond #qnum #resp.

leave q01 to q10.
if (#qnum=1) q01=#resp.
if (#qnum=2) q02=#resp.
if (#qnum=3) q03=#resp.
if (#qnum=4) q04=#resp.
if (#qnum=5) q05=#resp.
if (#qnum=6) q06=#resp.
if (#qnum=7) q07=#resp.
if (#qnum=8) q08=#resp.
if (#qnum=9) q09=#resp.
if (#qnum=10) q10=#resp.
do if #qnum=10.
+    compute subject = #oldsub.
+    compute cond = #oldcond.
+    compute ave=(q01+q02+q03+q04+q05+q06+q07+q08+q09+q10)/10.
+    end case.
end if.
end input program.

begin data
1 A 1 6
1 A 2 1
  ⋮
3 B 10 1
end data.
execute.
```

Figure 6.3: Sample SPSS program to convert data to a larger unit of analysis.

The very first line starts the input program. Line 2 initializes a set of variables that will be used to read data in from the original data set. Lines 3-5 define the variables we will include in the new data set, as well as a vector referring to the variables holding subjects' responses to the 10 questions. Line 6 is used to read in the data records. Notice that instead of reading the values of **subject** and **cond** in with one set of variables and then assigning them to new variables as we did in the prior example, this time we read the final values directly from the original data set.

The remainder of the input program is used to perform the conversion. Line 1 in the second set of commands (following the first blank line) tells SPSS to retain the values of the 10 variables holding subjects' responses as it reads in the records of the original data set. Line 2 is used to assign the current value of **#resp** to the appropriate variable in the **Q** vector. The variable **#qnum** is the question number of current record being read and so can tell us which of the 10 variables in the **Q** array should be assigned to the value of **#resp**. Line 3 sets the value of **#sum** equal to 0 each time we read in data from a new subject. Since the data are sorted by question number we know that we are working with a new subject every time **#qnum** has the value of 1. The variable **#sum** will be used to hold a running sum of the responses, to be used later in calculating the average response. Line 4 adds the current value to the previously calculated sum. For this to work, we need the value of **#sum**

```
input program.
numeric #qnum #resp #sum (F3.0).
numeric subject q01 to q10 (F3.0) ave(F3.1).
vector Q=q01 to q10.
string cond(A1).
data list free / subject cond #qnum #resp.

leave q01 to q10.
compute Q(#qnum)=#resp.
if (#qnum=1) #sum=0.
compute #sum=#sum+#resp.
do if (#qnum=10).
+    compute ave=#sum/10.
+    end case.
end if.
end input program.

begin data
1 A 1 6
1 A 2 1
  ⋮
3 A 10 1
end data.
execute.
```

Figure 6.4: Revised SPSS program to convert data to a larger unit of analysis.

to be retained as new records are read from the original data set. Notice, however, that we did not include **#sum** in the **leave** statement in line 1. This is because all scratch variables are automatically retained from record to record.

Lines 5-8 perform the final tasks once all of the records that will be used to create the current case have been read. With this data set we know that this is when **#qnum**=10. The first thing we do is divide the variable **#sum** by 10 to get the average response. The second thing we do is tell SPSS to create the case in the new data set.

At this point the data set now looks just like the data in table 6.4. The last part of the program simply contains the data that was used in this example.

6.5 Using SAS

- This section provides examples demonstrating how to restructure data sets in SAS. The first example shows how to move to a smaller unit of analysis using the employee data, while the second example shows how to move to a larger unit of analysis using the survey data. For each example we will go through two programs that can be used to restructure the data set, one performing the transformation without loops and vectors and one performing the transformation using loops and vectors. It is easier to understand what is going on with the first program but it is generally more efficient to use more advanced programming techniques, especially if your data set has a large number of variables.
- Before trying to restructure a data set you must be aware of four features of the SAS language that are not commonly used but which appear in the examples.
    - The **output** statement. This tells SAS to create a case in the data set using the current variable values. If you do not explicitly use an **output** statement SAS creates the case at the end of the

data step, causing it to create one case from each record it reads. By explicitly stating where SAS should create a case, however, you can instead have it create several cases from a single record or have it read several records before it creates a case.

- The **keep** statement. This command is used to limit the variables that will be included in the final version of your data set. Variables not listed in the **keep** statement will disappear once the data step ends. However, this does not affect your ability to use the dropped variables in computations. If you do not explicitly issue a **keep** statement SAS will keep all of your variables.
- The **retain** statement. By default SAS resets every variable to missing each time it reads in a new record. This command is used to tell SAS not to re-initialize a variable when it reads in new records. Variables listed in a **retain** statement only change if specifically reassigned by the programmer. If you want to retain the value of a variable across several records you will need to include it in a **retain** statement.
- Variables created using **set** and **by**. Let us assume that you had a data set **one** that was sorted by a variable **svar**. If you read **one** into a new data set **two** using the following code:

  ```
  data two;
    set one;
    by svar;
  ```

  SAS would automatically create two variables, **first.svar** and **last.svar** in data set two. The variable **first.svar** would have the value 1 for a given case if it was the *first* case in the data set to have that particular value of **svar** and would have a value of 0 otherwise. The variable **last.svar** would have the value 1 for a given case if it was the *last* case in the data set to have that particular value of **svar** and would have a value of 0 otherwise. Any time a data set is set **by** a variable SAS creates the **first.***varname* and **last.***varname* variables, where *varname* is the variable in the **by** statement. However, SAS will not let you set a data set by a particular variable unless the original data set is already sorted on that variable. The **first.***varname* and **last.***varname* variables are automatically removed from your data set once the data step ends.

- The program in figure 6.5 reads in the employee data in the form shown in table 6.1 and converts it to the data set shown in table 6.2.

  This program creates two data sets. The first data set **old** reads in the data and is structured so that the company is the unit of analysis, as presented in table 6.1. Inside the second data set **new** the data set is restructured so that the specific division is the unit of analysis, as presented in table 6.2.

  The very first line of data set **old** declares the name of the data set. Line 2 reads in the data, declaring **oldcomp** to be a string variable. Lines 3-7 contain the data used in this example. In your own programs you may decide to read your records from an external file, in which case you would omit these statements.

  The first line of data set **new** (following the first blank line) declares the name of the data set. Line 2 tells SAS that we will be reading in the records of data set **old** to create this data set. Line 3 tells SAS that the only variables we want to include in the final version of this data set are **company, region**, **div**, and **size**. Lines 4-8 are used to create the first new case containing information about the production division. Line 4 assigns the company name, line 5 assigns the region code, line 6 records the name of the division and line 7 records the size of the division. Line 8 tells SAS to create a new case using the current values of the variables. In a similar way, lines 9-13 are used to create a case containing the information from the distribution division and lines 14-18 are used to create a case containing the information from the sales division.

- The program in figure 6.6 is basically the same thing as the program in figure 6.5 but uses more advanced programming techniques to simplify the code.

  This program creates two data sets. The first data set **old** reads in the data and is structured so that the company is the unit of analysis, as presented in table 6.1. Inside the second data set **new** the data set is restructured so that the specific division is the unit of analysis, as presented in table 6.2.

```
data old;
input oldcomp$ oldreg prod dist sales;
cards;
JHF 1 25 10 10
SRU 2 100 35 20
IHOP 2 10 0 10
FFF 1 150 80 25
;

data new;
   set old;
keep company region div size;
company = oldcomp;
region = oldreg;
div='product';
size = prod;
output;
company = oldcomp;
region = oldreg;
div='distrib';
size = dist;
output;
company = oldcomp;
region = oldreg;
div='sales';
size = sales;
output;
run;
```

Figure 6.5: Sample SAS program to convert data to a smaller unit of analysis.

The very first line of data set **old** declares the name of the data set. Line 2 reads in the data, declaring **company** to be a string variable. Lines 3-7 contain the data used in this example. In your own programs you may decide to read your records from an external file, in which case you would omit these statements.

The first line of data set **new** (following the first blank line) declares the name of the data set. Line 2 tells SAS that we will be reading in the records of data set **old** to create this data set. The **keep** statement in line 3 declares which variables we want in the final version of data set **new**. Notice that we are keeping the variables **company** and **region** just as they were defined in data set **old**. Instead of reading these values in with one set of variables and then assigning them to new variables as we did in the last example, this time we take them directly from data set **old** and simply hold their values as we create the cases for the new data set.

Lines 4-7 define a set of variables to refer to the department names and an array to refer to those variables. Line 8 defines an array to refer to the variables that will be used to read in the division size.

The loop in lines 9-13 has three cycles, one for each of the company divisions. We need to do two things inside this loop. First, we need to create any variables that apply to the specific divisions. Line 10 assigns the name of the division to the variable **div** while line 11 assigns the size of the division to the variable **size**. It is important that the order of the variables in the **Dname** array corresponds to the order in the **Dsize** array. Otherwise, we might end up with the departments being paired with the wrong sizes. The second thing we need to do inside the loop is to tell SAS to create a case in the new data set. This is accomplished by the **output** statement in line 12. Line 13 simply ends the loop.

```
data old;
input company$ region prod dist sales;
cards;
JHF 1 25 10 10
SRU 2 100 35 20
IHOP 2 10 0 10
FFF 1 150 80 25
;

data new;
  set old;
keep company region div size;
dname1='product';
dname2='distrib';
dname3='sales';
array dname[*] $ dname1-dname3;
array dsize[*] prod dist sales;
do t=1 to 3;
  div=dname[t];
  size=dsize[t];
  output;
end;
run;
```

Figure 6.6: Revised SAS program to convert data to a smaller unit of analysis.

- The program in figure 6.7 reads in the employee data in the form shown in table 6.3 and converts it to the data set shown in table 6.4.

  Just like the previous examples, this program also creates two data sets. The first data set **old** reads in the data and is structured so that the question item is the unit of analysis, as presented in table 6.3. Inside the second data set **new** the data set is restructured so that the subject is the unit of analysis, as presented in table 6.4.

  The very first line of data set **old** declares the name of the data set. Line 2 reads in the data, declaring **oldcond** to be a string variable. The next lines, several of which are omitted to save space, contain the data used in this example. In your own programs you may decide to read your records from an external file, in which case you would omit these statements. The last two lines of this data set tell SAS to sort this data set by the variable **oldsub**.

  The first line of data set **new** (following the first blank line) declares the name of the data set. Line 2 tells SAS that we will be reading in the records of data set **old** to create this data set. Line 3 states that the records in data set **old** are sorted by the variable **oldsub** and that we want SAS to create the variables **first.oldsub** and **last.oldsub** (see the note on these variables at the beginning of this section). Line 4 declares which variables we want to keep final version of data set **new**. Line 5 tells SAS that we want to hold onto the values of the variables **q01-q10** as we read multiple records from the original data set. Normally these variables would be re-initialized each time a new record was read. However, we must read in multiple records before we output a new case so we must retain these values.

  A total of ten records will be read in from the original data set to create each case in the new data set. The ten **if** statements in lines 6-15 of data set **new** are used to assign the value of **resp** to one of the variables **q01** to **q10**, depending on which question is currently being processed. Lines 16-21 complete the processing of each new observation. Line 16 is used to determine when all of the records pertaining to the same subject have been read. At this point we want to do four things. Line 17 assigns the value of **subject** while line 18 assigns the value of **cond**. Line 19 calculates the average of the responses to the 10 questions. Line 20 tells SAS to add the case to the new data set.

```
data old;
input oldsub oldcond$ qnum resp;
cards;
1 A 1 6
1 A 2 1
  ⋮
3 A 10 1
;
proc sort;
  by oldsub;
run;

data new;
  set old;
  by oldsub;
keep subject cond q01-q10 ave;
retain q01-q10;
if qnum=1 then q01=resp;
if qnum=2 then q02=resp;
if qnum=3 then q03=resp;
if qnum=4 then q04=resp;
if qnum=5 then q05=resp;
if qnum=6 then q06=resp;
if qnum=7 then q07=resp;
if qnum=8 then q08=resp;
if qnum=9 then q09=resp;
if qnum=10 then q10=resp;
if (last.oldsub)=1 then do;
  subject = oldsub;
  cond = oldcond;
  ave = (q01+q02+q03+q04+q05+q06+q07+q08+q09+q10)/10;
  output;
end;
run;
```

Figure 6.7: Sample SAS program to convert data to a larger unit of analysis.

At this point the data set now looks just like the data in table 6.4.

- The program in figure 6.8 is basically the same thing as the program in figure 6.7 but uses more advanced programming techniques to simplify the code.

  The very first line of data set **old** declares the name of the data set. Line 2 reads in the data, declaring **cond** to be a string variable. The next lines, several of which are omitted to save space, contain the data used in this example. The last two lines of this data set tell SAS to sort this data set by the variable **subject**.

  The first line of data set **new** (following the first blank line) declares the name of the data set. Line 2 tells SAS that we will be reading in the records of data set **old** to create this data set. Line 3 states that the records in data set **old** are sorted by the variable **subject** and that we want SAS to create the variables **first.subject** and **last.subject**.

  Line 4 declares which variables we want to keep final version of data set **new**. Instead of reading the values of **subject** and **cond** in with one set of variables and then assigning them to new variables as we did in the prior example, this time we simply read them directly from the original data set. Line 5

```
data old;
input subject cond$ qnum resp;
cards;
1 A 1 6
1 A 2 1
  ⋮
3 A 10 1
;
proc sort;
  by subject;

data new;
  set old;
  by subject;
keep subject cond q01-q10 ave;
retain q01-q10 sum;
array Q[*] q01-q10;
Q[qnum]=resp;
if (first.subject=1) then sum=0;
sum=sum+resp;
if (last.subject=1) then do;
  ave = sum/10;
  output;
end;
run;
```

Figure 6.8: Revised SAS program to convert data to a larger unit of analysis.

tells SAS that we want to hold onto the values of the variables **q01-q10** and **sum** as we read multiple records from the original data set.

Line 6 defines an array referring to 10 variables that we will use to store subjects' responses to the questions. Line 7 is used to assign the current value of **resp** to the appropriate variable in the **Q** array. The variable **qnum** is the question number of current record being read and so can tell us which of the 10 variables in the **Q** array should be assigned the value of **resp**. Lines 8 and 9 are used to create a running sum of a subject's survey responses. This will be used later to calculate the average score. Line 8 checks to see if this is the first observation from a new subject. If it is, it resets the value of **sum** to 0. Line 9 adds the current value to the previously calculated sum.

Lines 10-13 perform the final tasks once all of the records that will be used to create the current case have been read. The first thing we do is divide the variable **sum** by 10 to get the average response. The second thing we do is tell SAS to create the case in the new data set.

At this point the data set now looks just like the data in table 6.4.

References

Newell, A., & Rosenbloom, P. S. (1981). Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 1-55).

SPSS Inc. (1997). *SPSS Base 7.5 Syntax Reference Guide*. Chigago, IL: SPSS Inc.

SAS Institute Inc. (1990). *SAS Language* (version 6). Cary, NC: SAS Institute Inc.

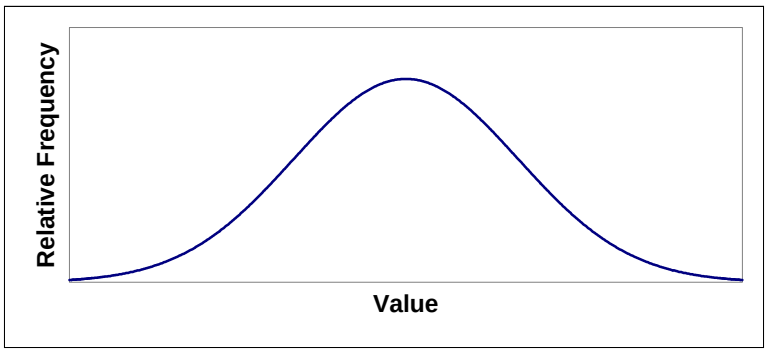

Figure 3.1: Example of a normal distribution

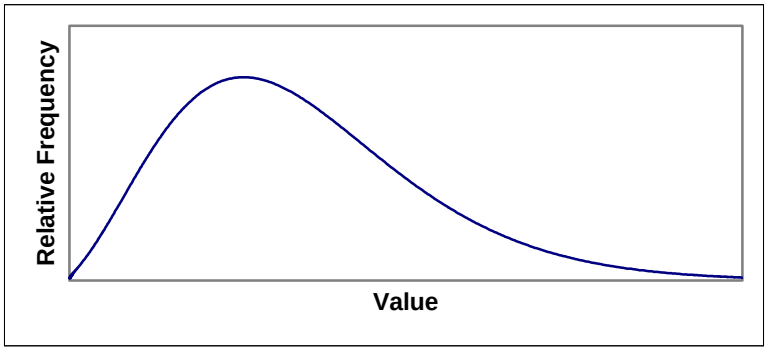

Figure 3.2: Example of data requiring a square-root transformation

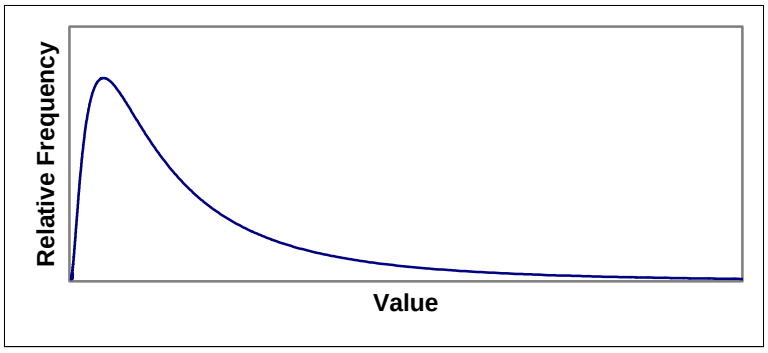

Figure 3.3: Example of data requiring a logarithmic transformation.

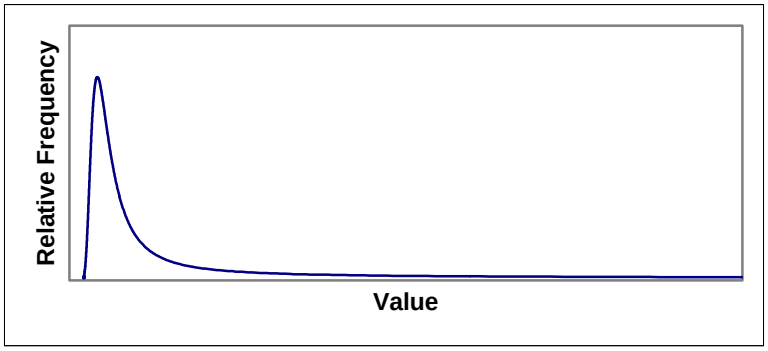

Figure 3.4: Example of data requiring an inverse transformation

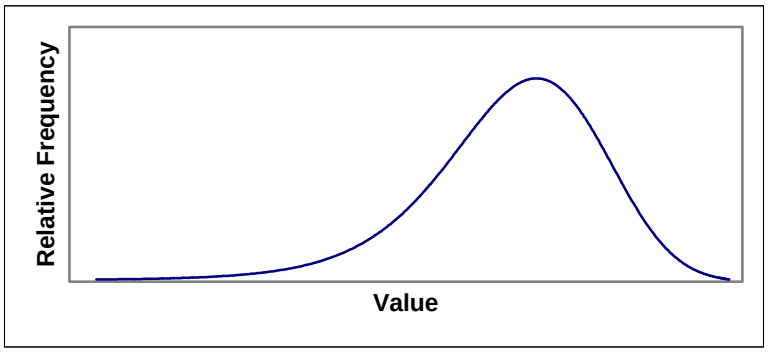

Figure 3.5: Example of data requiring a square transformation.

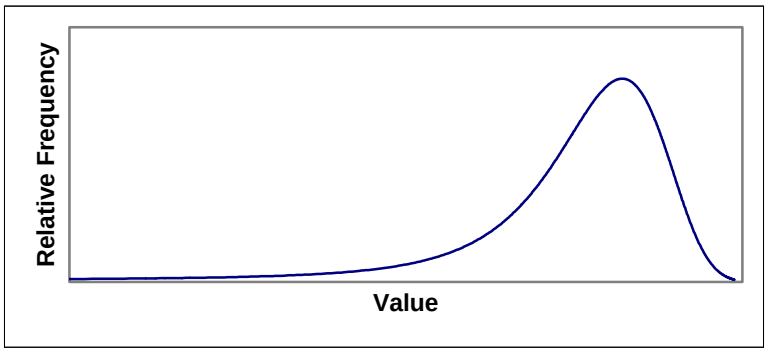

Figure 3.6: Example of data requiring an exponential transformation.