"Practice is the best of all instructors."

PUBLIUS SYRUS, CIRCA 42 B.C

"We all learned by doing, by experimenting (and often failing), and by asking questions."

JAY JACOB WIND

From *Bartlett's Familiar Quotations* 13th edition, by John Bartlett, copyright 1955 by Little Brown & Company. Public domain.
From the SAS L Listserv, March 15, 1994. Reprinted by permission of the author.

CHAPTER 2

Getting Your Data into SAS®

2.1 Methods for Getting Your Data into SAS

Data come in many different forms. Your data may be handwritten on a piece of paper, or typed into a raw data file on your computer. Perhaps your data are in a database file on your personal computer, or in a database management system (DBMS) on the mainframe computer at your office. Wherever your data reside, there is a way for SAS to use them. You may need to convert your data from one form to another, or SAS may be able to use your data in their current form. This section outlines several methods for getting your data into SAS. Most of these methods are covered in this book, but a few of the more advanced methods are merely mentioned so that you know they exist. We do not attempt to cover all methods available for getting your data into SAS, as new methods are continually being developed, and creative SAS users can always come up with clever methods that work for their own situations. But there should be at least one method explained in this book that will work for you.

Methods for getting your data into SAS can be put into four general categories:

- entering data directly into SAS data sets
- creating SAS data sets from raw data files
- converting other software's data files into SAS data sets
- reading other software's data files directly.

Naturally, the method you choose will depend on where your data are located, and what software tools are available to you.

Entering data directly into SAS data sets Sometimes the best method for getting your data into SAS is to enter the data directly into SAS data sets through your keyboard.

- The Viewtable window, discussed in section 2.2, is included with Base SAS software. Viewtable allows you to enter your data in a tabular format. You can define variables, or columns, and give them attributes such as name, length, and type (character or numeric).
- SAS Enterprise Guide software, a Windows only application, has a data entry window that is very similar to the Viewtable window. As with Viewtable, you can define variables and give them attributes.
- SAS/FSP software, short for Full Screen Product, allows you to design custom data entry screens. It also has the capability for detecting data entry errors as they happen. The SAS/FSP product is licensed separately from Base SAS software.

Creating SAS data sets from raw data files Much of this chapter is devoted to reading raw data files (also referred to as text, ASCII, sequential, or flat files). You can always read a raw data file since the DATA step is an integral part of Base SAS software. And, if your data are not already in a raw data file, chances are you can convert your data into a raw data file. There are two general methods for reading raw data files:

- The DATA step is so versatile that it can read almost any type of raw data file. This method is covered in this chapter starting with section 2.4.

- The Import Wizard, covered in section 2.3 and its cousin the IMPORT procedure, covered in section 2.16, are available for UNIX, OpenVMS, and Windows operating environments. These are simple methods for reading particular types of raw data files including comma-separated values (CSV) files, and other delimited files.

Converting other software's data files into SAS data sets Each software application has its own form for data files. While this is useful for software developers, it is troublesome for software users—especially when your data are in one application, but you need to analyze them with another. There are several options for converting data from applications into SAS data sets:

- The IMPORT procedure and the Import Wizard can be used to convert Microsoft Excel, Lotus, dBase, and Microsoft Access files into SAS data sets if you have SAS/ACCESS for PC File Formats software installed on your computer. This is covered in sections 2.3 and 2.17.
- If you don't have SAS/ACCESS software, then you can always create a raw data file from your application and read the raw data file with either the DATA step or the IMPORT procedure. Many applications can create CSV files, which are easily read using the Import Wizard or IMPORTprocedure (covered in sections 2.3 and 2.16) or the DATA step (covered in section 2.15).
- Dynamic Data Exchange (DDE), covered in section 2.18, is available only for those working in the Windows operating environment. To use DDE, you must have the other Windows application (Microsoft Excel for example) running on your computer at the same time as SAS. Then using DDE and the DATA step, you can convert data into SAS data sets.

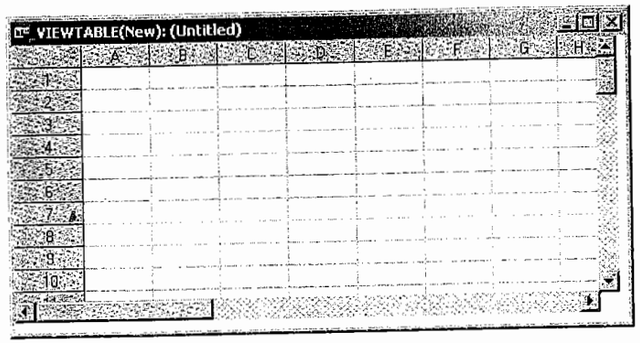
Reading other software's data files directly Under certain circumstances you may be able to read data without converting to a SAS data set. This method is particularly useful when you have many people updating data files, and you want to make sure that you are using the most current data.

- The SAS/ACCESS products allow you to read data without converting your data into SAS data sets. There are SAS/ACCESS products for most of the popular database management systems including ORACLE, DB2, INGRES, and SYBASE. This method of data access is not covered in this book.
- We already mentioned using SAS/ACCESS for PC Files Formats software to convert several PC file types to SAS data sets, but you can also use the Excel and Access engines to read these types of files directly without converting. See the SAS Help and Documentation for more information on these engines.
- There are also data engines that allow you to read data directly but are part of Base SAS software. The SPSS engine is covered in Appendix D. There are also engines for OSIRIS, old versions of SAS data sets, and SAS data sets in transport format. Check the SAS Help and Documentation for your operating environment for a complete list of available engines.

Given all these methods for getting your data into SAS, you are sure to find at least one method that will work for you—probably more.

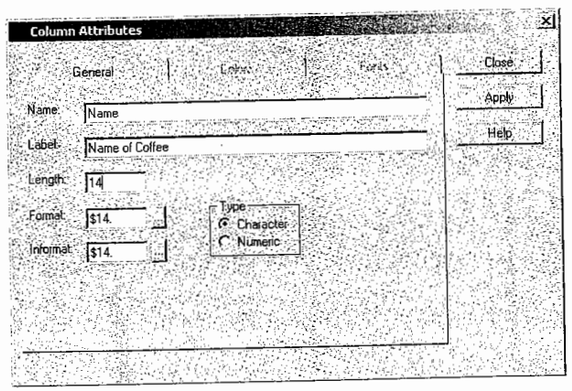
2.2 Entering Data with the Viewtable Window

The Viewtable window which is part of Base SAS software[1] is an easy way to create new data sets, or browse and edit existing data sets. True to its name, the Viewtable window displays tables (another name for data sets) in a tabular format. To open the Viewtable window, select `Table Editor` from the Tools menu. An empty Viewtable window will appear.



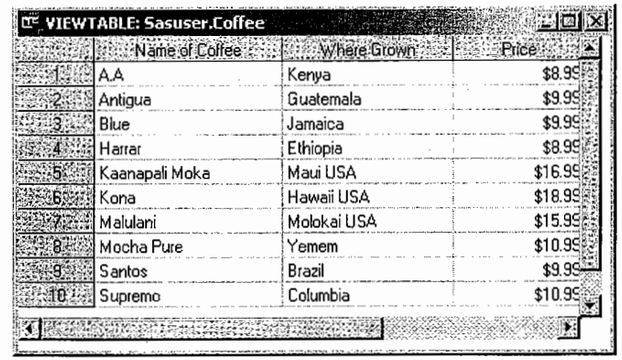
This table contains no data. Instead you see rows (or observations) labeled with numbers and columns (or variables) labeled with letters. You can start typing data into this default table, and SAS will automatically figure out if your columns are numeric or character. However, it's a good idea to tell SAS about your data so each column is set up the way you want. You do this with the Column Attributes window.

Column Attributes window The letters at the tops of columns are default variable names. By right-clicking on a letter, you can choose to open a Column Attributes window for that column. This window contains default values which you can replace with the values you desire. When you are satisfied with the values, click on `Apply`. To switch to a new column, click on that column in the Viewtable window. When you are finished changing column attributes click on `Close`.



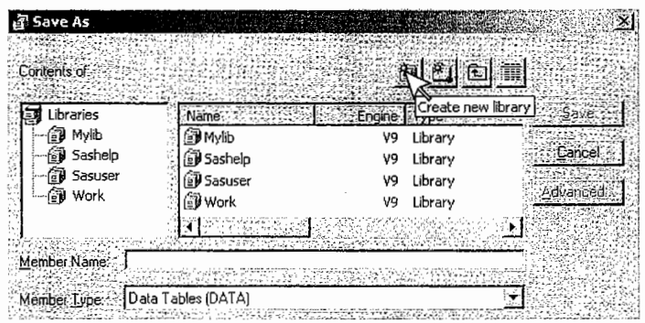
[1] If you are using a non-graphical monitor, then SAS uses FSVIEW to display your tables, so you also need SAS/FSP software which is licensed separately.

Entering data Once you have defined your columns you are ready to type in your data. To move the cursor, click on a field, or use tab and arrow keys. Here is a table with column attributes defined and data entered.



	Name of Coffee	Where Grown	Price
1	AA	Kenya	$8.95
2	Antigua	Guatemala	$9.95
3	Blue	Jamaica	$9.95
4	Harrar	Ethiopia	$8.95
5	Kaanapali Moka	Maui USA	$16.95
6	Kona	Hawaii USA	$18.95
7	Malulani	Molokai USA	$15.95
8	Mocha Pure	Yemem	$10.95
9	Santos	Brazil	$9.95
10	Supremo	Columbia	$10.95

Saving your table To save a table, select `Save As...` from the File menu. Select a library, and then specify the member name of your table. The libraries displayed correspond to locations (such as directories) on your computer. If you want to save your table in a different location, you can add another library by clicking on the `New Library` icon. Type in a name for the new library and its path. Then click on `OK`. Specify the member name by typing it in the `Member Name` field.



Opening an existing table To browse or edit an existing table, first select `Table Editor` from the Tools menu to open the Viewtable window. Then select `Open` from the File menu. Click on the library you want and then on the table name. If the table you want to open is not in any of the existing libraries, click on the `New Library` icon. Type in a name for the new library and its path. Then click on `OK`. To switch from browse mode (the default) to edit mode, select `Edit Mode` from the Edit menu. You can also open an existing table by navigating to it in the SAS Explorer window, and double clicking on it.

Other features The Viewtable window has many other features including sorting, printing, adding and deleting rows, and viewing multiple rows (the default, called Table View) or viewing one row at a time (called Form View). You can control these features using either menus or icons.

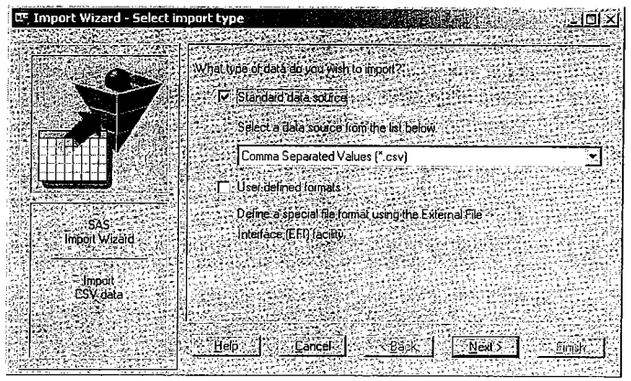
Using your table in a SAS program Tables that you create in Viewtable can be used in programs just as tables created in programs can be used in Viewtable. For example, if you saved your table in the SASUSER library and named it COFFEE, you could print it with this program:

```
PROC PRINT DATA=Sasuser.coffee;
RUN;
```

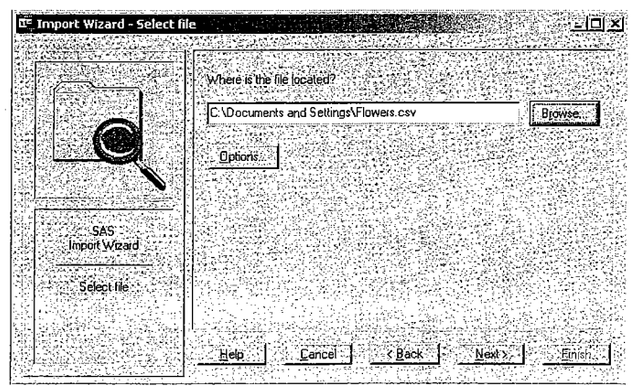
2.3 Reading Files with the Import Wizard

Using the Import Wizard[1], you can read a variety of data file types into SAS by simply answering a few questions. The Import Wizard will scan your file to determine variable types[2] and will, by default, use the first row of data for the variable names. The Import Wizard can read all types of delimited files including comma-separated values (CSV) files which are a common file type for moving data between applications. And, if you have SAS/ACCESS for PC File Formats software, then you can also read a number of popular PC file types[3].

Start the Import Wizard by choosing `Import Data...` from the File menu.



Select the type of file you are importing by choosing from the list of standard data sources such as comma-separated values (*.csv) files.

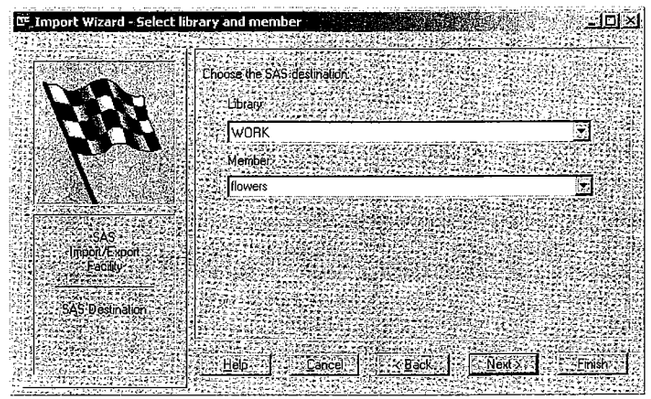


Now, specify the location of the file that you want to import. By default, SAS uses the first row in the file as the variable names for the SAS data set, and starts reading data in the second row. The `Options...` button takes you to another screen where you can change this default action.

[1] The Import Wizard is available in the Windows, UNIX, and OpenVMS operating environments.

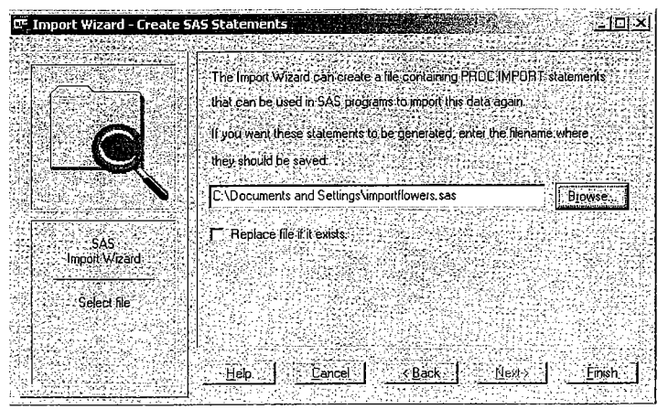
[2] By default the Import Wizard will scan the first 20 rows for delimited files and the first 8 rows for Microsoft Excel files. If you have all missing data in these rows, then the Import Wizard (and the IMPORT procedure) may not read the file correctly. See sections 2.16 and 2.17 for more information.

[3] Under the Windows operating environment, you can read Microsoft Excel, Microsoft Access, Lotus, and dBase files (if you are running Microsoft Windows 64-Bit Edition, then you cannot read Microsoft Access or Microsoft Excel 97, Excel 2000, or Excel 2002 files). Under the UNIX operating environment, you can read dBase files, and starting with SAS 9.1, UNIX users can also read Microsoft Excel and Microsoft Access files.



The next screen asks you to choose the SAS library and member name for the SAS data set that will be created. If you choose the WORK library, then the SAS data set will be deleted when you exit SAS. If you choose a different library, then the SAS data set will remain even after you exit SAS. There is no way to define a library from within the Import Wizard, so make sure your library is defined before entering the Import Wizard. You can define libraries using the New Library window discussed in section 1.11 (or using a LIBNAME statement as discussed in section 2.20). After choosing a library, enter a member name for the SAS data set.

In the last window, the Import Wizard gives you the option of saving the PROC IMPORT statements used for importing the file.



For some types of files, the Import Wizard asks additional questions. For example, if you are importing Microsoft Access files, then you will be asked for the database name and the table you want to import. You will also be given an opportunity to enter user ID and password information if applicable.

Using imported data in a SAS program Data that you import through the Import Wizard can be used in any SAS program. For example, if you saved your data in the WORK library and named it FLOWERS, you could print it with this program:

```
PROC PRINT DATA=WORK.flowers;
RUN;
```

Or, since WORK is the default library, you could also use:

```
PROC PRINT DATA=flowers;
RUN;
```

2.4 Telling SAS Where to Find Your Raw Data

If your data are in raw data files (also referred to as text, ASCII, sequential, or flat files), using the DATA step to read the data gives you the most flexibility. The first step toward reading raw data files is telling SAS where to find the raw data. Your raw data may be either internal to your SAS program, or in a separate file. Either way, you must tell SAS where to find your data.

A raw data file can be viewed using simple text editors or system commands. For PC users, raw data files will either have no program associated with them, or they will be associated with simple editors like Microsoft Notepad. In some operating environments, you can use commands to list the file, such as the `cat` or `more` commands in UNIX. Spreadsheet files are examples of data files that are not raw data. If you try using a text editor to look at a spreadsheet file, you will probably see lots of funny special characters you can't find on your keyboard. It may cause your computer to beep and chirp, making you wish you had that private office down the hall. It looks nothing like the nice neat rows and columns you see when you use your spreadsheet software to view the same file.

Internal raw data If you type raw data directly in your SAS program, then the data are internal to your program. You may want to do this when you have small amounts of data, or when you are testing a program with a small test data set. Use the DATALINES statement to indicate internal data. The DATALINES statement must be the last statement in the DATA step. All lines in the SAS program following the DATALINES statement are considered data until SAS encounters a semicolon. The semicolon can be on a line by itself or at the end of a SAS statement which follows the data lines. Any statements following the data are part of a new step. If you are old enough to remember punching computer cards, you might like to use the CARDS statement instead. The CARDS statement and the DATALINES statement are synonymous. The following SAS program illustrates the use of the DATALINES statement. (The DATA statement simply tells SAS to create a SAS data set named USPRESIDENTS, and the INPUT statement tells SAS how to read the data. The INPUT statement is discussed in sections 2.5 through 2.15.)

```
* Read internal data into SAS data set uspresidents;
DATA uspresidents;
   INPUT President $ Party $ Number;
   DATALINES;
Adams     F  2
Lincoln   R 16
Grant     R 18
Kennedy   D 35
   ;
RUN;
```

External raw data files Usually you will want to keep data in external files, separating the data from the program. This eliminates the chance that data will accidentally be altered when you are editing your SAS program. Use the INFILE statement to tell SAS the filename and path, if appropriate, of the external file containing the data. The INFILE statement follows the DATA statement and must precede the INPUT statement. After the INFILE keyword, the file path and name are enclosed in quotation marks. Examples from several operating environments follow:

Windows:	`INFILE 'c:\MyDir\President.dat';`
UNIX:	`INFILE '/home/mydir/president.dat';`
OpenVMS:	`INFILE '[username.mydir]president.dat';`
OS/390 or z/OS:	`INFILE 'MYID.PRESIDEN.DAT';`

Suppose the following data are in a file called President.dat in the directory MyRawData on the C drive (Windows):

```
Adams     F  2
Lincoln   R 16
Grant     R 18
Kennedy   D 35
```

The following program shows the use of the INFILE statement to read the external data file:

```
* Read data from external file into SAS data set;
DATA uspresidents;
   INFILE 'c:\MyRawData\President.dat';
   INPUT President $ Party $ Number;
RUN;
```

The SAS log Whenever you read data from an external file, SAS gives some very valuable information about the file in the SAS log. The following is an excerpt from the SAS log after running the previous program. Always check this information after you read a file as it could indicate problems. A simple comparison of the number of records read from the infile with the number of observations in the SAS data set can tell you a lot about whether SAS is reading your data correctly.

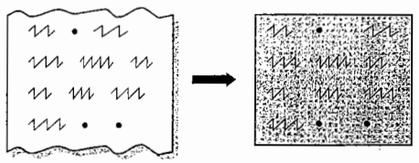
```
NOTE: The infile 'c:\MyRawData\President.dat' is:
      File Name=c:\MyRawData\President.dat,
      RECFM=V,LRECL=256
NOTE: 4 records were read from the infile 'c:\MyRawData\President.dat'.
      The minimum record length was 13.
      The maximum record length was 13.
NOTE: The data set WORK.USPRESIDENTS has 4 observations and 3 variables.
```

Long records In some operating environments, SAS assumes external files have a record length of 256 or less. (The record length is the number of characters, including spaces, in a data line.) If your data lines are long, and it looks like SAS is not reading all your data, then use the LRECL= option in the INFILE statement to specify a record length at least as long as the longest record in your data file.

```
INFILE 'c:\MyRawData\President.dat' LRECL=2000;
```

Check the SAS log to see that the maximum record length is as long as you think it should be.

2.5 Reading Raw Data Separated by Spaces



If the values in your raw data file are all separated by at least one space,[1] then using list input (also called free formatted input) to read the data may be appropriate. List input is an easy way to read raw data into SAS, but with ease come a few limitations. You must read all the data in a record—no skipping over unwanted values. Any missing data must be indicated with a period. Character data, if present, must be simple: no embedded spaces, and no values greater than eight characters in length.[2] If the data file contains dates or other values which need special treatment, then list input may not be appropriate. This may sound like a lot of restrictions, but a surprising number of data files can be read using list input.

The INPUT statement, which is part of the DATA step, tells SAS how to read your raw data. To write an INPUT statement using list input, simply list the variable names after the INPUT keyword in the order they appear in the data file. Generally, variable names must be 32 characters or fewer, start with a letter or an underscore, and contain only letters, underscores, or numerals. If the values are character (not numeric) then place a dollar sign ($) after the variable name. Leave at least one space between names, and remember to place a semicolon at the end of the statement. The following is an example of a simple list style INPUT statement.

```
INPUT Name $ Age Height;
```

This statement tells SAS to read three data values. The $ after Name indicates that it is a character variable, whereas the Age and Height variables are both numeric.

Example Your hometown has been overrun with toads this year. A local resident, having heard of frog jumping in California, had the idea of organizing a toad jump to cap off the annual town fair. For each contestant you have the toad's name, weight, and the jump distance from three separate attempts. If the toad is disqualified for any jump, then a period is used to indicate missing data. Here is what the data file ToadJump.dat looks like:

```
Lucky 2.3 1.9 . 3.0
Spot 4.6 2.5 3.1 .5
Tubs 7.1 . . 3.8
Hop 4.5 3.2 1.9 2.6
Noisy 3.8 1.3 1.8
1.5
Winner 5.7 . . .
```

This data file does not look very neat, but it does meet all the requirements for list input: the character data are eight characters or fewer and have no embedded spaces, all values are separated by at least one space, and missing data are indicated by a period. Notice that the data for Noisy have spilled over to the next data line. This is no problem since, by default, SAS will go to the next data line to read more data if there are more variables in the INPUT statement than there are values in the data line.

[1] SAS can read files with other delimiters such as commas or tabs using list input. See sections 2.14 and 2.15.

[2] It is possible to override this constraint using the LENGTH statement, discussed in section 10.13, which can change the length of character variables from the default of 8 to anything between 1 and 32,767.

Here is the SAS program that will read the data:

```
* Create a SAS data set named toads;
* Read the data file ToadJump.dat using list input;
DATA toads;
   INFILE 'c:\MyRawData\ToadJump.dat';
   INPUT ToadName $ Weight Jump1 Jump2 Jump3;
* Print the data to make sure the file was read correctly;
PROC PRINT DATA = toads;
   TITLE 'SAS Data Set Toads';
RUN;
```

The variables ToadName, Weight, Jump1, Jump2, and Jump3 are listed after the keyword INPUT in the same order as they appear in the file. A dollar sign ($) after ToadName indicates that it is a character variable; all the other variables are numeric. A PROC PRINT statement is used to print the data values after reading them to make sure they are correct. The PRINT procedure, in its simplest form, prints the values for all variables and all observations in a SAS data set. The TITLE statement after the PROC PRINT tells SAS to put the text enclosed in quotation marks on the top of each page of output. If you had no TITLE statement in your program, SAS would put the words "The SAS System" at the top of each page.

The output will look like this:

SAS Data Set Toads 1

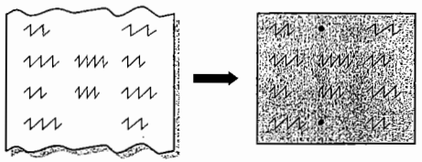
Obs	Toad Name	Weight	Jump1	Jump2	Jump3
1	Lucky	2.3	1.9	.	3.0
2	Spot	4.6	2.5	3.1	0.5
3	Tubs	7.1	.	.	3.8
4	Hop	4.5	3.2	1.9	2.6
5	Noisy	3.8	1.3	1.8	1.5
6	Winner	5.7	.	.	.

Because SAS had to go to a second data line to get the data for Noisy's final jump, the following note appears in the SAS log:

```
NOTE: SAS went to a new line when INPUT statement reached past the end of a line.
```

If you find this note in your SAS log when you didn't expect it, then you may have a problem. If so, look in section 10.4 which discusses this note in more detail.

2.6 Reading Raw Data Arranged in Columns



Some raw data files do not have spaces (or other delimiters) between all the values or periods for missing data—so the files can't be read using list input. But if each of the variable's values is always found in the same place in the data line, then you can use column input as long as all the values are character or standard numeric. Standard numeric data contain only numerals, decimal points, plus and minus signs, and E for scientific notation. Numbers with embedded commas or dates, for example, are not standard.

Column input has the following advantages over list input:

- spaces are not required between values
- missing values can be left blank
- character data can have embedded spaces
- you can skip unwanted variables.

Survey data are good candidates for column input. Most answers to survey questionnaires are single digits (0 through 9). If a space is entered between each value, then the file will be twice the size and require twice the typing of a file without spaces. Data files with street addresses, which often have embedded blanks, are also good candidates for column input. The street Martin Luther King Jr. Boulevard should be read as one variable not five, as it would be with list input. Data which can be read with column input can often also be read with formatted input or a combination of input styles (discussed in sections 2.7, 2.8, and 2.9).

With column input, the INPUT statement takes the following form: after the INPUT keyword, list the first variable's name. If the variable is character, leave a space; then place a $. After the $, or variable name if it is numeric, leave a space; then list the column or range of columns for that variable. The columns are positions of the characters or numbers in the data line and are not to be confused with columns like those you see in a spreadsheet. Repeat this for all the variables you want to read. The following shows a simple INPUT statement using column style:

```
INPUT Name $ 1-10 Age 11-13 Height 14-18;
```

The first variable, Name, is character and the data values are in columns 1 through 10. The Age and Height variables are both numeric, since they are not followed by a $, and data values for both of these variables are in the column ranges listed after their names.

Example The local minor league baseball team, the Walla Walla Sweets, is keeping records about concession sales. A ballpark favorite are the sweet onion rings which are sold at the concession stands and also by vendors in the bleachers. The ballpark owners have a feeling that in games with lots of hits and runs more onion rings are sold in the bleachers than at the concession stands. They think they should send more vendors out into the bleachers when the game heats up, but need more evidence to back up their feelings.

For each home game they have the following information: name of opposing team, number of onion ring sales at the concession stands and in the bleachers, the number of hits for each team, and the final score for each team. The following is a sample of the data file named Onions.dat. For your reference, a column ruler showing the column numbers has been placed above the data:

```
----+----1----+----2----+----3----+----4
Columbia Peaches      35  67  1 10  2  1
Plains Peanuts       210      2  5  0  2
Gilroy Garlics        151035 12 11  7  6
Sacramento Tomatoes  124  85 15  4  9  1
```

Notice that the data file has the following characteristics, all making it a prime candidate for column input. All the values line up in columns, the team names have embedded blanks, missing values are blank, and in one case there is not a space between data values. (Those Gilroy Garlics fans must really love onion rings.)

The following program shows how to read these data using column input:

```
* Create a SAS data set named sales;
* Read the data file Onions.dat using column input;
DATA sales;
   INFILE 'c:\MyRawData\Onions.dat';
   INPUT VisitingTeam $ 1-20 ConcessionSales 21-24 BleacherSales 25-28
         OurHits 29-31 TheirHits 32-34 OurRuns 35-37 TheirRuns 38-40;
* Print the data to make sure the file was read correctly;
PROC PRINT DATA = sales;
   TITLE 'SAS Data Set Sales';
RUN;
```

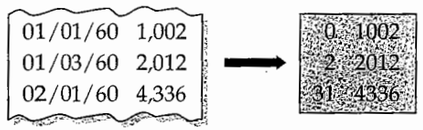
The variable VisitingTeam is character (indicated by a $) and reads the visiting team's name in columns 1 through 20. The variables ConcessionSales and BleacherSales read the concession and bleacher sales in columns 21 through 24 and 25 through 28, respectively. The number of hits for the home team, OurHits, and the visiting team, TheirHits, are read in columns 29 through 31 and 32 through 34, respectively. The number of runs for the home team, OurRuns, is read in columns 35 through 37, while the number of runs for the visiting team, TheirRuns, is in columns 38 through 40.

The output will look like this:

SAS Data Set Sales 1

Obs	VisitingTeam	Concession Sales	Bleacher Sales	Our Hits	Their Hits	Our Runs	Their Runs
1	Columbia Peaches	35	67	1	10	2	1
2	Plains Peanuts	210	.	2	5	0	2
3	Gilroy Garlics	15	1035	12	11	7	6
4	Sacramento Tomatoes	124	85	15	4	9	1

2.7 Reading Raw Data Not in Standard Format



Sometimes raw data are not straightforward numeric or character. For example, we humans easily read the number 1,000,001 as one million and one, but your trusty computer sees it as a character string. While the embedded commas make the number easier for us to interpret, they make the number impossible for the computer to recognize without some instructions. In SAS, informats are used to tell the computer how to interpret these types of data.

Informats are useful anytime you have non-standard data. (Standard numeric data contain only numerals, decimal points, minus signs, and E for scientific notation.) Numbers with embedded commas or dollar signs are examples of non-standard data. Other examples include data in hexadecimal or packed decimal formats. SAS has informats for reading these types of data as well.

Dates[1] are perhaps the most common non-standard data. Using date informats, SAS will convert conventional forms of dates like 10-31-2003 or 31OCT03 into a number, the number of days since January 1, 1960. This number is referred to as a SAS date value. (Why January 1, 1960? Who knows? Maybe 1960 was a good year for the SAS founders.) This turns out to be extremely useful when you want to do calculations with dates. For example, you can easily find the number of days between two dates by subtracting one from the other.

There are three general types of informats: character, numeric, and date. A table of selected SAS informats appears in section 2.8. The three types of informats have the following general forms:

Character	Numeric	Date
`$informatw.`	`informatw.d`	`informatw.`

The $ indicates character informats, INFORMAT is the name of the informat, *w* is the total width, and *d* is the number of decimal places (numeric informats only). The period is very important part of the informat name. Without a period, SAS may try to interpret the informat as a variable name, which by default, cannot contain any special characters except the underscore. Two informats do not have names: *$w.*, which reads standard character data, and *w.d*, which reads standard numeric data.

Use informats by placing the informat after the variable name in the INPUT statement; this is called formatted input. The following INPUT statement is an example of formatted input:

```
INPUT Name $10. Age 3. Height 5.1 BirthDate MMDDYY10.;
```

The columns read for each variable are determined by the starting point and the width of the informat. SAS always starts with the first column; so the data values for the first variable, Name, which has an informat of $10., are in columns 1 through 10. Now the starting point for the second variable is column 11, and SAS reads values for Age in columns 11 through 13. The values for the third variable, Height, are in columns 14 through 18. The five columns include the decimal place and the decimal point itself (150.3 for example). The values for the last variable, BirthDate, start in column 19 and are in a date form.

[1] Using dates in SAS is discussed in more detail in section 3.7.

Example This example illustrates the use of informats for reading data. The following data file, Pumpkin.dat, represents the results from a local pumpkin-carving contest. Each line includes the contestant's name, age, type (carved or decorated), the date the pumpkin was entered, and the scores from each of five judges.

```
Alicia Grossman  13 c 10-28-2003 7.8 6.5 7.2 8.0 7.9
Matthew Lee       9 D 10-30-2003 6.5 5.9 6.8 6.0 8.1
Elizabeth Garcia 10 C 10-29-2003 8.9 7.9 8.5 9.0 8.8
Lori Newcombe     6 D 10-30-2003 6.7 5.6 4.9 5.2 6.1
Jose Martinez     7 d 10-31-2003 8.9 9.510.0 9.7 9.0
Brian Williams   11 C 10-29-2003 7.8 8.4 8.5 7.9 8.0
```

The following program reads these data. Please note there are many ways to input these data, so if you imagined something else, that's OK.

```
* Create a SAS data set named contest;
* Read the file Pumpkin.dat using formatted input;
DATA contest;
   INFILE 'c:\MyRawData\Pumpkin.dat';
   INPUT Name $16. Age 3. +1 Type $1. +1 Date MMDDYY10.
         (Score1 Score2 Score3 Score4 Score5) (4.1);
* Print the data set to make sure the file was read correctly;
PROC PRINT DATA = contest;
   TITLE 'Pumpkin Carving Contest';
RUN;
```

The variable Name has an informat of $16., meaning that it is a character variable 16 columns wide. Variable Age has an informat of three, is numeric, three columns wide, and has no decimal places. The +1 skips over one column. Variable Type is character, and it is one column wide. Variable Date has an informat MMDDYY10. and reads dates in the form 10-31-2003 or 10/31/2003, each 10 columns wide. The remaining variables, Score1 through Score5, all require the same informat, 4.1. By putting the variables and the informat in separate sets of parentheses, you only have to list the informat once. Here are the results:

Pumpkin Carving Contest 1

Obs	Name	Age	Type	Date[2]	Score1	Score2	Score3	Score4	Score5
1	Alicia Grossman	13	c	16006	7.8	6.5	7.2	8.0	7.9
2	Matthew Lee	9	D	16008	6.5	5.9	6.8	6.0	8.1
3	Elizabeth Garcia	10	C	16007	8.9	7.9	8.5	9.0	8.8
4	Lori Newcombe	6	D	16008	6.7	5.6	4.9	5.2	6.1
5	Jose Martinez	7	d	16009	8.9	9.5	10.0	9.7	9.0
6	Brian Williams	11	C	16007	7.8	8.4	8.5	7.9	8.0

[2] Notice that these dates are printed as the number of days since January 1, 1960. Section 4.5 discusses how to format these values into readable dates.

2.8 Selected Informats

Definitions of commonly used informats[1] along with the width range and default width.

Informat	Definition	Width range	Default width
Character			
$CHAR*w*.	Reads character data—does not trim leading or trailing blanks	1-32,767	8 or length of variable
$HEX*w*.	Converts hexadecimal data to character data	1-32,767	2
$*w*.	Reads character data—trims leading blanks	1-32,767	none
Date, Time, and Datetime[2]			
DATE*w*.	Reads dates in form: *ddmmmyy* or *ddmmmyyyy*	7-32	7
DATETIME*w*.	Reads datetime values in the form: *ddmmmyy hh:mm:ss.ss*	13-40	18
DDMMYY*w*.	Reads dates in form: *ddmmyy* or *ddmmyyyy*	6-32	6
JULIAN*w*.	Reads Julian dates in form: *yyddd* or *yyyyddd*	5-32	5
MMDDYY*w*.	Reads dates in form: *mmddyy* or *mmddyyyy*	6-32	6
TIME*w*.	Reads time in form: *hh:mm:ss.ss* (hours:minutes:seconds—24-hour clock)	5-32	8
Numeric			
COMMA*w.d*	Removes embedded commas and $, converts left parentheses to minus sign	1-32	1
HEX*w*.	Converts hexadecimal to floating-point values if *w* is 16. Otherwise, converts to fixed-point.	1-16	8
IB*w.d*	Reads integer binary data	1-8	4
PD*w.d*	Reads packed decimal data	1-16	1
PERCENT*w*.	Converts percentages to numbers	1-32	6
w.d	Reads standard numeric data	1-32	none

[1] Check the SAS Help and Documentation for a complete list of informats.

[2] SAS date values are the number of days since January 1, 1960. Time values are the number of seconds past midnight, and datetime values are the number of seconds past midnight January 1, 1960.

Examples using the selected informats.

Informat	Input data	INPUT statement	Results
Character			
$CHAR*w*.	`my cat` `  my cat`	`INPUT Animal $CHAR10.;`	`my cat` `  my cat`
$HEX*w*.	`6C6C`	`INPUT Name $HEX4.;`	`11 (ASCII)or` `%% (EBCDIC)`[3]
$*w*.	`my cat` `  my cat`	`INPUT Animal $10.;`	`my cat` `my cat`
Date, Time, and Datetime			
DATE*w*.	`1jan1961` `1 jan 61`	`INPUT Day DATE10.;`	`366` `366`
DATETIME*w*.	`1jan1960 10:30:15` `1jan1961,10:30:15`	`INPUT Dt DATETIME18.;`	`37815` `31660215`
DDMMYY*w*.	`01.01.61` `02/01/61`	`INPUT Day DDMMYY8.;`	`366` `367`
JULIAN*w*.	`61001` `1961001`	`INPUT Day JULIAN7.;`	`366` `366`
MMDDYY*w*.	`01-01-61` `01/01/61`	`INPUT Day MMDDYY8.;`	`366` `366`
TIME*w*.	`10:30` `10:30:15`	`INPUT Time TIME8.;`	`37800` `37815`
Numeric			
COMMA*w.d*	`$1,000,001` `(1,234)`	`INPUT Income COMMA10.;`	`1000001` `-1234`
HEX*w*.	`F0F3`	`INPUT Value HEX4.;`	`61683`
IB*w.d*	[4]	`INPUT Value IB4.;`	`255`
PD*w.d*	[4]	`INPUT Value PD4.;`	`255`
PERCENT*w*.	`5%` `(20%)`	`INPUT Value PERCENT5.;`	`0.05` `-0.2`
w.d	`1234` `-12.3`	`INPUT Value 5.1;`	`123.4` `-12.3`

[3] The EBCDIC character set is used on most IBM mainframe computers, while the ASCII character set is used on most other computers. So, depending on the computer you are using, you will get one or the other.

[4] These values cannot be printed.

2.9 Mixing Input Styles

Each of the three major input styles has its own advantages. List style is the easiest; column style is a bit more work; and formatted style is the hardest of the three. However, column and formatted styles do not require spaces (or other delimiters) between variables and can read embedded blanks. Formatted style can read special data such as dates. Sometimes you use one style, sometimes another, and sometimes the easiest way is to use a combination of styles. SAS is so flexible that you can mix and match any of the input styles for your own convenience.

Example The following raw data contain information about U.S. national parks: name, state (or states as the case may be), year established, and size in acres:

```
Yellowstone             ID/MT/WY 1872    4,065,493
Everglades              FL 1934          1,398,800
Yosemite                CA 1864            760,917
Great Smoky Mountains NC/TN 1926           520,269
Wolf Trap Farm          VA 1966                130
```

You could write the INPUT statement for these data in many ways—that is the point of this section. The following program shows one way to do it:

```
* Create a SAS data set named nationalparks;
* Read a data file Park.dat mixing input styles;
DATA nationalparks;
   INFILE 'c:\MyRawData\Park.dat';
   INPUT ParkName $ 1-22 State $ Year @40 Acreage COMMA9.;
PROC PRINT DATA = nationalparks;
   TITLE 'Selected National Parks';
RUN;
```

Notice that the variable ParkName is read with column style input, State and Year are read with list style input, and Acreage is read with formatted style input. The output looks like this:

Selected National Parks 1

Obs	ParkName	State	Year	Acreage
1	Yellowstone	ID/MT/WY	1872	4065493
2	Everglades	FL	1934	1398800
3	Yosemite	CA	1864	760917
4	Great Smoky Mountains	NC/TN	1926	520269
5	Wolf Trap Farm	VA	1966	130

Sometimes programmers run into problems when they mix input styles. When SAS reads a line of raw data it uses a pointer to mark its place, but each style of input uses the pointer a little differently. With list style input, SAS automatically scans to the next non-blank field and starts reading. With column style input, SAS starts reading in the exact column you specify. But with formatted input, SAS just starts reading—wherever the pointer is, that is where SAS reads. Sometimes you need to move the pointer explicitly, and you can do that by using the column pointer, *@n*, where *n* is the number of the column SAS should move to.

In the preceding program, the column pointer @40 tells SAS to move to column 40 before reading the value for Acreage. If you removed the column pointer from the INPUT statement, as shown in the following statement, then SAS would start reading Acreage right after Year:

```
INPUT ParkName $ 1-22 State $ Year Acreage COMMA9.;
```

The resulting output would look like this:

Selected National Parks 1

Obs	ParkName	State	Year	Acreage
1	Yellowstone	ID/MT/WY	1872	4065
2	Everglades	FL	1934	.
3	Yosemite	CA	1864	.
4	Great Smoky Mountains	NC/TN	1926	5
5	Wolf Trap Farm	VA	1966	.

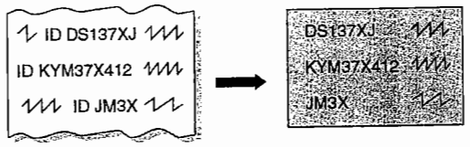
Because Acreage was read with formatted input, SAS started reading right where the pointer was. Here is the data file with a column ruler for counting columns at the top and asterisks marking the place where SAS started reading the values of Acreage:

```
----+----1----+----2----+----3----+----4----+----5
Yellowstone             ID/MT/WY 1872 *  4,065,493
Everglades              FL 1934 *        1,398,800
Yosemite                CA 1864 *          760,917
Great Smoky Mountains NC/TN 1926 *         520,269
Wolf Trap Farm          VA 1966 *              130
```

The COMMA9. informat told SAS to read nine columns, and SAS did that even when those columns were completely blank.

The column pointer, *@n*, has other uses too and can be used anytime you want SAS to skip backwards or forwards within a data line. You could use it, for example, to skip over unneeded data, or to read a variable twice using different informats.

2.10 Reading Messy Raw Data



Sometimes you need to read data that just don't line up in nice columns or have pre-dictable lengths. When you have these types of messy files, ordinary list, column, or formatted input simply aren't enough. You need more tools in your bag: tools like the @'character' column pointer and the colon modifier.

The @*'character'* column pointer In section 2.9 we showed you how you can use the @ column pointer to move to a particular column before reading data. However, sometimes you don't know the starting column of the data, but you do know that it always comes after a particular character or word. For these types of situations, you can use the @*'character'* column pointer. For example, suppose you have a data file that has information about dog ownership. Nothing in the file lines up, but you know that the breed of the dog always follows the word Breed:. You could read the dog's breed using the following INPUT statement:

```
INPUT @'Breed:' DogBreed $;
```

The colon modifier The above INPUT statement will work just fine as long as the dog's breed name is 8 characters or less (the default length for a character variable). So if the dog is a Shepherd you're fine, but if the dog is a Rottweiler, all you will get is Rottweil. If you assign the variable an informat in the INPUT statement such as $20. to tell SAS that the variable's field is 20 characters, then SAS will read for 20 columns whether or not there is a space in those columns.[1] So the DogBreed variable may include unwanted characters which appear after the dog's breed on the data line. If you only want SAS to read until it encounters a space[2], then you can use a colon modifier on the informat. To use a colon modifier, simply put a colon (:) before the informat (e.g. :$20. instead of $20.).

For example, given this line of raw data,

```
My dog Sam  Breed: Rottweiler  Vet Bills: $478
```

the following table shows the results you would get using different INPUT statements:

Statements	Value of variable DogBreed
`INPUT @'Breed:' DogBreed $;`	`Rottweil`
`INPUT @'Breed:' DogBreed $20.;`	`Rottweiler Vet Bill`
`INPUT @'Breed:' DogBreed :$20.;`	`Rottweiler`

[1] It is also possible to define a variable's length in a LENGTH or INFORMAT statement instead of in an INPUT statement. When a variable's length is defined before the INPUT statement, then SAS will read until it encounters a space or reaches the length of the variable—the same behavior as using the colon modifier. The INFORMAT statement is covered in section 2.21 and the LENGTH statement is covered in section 10.13.

[2] A space is the default delimiter. This method works for files with other delimiters as well. See sections 2.15 and 2.16 for more information on reading delimited data.

Example Web logs are a good example of messy data. The following data lines are part of a web log for a dog care business website. The data lines start with the IP address of the computer accessing the web page followed by other information including the date the file was accessed and the file name.

```
130.192.70.235 - - [08/Jun/2001:23:51:32 -0700] "GET /rover.jpg HTTP/1.1" 200 66820
128.32.236.8 - - [08/Jun/2001:23:51:40 -0700] "GET /grooming.html HTTP/1.0" 200 8471
128.32.236.8 - - [08/Jun/2001:23:51:40 -0700] "GET /Icons/brush.gif HTTP/1.0" 200 89
128.32.236.8 - - [08/Jun/2001:23:51:40 -0700] "GET /H_poodle.gif HTTP/1.0" 200 1852
118.171.121.37 - - [08/Jun/2001:23:56:46 -0700] "GET /bath.gif HTTP/1.0" 200 14079
128.123.121.37 - - [09/Jun/2001:00:57:49 -0700] "GET /lobo.gif HTTP/1.0" 200 18312
128.123.121.37 - - [09/Jun/2001:00:57:49 -0700] "GET /statemnt.htm HTTP/1.0" 200 238
128.75.226.8 - - [09/Jun/2001:01:59:40 -0700] "GET /Icons/leash.gif HTTP/1.0" 200 98
```

We are interested in the date the files were accessed and the filename. You can see that because the IP address is not always the same number of characters, the date does not line up in the same column all the time. Also, not only does the filename not line up in columns, but the length of the filename is highly variable. Here is a SAS program that can read this file:

```
DATA weblogs;
  INFILE 'c:\MyWebLogs\dogweblogs.txt';
  INPUT @'[' AccessDate DATE11. @'GET' File :$20.;
PROC PRINT DATA = weblogs;
  TITLE 'Dog Care Web Logs';
RUN;
```

This INPUT statement uses @'[' to position the column pointer to read the date, then uses @'GET' to position the column pointer to read the filename. Because the filename is more than 8 characters, but not always the same number of characters, an informat with a colon modifier :$20. is used to read the filename.

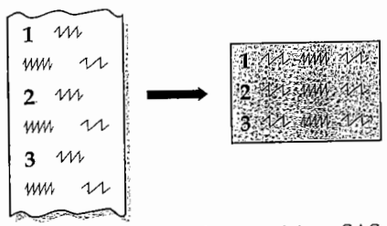
Here are the results of this program:

Dog Care Web Logs 1

Obs	AccessDate[3]	File
1	15134	/rover.jpg
2	15134	/grooming.html
3	15134	/Icons/brush.gif
4	15134	/H_poodle.gif
5	15134	/bath.gif
6	15134	/lobo.gif
7	15135	/statemnt.htm
8	15135	/Icons/leash.gif

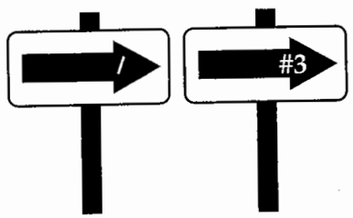
[3] Notice that these dates are printed as the number of days since January 1, 1960. Section 4.5 discusses how to format these values into readable dates.

2.11 Reading Multiple Lines of Raw Data per Observation



In a typical raw data file each line of data represents one observation, but sometimes the data for each observation are spread over more than one line. Since SAS will automatically go to the next line if it runs out of data before it has read all the variables in an INPUT statement, you could just let SAS take care of figuring out when to go to a new line. But if you know that your data file has multiple lines of raw data per observation, it is better for you to explicitly tell SAS when to go to the next line than to make SAS figure it out. That way you won't get that suspicious SAS-went-to-a-new-line note in your log. To tell SAS when to skip to a new line, you simply add line pointers to your INPUT statement.

The line pointers, slash (/) and pound-*n* (#*n*), are like road signs telling SAS, "Go this way." To read more than one line of raw data for a single observation, you simply insert a slash into your INPUT statement when you want to skip to the next line of raw data. The #*n* line pointer performs the same action except that you specify the line number. The *n* in #*n* stands for the number of the line of raw data for that observation; so #2 means to go to the second line for that observation, and #4 means go to the fourth line. You can even go backwards using the #*n* line pointer, reading from line 4 and then from line 3, for example. The slash is simpler, but #*n* is more flexible.



Example A colleague is trying to plan his next summer vacation, but he wants to go someplace where the weather is just right. He obtains data from a meteorology database. Unfortunately, he has not quite figured out how to export from this database and makes a rather odd file.

The file contains information about temperatures for the month of July for Alaska, Florida, and North Carolina. (If your colleague chooses the last state, maybe he can visit SAS headquarters.) The first line contains the city and state, the second line lists the normal high temperature and normal low (in degrees Fahrenheit), and the third line contains the record high and low:

```
Nome AK
55 44
88 29
Miami FL
90 75
97 65
Raleigh NC
88 68
105 50
```


The following program reads the weather data from a file named Temperature.dat:

```
* Create a SAS data set named highlow;
* Read the data file using line pointers;
DATA highlow;
   INFILE 'c:\MyRawData\Temperature.dat';
   INPUT City $ 1-5 State $
         NormalHigh NormalLow
         /#3 RecordHigh RecordLow;
PROC PRINT DATA = highlow;
   TITLE 'High and Low Temperatures for July';
RUN;
```

The INPUT statement reads the values for City and State from the first line of data. Then the slash tells SAS to move to column 1 of the next line of data before reading NormalHigh and NormalLow. Likewise, the #3 tells SAS to move to column 1 of the third line of data for that observation before reading RecordHigh and RecordLow. As usual, there is more than one way to write this INPUT statement. You could replace the slash with #2 or replace #3 with a slash.

This note appears in the log:

```
NOTE: 9 records were read from the infile 'c:\MyRawData\Temperature.dat'.
      The minimum record length was 5.
      The maximum record length was 10.

NOTE: The data set WORK.HIGHLOW has 3 observations and 6 variables.
```

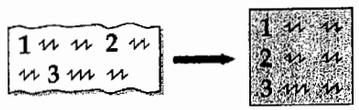
Notice that while nine records were read from the infile, the SAS data set contains just three observations. Usually this would set off alarms in your mind, but here it confirms that indeed three data lines were read for every observation just as planned. You should always check your log, particularly when using line pointers.

The output looks like this:

High and Low Temperatures for July 1

Obs	City	State	Normal High	Normal Low	Record High	Record Low
1	Nome	AK	55	44	88	29
2	Miami	FL	90	75	97	65
3	Raleigh	NC	88	68	105	50

2.12 Reading Multiple Observations per Line of Raw Data



There ought to be a Murphy's law of data: whatever form data can take, it will. Normally SAS assumes that each line of raw data represents no more than one observation. When you have multiple observations per line of raw data, you can use double trailing at signs (@@) at the end of your INPUT statement. This line-hold specifier is like a stop sign telling SAS, "Stop, hold that line of raw data." SAS will hold that line of data, continuing to read observations until it either runs out of data or reaches an INPUT statement that does not end with a double trailing @.

Example Suppose you have a colleague who is planning a vacation and has obtained a file containing data about rainfall (in inches) for the three cities he is considering. The file contains the name of each city, the state, average rainfall for the month of July, and average number of days with measurable precipitation in July. The raw data look like this:

```
Nome AK 2.5 15 Miami FL 6.75
18 Raleigh NC . 12
```

Notice that in this data file the first line stops in the middle of the second observation. The following program reads these data from a file named Precipitation.dat and uses an @@ so SAS does not automatically go to a new line of raw data for each observation:

```
* Input more than one observation from each record;
DATA rainfall;
   INFILE 'c:\MyRawData\Precipitation.dat';
   INPUT City $ State $ NormalRain MeanDaysRain @@;
PROC PRINT DATA = rainfall;
   TITLE 'Normal Total Precipitation and';
   TITLE2 'Mean Days with Precipitation for July';
RUN;
```


These notes will appear in the log:

```
NOTE: 2 records were read from the infile 'c:\MyRawData\Precipitation.dat'
      The minimum record length was 18.
      The maximum record length was 28.

NOTE: SAS went to a new line when INPUT statement reached past the
      end of a line.

NOTE: The data set WORK.RAINFALL has 3 observations and
      4 variables.
```

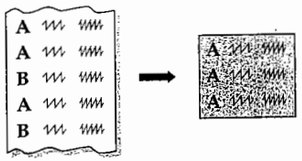
While only two records were read from the raw data file, the RAINFALL data set contains three observations. The log also includes a note saying SAS went to a new line when the INPUT statement reached past the end of a line. This means that SAS came to the end of a line in the middle of an observation and continued reading with the next line of raw data. Normally these messages would indicate a problem, but in this case they are exactly what you want.

The output looks like this:

Normal Total Precipitation and 1
Mean Days with Precipitation for July

Obs	City	State	Normal Rain	Mean DaysRain
1	Nome	AK	2.50	15
2	Miami	FL	6.75	18
3	Raleigh	NC	.	12

2.13 Reading Part of a Raw Data File



At some time you may find that you need to read a small fraction of the records in a large data file. For example, you might be reading U.S. census data and want only female heads-of-household who have incomes above $225,000 and live in Walla Walla, Washington. You could read all the records in the data file and then throw out the unneeded ones, but that would waste time.

Luckily, you don't have to read all the data before you tell SAS whether to keep an observation. Instead, you can read just enough variables to decide whether to keep the current observation, then end the INPUT statement with an at sign (@), called a trailing at. This tells SAS to hold that line of raw data. While the trailing @ holds that line, you can test the observation with an IF statement to see if it's one you want to keep. If it is, then you can read data for the remaining variables with a second INPUT statement. Without the trailing @, SAS would automatically start reading the next line of raw data with each INPUT statement.

The trailing @ is similar to the column pointer, *@n*, introduced in section 2.9. By specifying a number after the @ sign, you tell SAS to move to a particular column. By using an @ without specifying a column, it is as if you are telling SAS, "Stay tuned for more information. Don't touch that dial!" SAS will hold that line of data until it reaches either the end of the DATA step, or an INPUT statement that does not end with a trailing @.

Example You want to read part of a raw data file containing local traffic data for freeways and surface streets. The data include information about the type of street, name of street, the average number of vehicles per hour traveling that street during the morning, and the average number of vehicles per hour for the evening. Here are the raw data:

```
freeway 408                              3684 3459
surface Martin Luther King Jr. Blvd.     1590 1234
surface Broadway                         1259 1290
surface Rodeo Dr.                        1890 2067
freeway 608                              4583 3860
freeway 808                              2386 2518
surface Lake Shore Dr.                   1590 1234
surface Pennsylvania Ave.                1259 1290
```

Suppose you want to see only the freeway data at this point so you read the raw data file, Traffic.dat, with this program:

```
* Use a trailing @, then delete surface streets;
DATA freeways;
   INFILE 'c:\MyRawData\Traffic.dat';
   INPUT Type $ @;
   IF Type = 'surface' THEN DELETE;
   INPUT Name $ 9-38 AMTraffic PMTraffic;
PROC PRINT DATA = freeways;
   TITLE 'Traffic for Freeways';
RUN;
```


Notice that there are two INPUT statements. The first reads the character variable Type and then ends with an @. The trailing @ holds each line of data while the IF statement tests it. The second INPUT statement reads Name (in columns 9 through 38), AMTraffic, and PMTraffic. If an observation has a value of surface for the variable Type, then the second INPUT statement never executes. Instead SAS returns to the beginning of the DATA step to process the next observation and does not add the unwanted observation to the FREEWAYS data set. (Do not pass go, do not collect $200.)

When you run this program, the log will contain the following two notes, one saying that eight records were read from the input file and another saying that the new data set contains only three observations:

```
NOTE: 8 records were read from the infile 'c:\MyRawData\Traffic.dat'.
      The minimum record length was 47.
      The maximum record length was 47.

NOTE: The data set WORK.FREEWAYS has 3 observations and 4 variables.
```

The other five observations had a value of surface for the variable Type and were deleted by the IF statement. The output looks like this:

Traffic for Freeways 1

Obs	Type	Name	AMTraffic	PMTraffic
1	freeway	408	3684	3459
2	freeway	608	4583	3860
3	freeway	808	2386	2518

Trailing @ versus double trailing @ The double trailing @, discussed in the previous section, is similar to the trailing @. Both are line-hold specifiers; the difference is how long they hold a line of data for input. The trailing @ holds a line of data for subsequent INPUT statements, but releases that line of data when SAS returns to the top of the DATA step to begin building the next observation. The double trailing @ holds a line of data for subsequent INPUT statements even when SAS starts building a new observation. In both cases, the line of data is released if SAS reaches a subsequent INPUT statement that does not contain a line-hold specifier.

2.14 Controlling Input with Options in the INFILE Statement

So far in this chapter, we have seen ways to use the INPUT statement to read many different types of raw data. When reading raw data files, SAS makes certain assumptions. For example, SAS starts reading with the first data line and, if SAS runs out of data on a line, it automatically goes to the next line to read values for the rest of the variables. Most of the time this is OK, but some data files can't be read using the default assumptions. The options in the INFILE statement change the way SAS reads raw data files. The following options are useful for reading particular types of data files. Place these options after the filename in the INFILE statement.

FIRSTOBS= The FIRSTOBS= option tells SAS at what line to begin reading data. This is useful if you have a data file that contains descriptive text or header information at the beginning, and you want to skip over these lines to begin reading the data. The following data file, for example, has a description of the data in the first two lines:

```
Ice-cream sales data for the summer
Flavor     Location   Boxes sold
Chocolate  213        123
Vanilla    213        512
Chocolate  415        242
```

The following program uses the FIRSTOBS= option to tell SAS to start reading data on the third line of the file:

```
DATA icecream;
   INFILE 'c:\MyRawData\Sales.dat' FIRSTOBS = 3;
   INPUT Flavor $ 1-9 Location BoxesSold;
RUN;
```

OBS= The OBS= option can be used anytime you want to read only a part of your data file. It tells SAS to stop reading when it gets to that line in the raw data file. Note that it does not necessarily correspond to the number of observations. If, for example, you are reading two raw data lines for each observation, then an OBS=100 would read 100 data lines, and the resulting SAS data set would have 50 observations. The OBS= option can be used with the FIRSTOBS= option to read lines from the middle of the file. For example, suppose the ice-cream sales data had a remark at the end of the file that was not part of the data.

```
Ice-cream sales data for the summer
Flavor     Location   Boxes sold
Chocolate  213        123
Vanilla    213        512
Chocolate  415        242
Data verified by Blake White
```

With FIRSTOBS=3 and OBS=5, SAS will start reading this file on the third data line and stop reading after the fifth data line.

```
DATA icecream;
   INFILE 'c:\MyRawData\Sales.dat' FIRSTOBS = 3 OBS=5;
   INPUT Flavor $ 1-9 Location BoxesSold;
RUN;
```


MISSOVER By default, SAS will go to the next data line to read more data if SAS has reached the end of the data line and there are still more variables in the INPUT statement that have not been assigned values. The MISSOVER option tells SAS that if it runs out of data, don't go to the next data line. Instead, assign missing values to any remaining variables. The following data file illustrates where this option may be useful. This file contains test scores for a self-paced course. Since not all students complete all the tests, some have more scores than others.

```
Nguyen    89 76 91 82
Ramos     67 72 80 76 86
Robbins   76 65 79
```

The following program reads the data for the five test scores, assigning missing values to tests not completed:

```
DATA class102;
   INFILE 'c:\MyRawData\Scores.dat' MISSOVER;
   INPUT Name $ Test1 Test2 Test3 Test4 Test5;
RUN;
```

TRUNCOVER You need the TRUNCOVER option when you are reading data using column or formatted input and some data lines are shorter than others. If a variable's field extends past the end of the data line, then, by default, SAS will go to the next line to start reading the variable's value. This option tells SAS to read data for the variable until it reaches the end of the data line, or the last column specified in the format or column range, whichever comes first. The next file contains addresses and must be read using column or formatted input because the street names have embedded blanks. Note that the data lines are all different lengths:

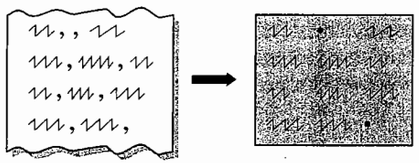
```
John Garcia     114  Maple Ave.
Sylvia Chung   1302  Washington Drive
Martha Newton    45  S.E. 14th St.
```

This program uses column input to read the address file. Because some of the addresses stop before the end of the variable Street's field (columns 22 through 37), you need the TRUNCOVER option. Without the TRUNCOVER option, SAS would try to go to the next line to read the data for Street on the first and third records.

```
DATA homeaddress;
   INFILE 'c:\MyRawData\Address.dat' TRUNCOVER;
   INPUT Name $ 1-15 Number 16-19 Street $ 22-37;
RUN;
```

TRUNCOVER is similar to MISSOVER. Both will assign missing values to variables if the data line ends before the variable's field starts. But when the data line ends in the middle of a variable field, TRUNCOVER will take as much as is there, whereas MISSOVER will assign the variable a missing value.

2.15 Reading Delimited Files with the DATA Step



Delimited files are raw data files that have a special character separating data values. Many programs can save data as delimited files, often with commas or tab characters for delimiters. SAS gives you two options for the INFILE statement that make it easy to read delimited data files: the DLM= option and the DSD option.

The DLM= option If you read your data using list input, the DATA step expects your file to have spaces between your data values. The DELIMITER=, or DLM=, option in the INFILE statement allows you to read data files with other delimiters. The comma and tab characters are common delimiters found in data files, but you could read data files with any delimiter character by just enclosing the delimiter character in quotation marks after the DLM= option (i.e., DLM='&').

Example The following file is comma-delimited where students' names are followed by the number of books they read for each week in a summer reading program:

```
Grace,3,1,5,2,6
Martin,1,2,4,1,3
Scott,9,10,4,8,6
```

This program uses list input to read the books data file specifying the comma as the delimiter:

```
DATA reading;
   INFILE 'c:\MyRawData\Books.dat' DLM = ',';
   INPUT Name $ Week1 Week2 Week3 Week4 Week5;
RUN;
```

If the same data had tab characters between values instead of commas, then you could use the following program to read the file. This program uses the DLM='09'X option. In ASCII, 09 is the hexadecimal equivalent of a tab character, and the notation '09'X means a hexadecimal 09. If your computer uses EBCDIC (IBM mainframes) instead of ASCII, then use DLM='05'X.

```
DATA reading;
   INFILE 'c:\MyRawData\Books.txt' DLM = '09'X;
   INPUT Name $ Week1 Week2 Week3 Week4 Week5;
RUN;
```

By default, SAS interprets two or more delimiters in a row as a single delimiter. If your file has missing values, and two delimiters in a row indicate a missing value, then you will also need the DSD option in the INFILE statement.

The DSD option The DSD (Delimiter-Sensitive Data) option for the INFILE statement does three things for you. First, it ignores delimiters in data values enclosed in quotation marks. Second, it does not read quotation marks as part of the data value. Third, it treats two delimiters in a row as a missing value. The DSD option assumes that the delimiter is a comma. If your delimiter is not a comma then you can use the DLM= option with the DSD option to specify the delimiter. For example, to read a tab-delimited ASCII file with missing values indicated by two consecutive tab characters use

```
INFILE 'file-specification' DLM='09'X DSD;
```


CSV files Comma-separated values files, or CSV files, are a common type of file that can be read with the DSD option. Many programs, such as Microsoft Excel, can save data in CSV format. These files have commas for delimiters and consecutive commas for missing values; if there are commas in any of the data values, then those values are enclosed in quotation marks.

Example The following example illustrates how to read a CSV file using the DSD option. Jerry's Coffee Shop employs local bands to attract customers. Jerry keeps records of the number of customers for each band, for each night they play in his shop. The band's name is followed by the date and the number of customers present at 8 p.m., 9 p.m., 10 p.m., and 11 p.m.

```
Lupine Lights,12/3/2003,45,63,70,
Awesome Octaves,12/15/2003,17,28,44,12
"Stop, Drop, and Rock-N-Roll",1/5/2004,34,62,77,91
The Silveyville Jazz Quartet,1/18/2004,38,30,42,43
Catalina Converts,1/31/2004,56,,65,34
```

Notice that one group's name has embedded commas, and is enclosed in quotation marks. Also, the last group has a missing data point for the 9 p.m. hour as indicated by two consecutive commas. Use the DSD option in the INFILE statement to read this data file. It is also prudent, when using the DSD option, to add the MISSOVER option if there is any chance that you have missing data at the end of your data lines (as in the first line of this data file). The MISSOVER option tells SAS that if it runs out of data, don't go to the next data line to continue reading. Here is the program that will read this data file:

```
DATA music;
   INFILE 'c:\MyRawData\Bands.csv' DLM = ',' DSD MISSOVER;
   INPUT BandName :$30. GigDate :MMDDYY10. EightPM NinePM TenPM ElevenPM;
PROC PRINT DATA = music;
   TITLE 'Customers at Each Gig';
RUN;
```

Notice that for BandName and GigDate we use colon modified informats. The colon modifier tells SAS to read for the length of the informat (30 for BandName and 10 for GigDate), or until it encounters a delimiter, whichever comes first. Because the names of the bands are longer than the default length of 8 characters, we use the :$30. informat for BandName to read up to 30 characters.

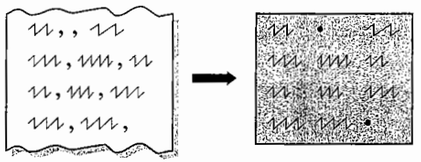
Here are the results of the PROC PRINT:

Customers at Each Gig 1

Obs	BandName	Gig Date[1]	Eight PM	Nine PM	Ten PM	Eleven PM
1	Lupine Lights	16042	45	63	70	.
2	Awesome Octaves	16054	17	28	44	12
3	Stop, Drop, and Rock-N-Roll	16075	34	62	77	91
4	The Silveyville Jazz Quartet	16088	38	30	42	43
5	Catalina Converts	16101	56	.	65	34

[1] Notice that these dates are printed as the number of days since January 1, 1960. Section 4.5 discusses how to format these values into readable dates.

2.16 Reading Delimited Files with the IMPORT Procedure



We suspect that by now you have realized that with SAS there is usually more than one way to accomplish the same result. In section 2.15 we showed you how to read delimited data files using the DATA step; now we are going to show you how to read delimited files a different way: using the IMPORT procedure.[1]

There are a few things that PROC IMPORT does for you that make it easy to read certain types of data files. PROC IMPORT will scan your data file and automatically determine the variable types (character or numeric), will assign proper lengths to the character variables, and can recognize some date formats.[2] PROC IMPORT will treat two consecutive delimiters in your data file as a missing value, will read values enclosed by quotation marks, and assign missing values to variables when it runs out of data on a line. Also, if you want, you can use the first line in your data file for the variable names. The IMPORT procedure actually writes a DATA step for you, and after you submit your program, you can look in the Log window to see the DATA step it produced.

The simplest form of the IMPORT procedure is

```
PROC IMPORT DATAFILE = 'filename' OUT = data-set;
```

where the file you want to read follows the DATAFILE= option, and the name of the SAS data set you want to create follows the OUT= option. SAS will determine the file type by the extension of the file as shown in the following table.

Type of File	Extension	DBMS Identifier
Comma-delimited	.csv	CSV
Tab-delimited	.txt	TAB
Delimiters other than commas or tabs		DLM

If your file does not have the proper extension, or your file is of type DLM, then you must use the DBMS= option in the PROC IMPORT statement. Use the REPLACE option if you already have a SAS data set with the name you specified in the OUT= option, and you want to overwrite it. Here is the general form of PROC IMPORT with both the REPLACE and the DBMS options:

```
PROC IMPORT DATAFILE = 'filename' OUT = data-set
     DBMS = identifier REPLACE;
```

The IMPORT procedure will, by default, get variable names from the first line in your data file. If you do not want this, then add the GETNAMES=NO statement after the PROC IMPORT statement. PROC IMPORT will assign the variables the names VAR1, VAR2, VAR3, and so on. Also if your data file is type DLM, PROC IMPORT assumes that the delimiter is a space. If you have a

[1] The IMPORT procedure is available on UNIX, OpenVMS, and Windows only.

[2] By default the IMPORT procedure will scan the first 20 rows of delimited files. If you have all missing data in these rows, then the Import Wizard may not read the file correctly. To change the number of rows, enter the REGEDIT command on the SAS command line, then select Find from the Edit menu and search for "GuessingRows" (make sure Value Names is checked). Then double click on "GuessingRows" to change the value.

different delimiter, then specify it in the DELIMITER= statement. The following shows both these statements:

```
PROC IMPORT DATAFILE = 'filename' OUT = data-set
     DBMS = DLM REPLACE;
   GETNAMES = NO;
   DELIMITER = 'delimiter-character';
RUN;
```

Example The following example uses data about Jerry's Coffee Shop where Jerry employs local bands to attract customers. Jerry keeps records of the number of customers present throughout the evening for each band. The data are the band name, followed by the gig date, and the number of customers present at 8 p.m., 9 p.m., 10 p.m., and 11 p.m. Notice that one of the bands, "Stop, Drop, and Rock-N-Roll," has commas in the name of the band. When a data value contains the delimiter, then the value must be enclosed in quotation marks.

```
Band Name,Gig Date,Eight PM,Nine PM,Ten PM,Eleven PM
Lupine Lights,12/3/2003,45,63,70,
Awesome Octaves,12/15/2003,17,28,44,12
"Stop, Drop, and Rock-N-Roll",1/5/2004,34,62,77,91
The Silveyville Jazz Quartet,1/18/2004,38,30,42,43
Catalina Converts,1/31/2004,56,,65,34
```

Here is the program that will read this data file and print out the SAS data set after importing:

```
PROC IMPORT DATAFILE = 'c:\MyRawData\Bands.csv' OUT = music REPLACE;
PROC PRINT DATA = music;
   TITLE 'Customers at Each Gig';
RUN;
```

Here are the results of the PROC PRINT. Notice that GigDate is a readable date. This is because IMPORT automatically assigns informats and formats to some forms of dates. (See section 4.5 for a discussion of formats.)

```
                        Customers at Each Gig                        1

Obs    Band_Name                              Gig_Date        Eight_PM

 1     Lupine Lights                          12/03/2003            45
 2     Awesome Octaves                        12/15/2003            17
 3     Stop, Drop, and Rock-N-Roll            01/05/2004            34
 4     The Silveyville Jazz Quartet           01/18/2004            38
 5     Catalina Converts                      01/31/2004            56

Obs        Nine_PM         Ten_PM       Eleven_PM

 1             63              70               .
 2             28              44              12
 3             62              77              91
 4             30              42              43
 5              .              65              34
```

2.17 Reading PC Files with the IMPORT Procedure

If you have SAS/ACCESS for PC File Formats software, then you can use the IMPORT procedure to import several types of PC files. The IMPORT procedure will scan your file to determine variable types[1] and will, by default, use the first row of data for the variable names. In the Windows operating environment, you can import Microsoft Excel, Lotus, dBase, and Microsoft Access files[2]. On UNIX systems you can import dBase files, and starting with SAS 9.1, UNIX users can also read Microsoft Excel and Microsoft Access files. An alternative method of reading PC files in the Windows operating environment which does not require SAS/ACCESS is Dynamic Data Exchange (DDE) which is covered in section 2.18.

Microsoft Excel, Lotus, and dBase files Here is the general form of the IMPORT procedure for reading PC files:

```
PROC IMPORT DATAFILE = 'filename' OUT = data-set
    DBMS = identifier  REPLACE;
```

where *filename* is the file you want to read and *data-set* is the name of the SAS data set you want to create. The REPLACE option tells SAS to replace the SAS data set named in the OUT= option if it already exists. If your data file has the proper extension, as shown in the following table, then you may not need the DBMS= option. Of course, it never hurts to specify the DBMS.

Type of File	Extension	DBMS Identifier
Microsoft Excel	.xls	EXCEL[3]
		EXCEL5
		EXCEL4
Lotus	.wk4	WK4
	.wk3	WK3
	.wk1	WK1
dBase	.dbf	DBF

If you are reading a Microsoft Excel file, and you have more than one sheet in your file, then you can specify which sheet to read using the following statement:

```
SHEET=name-of-sheet;
```

By default, the IMPORT procedure will take the variable names from the first row of the spreadsheet (Microsoft Excel and Lotus only). If you do not want this, then you can add the following statement to the procedure and SAS will name the variables F1, F2, and so on.

```
GETNAMES=NO;
```

[1] By default the IMPORT procedure will scan the first 8 rows for Microsoft Excel files. If you have all missing data in these rows, then the IMPORT procedure may not read the file correctly. To change the number of rows, submit the REGEDIT command from the Windows command line (from the Start menu, select Run). Select Find from the Edit menu, and search for "TypeGuessRows". Double-click on TypeGuessRows to change the value.

[2] If you are running Microsoft Windows 64-Bit Edition, then you cannot read Microsoft Access or Microsoft Excel 97, Excel 2000, or Excel 2002 files.

[3] The DBMS identifiers, EXCEL, EXCEL2002, EXCEL2000, and EXCEL97, are interchangeable since all these types of Microsoft Excel files have the same format. If you want to read a Microsoft Excel4 or Microsoft Excel5 file, then you must specify the DBMS identifier.

Microsoft Access Files If you want to read Microsoft Access files, then instead of using the DATAFILE= option, you need a DATABASE= and a DATATABLE=option as follows[4]:

```
PROC IMPORT DATABASE = 'database-path' DATATABLE = 'table-name'
     OUT = data-set DBMS = identifier  REPLACE;
```

The following are the DBMS identifiers for Microsoft Access:

Type of File	Extension	DBMS Identifier
Microsoft Access	.mdb	ACCESS[5]
		ACCESS97

Example Suppose you have the following Microsoft Excel spreadsheet which contains data about onion ring sales for the local minor league baseball team games. The visiting team name is followed by the sales in the concession stands and in the bleachers, then the number of hits and runs for each team.

	A	B	C	D	E	F	G
1	Visiting Team	C Sales	B Sales	Our Hits	Their Hits	Our Runs	Their Runs
2	Columbia Peaches	35	67	1	10	2	1
3	Plains Peanuts	210		2	5	0	2
4	Gilroy Garlics	15	1035	12	11	7	6
5	Sacramento Tomatoes	124	85	15	4	9	1

The following program reads the Microsoft Excel file using the IMPORT procedure. Microsoft Excel does not need to be running to use the IMPORT procedure.

```
* Read an Excel spreadsheet using PROC IMPORT;
PROC IMPORT DATAFILE = 'c:\MyExcelFiles\Onions.xls' OUT = sales;
PROC PRINT DATA = sales;
   TITLE 'SAS Data Set Read From Excel File';
RUN;
```

Here are the results:

SAS Data Set Read From Excel File 1

Obs	Visiting_Team	C_Sales	B_Sales	Our_Hits	Their_ Hits	Our_Runs	Their_ Runs
1	Columbia Peaches	35	67	1	10	2	1
2	Plains Peanuts	210	.	2	5	0	2
3	Gilroy Garlics	15	1035	12	11	7	6
4	Sacramento Tomatoes	124	85	15	4	9	1

[4] Additional options may be needed if your Microsoft Access database is password protected. See the SAS Help and Documentation for more information.

[5] The DBMS identifiers ACCESS, ACCESS2000, and ACCESS2002 are interchangeable since all these types of Microsoft Access files have the same format. If you want to read a Microsoft Access 97 file, then you must specify the DBMS identifier.

2.18 Reading PC Files with DDE

One method for reading PC files is Dynamic Data Exchange (DDE). DDE has some advantages and disadvantages when compared to other methods of reading PC files. DDE can only be used in the Windows operating environment, and the application (such as Microsoft Excel) must be running on the computer while SAS is accessing the file. But DDE does allow you to directly access data stored in PC files and it does not require any additional SAS products to be licensed. There are several ways to access data through DDE. We will present three methods:

- copying data to the clipboard
- specifying the DDE triplet
- starting the PC application from SAS, then reading the data.

Copying data to the clipboard If you don't want to be bothered with determining the DDE triplet, then you can just copy the rows and columns that you want to read into SAS onto the clipboard. Then you use the CLIPBOARD keyword in the DDE FILENAME statement. For example, suppose you have the following spreadsheet open in Microsoft Excel.

	Visiting Team	C Sales	B Sales	Our Hits	Their Hits	Our Runs	Their Runs
	Columbia Peaches	35	67	1	10	2	1
	Plains Peanuts	210		2	5	0	2
	Gilroy Garlics	15	1035	12	11	7	6
	Sacramento Tomatoes	124	85	15	4	9	1

Copy the rows and columns you want to read into SAS (A2 to G5) onto the clipboard, then, without closing Microsoft Excel, submit the following SAS program:

```
* Read an Excel spreadsheet using DDE;
FILENAME baseball DDE 'CLIPBOARD';
DATA sales;
  INFILE baseball NOTAB DLM='09'x DSD MISSOVER;
  LENGTH VisitingTeam $ 20;
  INPUT VisitingTeam CSales BSales OurHits TheirHits OurRuns TheirRuns;
RUN;
```

The FILENAME statement defines a fileref (BASEBALL) as type DDE and specifies that you want to read the contents of the clipboard. By default, DDE assumes there are spaces between your data values. So, if you have embedded spaces in your data, then you will need the NOTAB and the DLM='09'x options in the INFILE statement. These two options tell SAS to put a tab character (NOTAB) between values and define the tab character as the delimiter (DLM='09'x). In addition, if you have missing values in your data, you will want to add the DSD and MISSOVER options to the INFILE statement. The DSD option treats two delimiters in a row as missing data and the MISSOVER options tells SAS not to go to the next data line to continue reading data if it runs out of data on the current line.

Specifying the DDE triplet The clipboard method is easy, but it requires you to take the extra step of copying the data to the clipboard before you run the SAS program. If you know the DDE triplet for the data you want to read, then you can just specify the triplet in the FILENAME statement. However figuring out what the DDE triplet is, can be a little tricky. Each application has its own way of specifying a DDE triplet. In general, the DDE triplet takes on the following form:

application | *topic* ! *item*

Specific information about DDE triplets can be found in the documentation for the PC application. However, there is a way to find out the DDE triplet for your data from within SAS. First, copy the data you want onto the clipboard, then toggle to your SAS session. From the Solutions menu, select `Accessories`. Then select `DDE Triplet`. A window will appear that will give the DDE triplet for the data that you copied to the clipboard. For example, the DDE triplet for the spreadsheet shown is

```
Excel|C:\MyFiles\[BaseBall.xls]sheet1!R2C1:R5C7
```

So, to read the same data by specifying the DDE triplet, you would use the following FILENAME statement and the rest of the program is the same:

```
FILENAME baseball DDE 'Excel|C:\MyFiles\[BaseBall.xls]sheet1!R2C1:R5C7';
```

Starting the application from SAS With both the previous examples, the PC application must first be running before you can run the SAS program. Since this is sometimes inconvenient, you may want to start the application from within SAS, then read the data using DDE. You need to add two things to your SAS program to do this. First, you need the NOXWAIT and NOXSYNC systems options, then you need to use the X statement. Here is an example program:

```
* Read an Excel spreadsheet using DDE;
OPTIONS NOXSYNC NOXWAIT;
X '"C:\MyFiles\BaseBall.xls"';
FILENAME baseball DDE 'Excel|C:\MyFiles\[BaseBall.xls]sheet1!R2C1:R5C7';
DATA sales;
  INFILE baseball NOTAB DLM='09'x DSD MISSOVER;
  LENGTH VisitingTeam $ 20;
  INPUT VisitingTeam CSales BSales OurHits TheirHits OurRuns TheirRuns;
RUN;
```

The NOXWAIT and the NOXSYNC options tell SAS not to wait for input from the user, and to return control back to SAS after executing the command. The X statement simply tells Windows to execute the program or open the file that follows in quotation marks. Notice that there are two sets of quotation marks around the filename. If you have embedded spaces in the path for your filename, then you need to enclose the filename in two sets of quotation marks. Note that using this method, you must specify the DDE triplet in the FILENAME statement.

2.19 Temporary versus Permanent SAS Data Sets

SAS data sets are available in two varieties: temporary and permanent. A temporary SAS data set is one that exists only during the current job or session and is automatically erased by SAS when you finish. If a SAS data set is permanent, that doesn't mean that it lasts for eternity, just that it remains when the job or session is finished.

Each type of data set has its own advantages. Sometimes you want to keep a data set for later use, and sometimes you don't. In this book, most of our examples use temporary data sets because we don't want to clutter up your disks. But, in general, if you use a data set more than once, it is more efficient to save it as a permanent SAS data set than to create a new temporary SAS data set every time you want to use the data.

SAS data set names All SAS data sets have a two-level name such as WORK.BIKESALES, with the two levels separated by a period. The first level of a SAS data set name, WORK in this case, is called its libref (short for SAS data library reference). A libref is like an arrow pointing to a particular location. Sometimes a libref refers to a physical location, such as a floppy disk or CD, while other times it refers to a logical location such as a directory or folder. The second level, BIKESALES, is the member name that uniquely identifies the data set within the library.

Both the libref and member name follow the standard rules for valid SAS names. They must start with a letter or underscore and contain only letters, numerals, or underscores. However, librefs cannot be longer than 8 characters while member names can be up to 32 characters long.

You never explicitly tell SAS to make a data set temporary or permanent, it is just implied by the name you give the data set when you create it. Most data sets are created in DATA steps, but PROC steps can also create data sets. If you specify a two-level name (and the libref is something other than WORK) then your data set will be permanent. If you specify just one level of the data set name (as we have in most of the examples in this book), then your data set will be temporary. SAS will use your one-level name as the member name and automatically append the libref WORK. By definition, any SAS data set with a libref of WORK is a temporary data set and will be erased by SAS at the end of your job or session. Here are some sample DATA statements and the characteristics of the data sets they create:

DATA statement	Libref	Member name	Type
DATA ironman;	WORK	ironman	temporary
DATA WORK.tourdefrance;	WORK	tourdefrance	temporary
DATA Mylib.doublecentury;	Mylib	doublecentury	permanent

Temporary SAS data sets The following program creates and then prints a temporary SAS data set named DISTANCE:

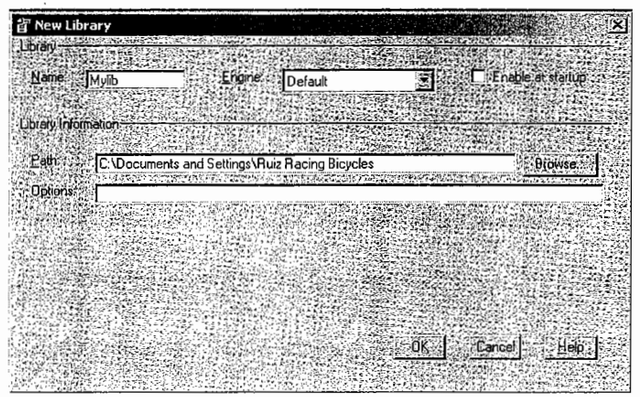
```
DATA distance;
   Miles = 26.22;
   Kilometers = 1.61 * Miles;
PROC PRINT DATA = distance;
RUN;
```


Notice that the libref WORK does not appear in the DATA statement. Because the data set has just a one-level name, SAS assigns the default library, WORK, and uses DISTANCE as the member name within that library. The log contains this note with the complete, two-level name:

```
NOTE: The data set WORK.DISTANCE has 1 observations and 2 variables.
```

Permanent SAS data sets Before you can use a libref, you need to define it. You can define libraries using the New Library window in SAS Explorer (covered in section 1.11). You can also use the LIBNAME statement (covered in section 2.20) or you can let SAS define the libref for you using direct referencing (covered in section 2.21)[1].

The Mylib library, defined in the New Library window shown in the figure, points to the 'Ruiz Racing Bicycles' folder under the 'Documents and Settings' folder, on the C drive (Windows).



The following program is the same as the preceding one except that it creates a permanent SAS data set. Notice that a two-level name appears in the DATA statement.

```
DATA Mylib.distance;
   Miles = 26.22;
   Kilometers = 1.61 * Miles;
PROC PRINT DATA = Mylib.distance;
RUN;
```

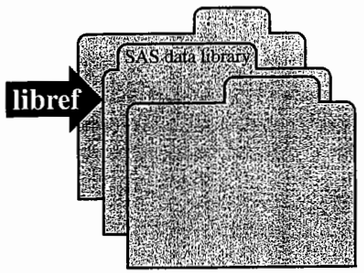
This time the log contains this note:

```
NOTE: The data set MYLIB.DISTANCE has 1 observations and 2 variables.
```

This is a permanent SAS data set because the libref is not WORK.

[1] With batch processing under OS/390 or z/OS, you may also use Job Control Language (JCL). The DDname is your libref.

2.20 Using Permanent SAS Data Sets with LIBNAME Statements



A libref is a nickname that corresponds to the location of a SAS data library. When you use a libref as the first level in the name of a SAS data set, SAS knows to look for that data set in that location. This section shows you how to define a libref using the LIBNAME statement which is the most universal (and therefore most portable) method of creating a libref. You can also define a libref using the New Library window (covered in section 1.11) or for some computers, operating environment control language.[1] The basic form of the LIBNAME statement is

```
LIBNAME libref 'your-SAS-data-library';
```

After the keyword LIBNAME, you specify the libref and then the location of your permanent SAS data set in quotation marks. Librefs must be eight characters or shorter; start with a letter or underscore; and contain only letters, numerals, or underscores. Here is the general form of LIBNAME statements for individual operating environments:

```
Windows:        LIBNAME libref 'drive:\directory';
UNIX:           LIBNAME libref '/home/path';
OpenVMS:        LIBNAME libref '[userid.directory]';
OS/390 or z/OS: LIBNAME libref 'data-set-name';
```

Creating a permanent SAS data set The following example creates a permanent SAS data set containing information about magnolia trees. For each type of tree the raw data file includes the scientific name, common name, maximum height, age at first blooming when planted from seed, whether evergreen or deciduous, and color of flowers.

```
M. grandiflora Southern Magnolia 80 15 E white
M. campbellii                    80 20 D rose
M. liliiflora  Lily Magnolia     12  4 D purple
M. soulangiana Saucer Magnolia   25  3 D pink
M. stellata    Star Magnolia     10  3 D white
```

This program sets up a libref named PLANTS pointing to the MySASLib directory on the C drive (Windows). Then it reads the raw data from a file called Mag.dat, creating a permanent SAS data set named MAGNOLIA which is stored in the PLANTS library.

```
LIBNAME plants 'c:\MySASLib';
DATA plants.magnolia;
   INFILE 'c:\MyRawData\Mag.dat';
   INPUT ScientificName $ 1-14 CommonName $ 16-32 MaximumHeight
      AgeBloom Type $ Color $;
RUN;
```

[1] With batch processing under OS/390 or z/OS, you may use Job Control Language (JCL). The DDname is your libref.

The log contains this note showing the two-level data set name:

```
NOTE: The data set PLANTS.MAGNOLIA has 5 observations and 6 variables.
```

If you print a directory of files on your computer, you will not see a file named PLANTS.MAGNOLIA. That is because operating environments have their own systems for naming files. When run under Windows, UNIX, or OpenVMS, this data set will be called `magnolia.sas7bdat`. Under OS/390 or z/OS, the filename would be the *data-set-name* specified in the LIBNAME statement.

Reading a permanent SAS data set To use a permanent SAS data set, you can include a LIBNAME statement in your program and refer to the data set by its two-level name. For instance, if you wanted to go back later and print the permanent data set created in the last example, you could use the following statements:

```
LIBNAME example 'c:\MySASLib';
PROC PRINT DATA=example.magnolia;
   TITLE 'Magnolias';
RUN;
```

This time the libref in the LIBNAME statement is EXAMPLE instead of PLANTS, but it points to the same location as before, the MySASLib directory on the C drive. The libref can change, but the member name, MAGNOLIA, stays the same.

The output looks like this:

Magnolias 1

Obs	ScientificName	CommonName	Maximum Height	Age Bloom	Type	Color
1	M. grandiflora	Southern Magnolia	80	15	E	white
2	M. campbellii		80	20	D	rose
3	M. liliiflora	Lily Magnolia	12	4	D	purple
4	M. soulangiana	Saucer Magnolia	25	3	D	pink
5	M. stellata	Star Magnolia	10	3	D	white

2.21 Using Permanent SAS Data Sets by Direct Referencing

If you don't want to be bothered with setting up librefs and defining SAS libraries, but you still want to use permanent SAS data sets, then you can use direct referencing. Direct referencing still uses SAS libraries, but instead of defining the library yourself, you let SAS do it for you.

Using direct referencing is easy, just take your operating environment's name for a file, enclose it in quotation marks, and put it in your program. The quotation marks tell SAS that this is a permanent SAS data set. Here is the general form of the DATA statement for creating permanent SAS data sets under different operating environments:

```
Windows:        DATA 'drive:\directory\filename';
UNIX:           DATA '/home/path/filename';
OpenVMS:        DATA '[userid.directory]filename';
OS/390 or z/OS: DATA 'data-set-name';
```

For directory-based operating environments, if you leave out the directory or path, then SAS uses the current working directory. For example, this statement would create a permanent SAS data set named TREES in your current working directory.

```
DATA 'trees';
```

For UNIX and OpenVMS operating environments, by default, your current directory is the directory where you started SAS. You can change the current directory for the SAS session by choosing `Change Directory` from the Options menu of the Tools pull-down menu. Under Windows the name of the current working directory is displayed at the bottom of the SAS window. You can change the directory for the current SAS session by double-clicking on the directory name which will open the Change Folder window.

Example The following example creates a permanent SAS data set containing information about magnolia trees. For each type of tree the raw data file includes the scientific name, common name, maximum height, age at first blooming when planted from seed, whether evergreen or deciduous, and color of flowers.

```
M. grandiflora Southern Magnolia 80 15 E white
M. campbellii                    80 20 D rose
M. liliiflora  Lily Magnolia     12  4 D purple
M. soulangiana Saucer Magnolia   25  3 D pink
M. stellata    Star Magnolia     10  3 D white
```

This program reads the raw data from a file called Mag.dat, creating a permanent SAS data set named MAGNOLIA. The MAGNOLIA data set is stored in the MySASLib directory on the C drive (Windows).

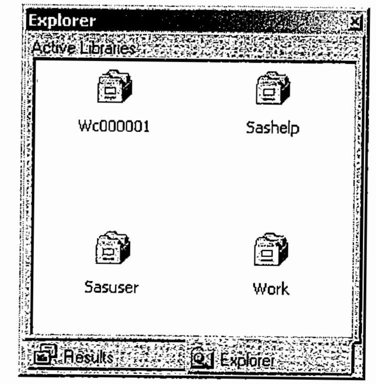
```
DATA 'c:\MySASLib\magnolia';
   INFILE 'c:\MyRawData\Mag.dat';
   INPUT ScientificName $ 1-14 CommonName $ 16-32 MaximumHeight
      AgeBloom Type $ Color $;
RUN;
```

If you look in your SAS log you will see this note:

```
NOTE: The data set c:\MySASLib\magnolia has 5 observations and 6 variables.
```


This is a permanent SAS data set, so SAS will not erase it. If you list the files in the MySASLib directory, you will see a file named magnolia.sas7bdat. Notice that SAS automatically appended a file extension, even though no extension appeared in the SAS program.

When you put quotation marks around your data set name, you are using direct referencing, and SAS creates a permanent SAS data set. Since you haven't specified a libref, SAS makes up a libref for you. You don't need to know the name of the libref that SAS makes up, but it is there and, you can see it in the Active Libraries window. This is what the Active Libraries window looks like after running the previous program. SAS has created a library named Wc000001 which contains the MAGNOLIA data set.



Reading SAS data sets using direct referencing To read a permanent SAS data set using direct referencing, simply enclose the path and name for the data set in quotation marks wherever you would use a SAS data set name. For example, to print the MAGNOLIA data set, you could use the following statements:

```
PROC PRINT DATA = 'c:\MySASLib\magnolia';
   TITLE 'Magnolias';
RUN;
```

The output looks like this:

Magnolias 1

Obs	ScientificName	CommonName	Maximum Height	Age Bloom	Type	Color
1	M. grandiflora	Southern Magnolia	80	15	E	white
2	M. campbellii		80	20	D	rose
3	M. liliiflora	Lily Magnolia	12	4	D	purple
4	M. soulangiana	Saucer Magnolia	25	3	D	pink
5	M. stellata	Star Magnolia	10	3	D	white

2.22 Listing the Contents of a SAS Data Set

To use a SAS data set, all you need to do is tell SAS the name and location of the data set you want, and SAS will figure out what is in it. SAS can do this because SAS data sets are self-documenting, which is another way of saying that SAS automatically stores information about the data set (also called the descriptor portion) along with the data. You can't display a SAS data set on your computer screen using a word processor. However, there is an easy way to get a description of a SAS data set; you simply run the CONTENTS procedure.

PROC CONTENTS is a simple procedure. In most cases you just type the keywords PROC CONTENTS and specify the data set you want with the DATA= option:

```
PROC CONTENTS DATA = data-set;
```

If you omit the DATA= option, then by default SAS will use the most recently created data set.

Example The following DATA step creates a data set so we can run PROC CONTENTS:

```
DATA funnies;
   INPUT Id Name $ Height Weight DoB;
   LABEL Id   = 'Identification no.'
      Height = 'Height in inches'
      Weight = 'Weight in pounds'
      DoB    = 'Date of birth';
   INFORMAT DoB MMDDYY8.;
   FORMAT DoB WORDDATE18.;
   DATALINES;
53 Susie    42  41  07-11-81
54 Charlie  46  55  10-26-54
55 Calvin   40  35  01-10-81
56 Lucy     46  52  01-13-55
   ;
* Use PROC CONTENTS to describe data set funnies;
PROC CONTENTS DATA = funnies;
RUN;
```

Note that the DATA step above includes a LABEL statement. For each variable, you can specify a label up to 256 characters long. These optional labels allow you to document your variables in more detail than is possible with just variable names. If you specify a LABEL statement in a DATA step, then the descriptions will be stored in the data set and will be printed by PROC CONTENTS. You can also use LABEL statements in PROC steps to customize your reports, but then the labels apply only for the duration of the PROC step and are not stored in the data set.

INFORMAT and FORMAT statements also appear in this program. You can use these optional statements to associate informats or formats with variables. Just as informats give SAS special instructions for reading a variable, formats give SAS special instructions for writing a variable. If you specify an INFORMAT or FORMAT statement in a DATA step, then the name of that informat or format will be saved in the data set and printed by PROC CONTENTS. FORMAT statements, like LABEL statements, can be used in PROC steps to customize your reports, but then the name of the format is not stored in the data set.[1]

The output from PROC CONTENTS is like a table of contents for your data set:

```
                    The CONTENTS Procedure

❶ Data Set Name      WORK.FUNNIES                ❷ Observations          4
  Member Type        DATA                        ❸ Variables             5
  Engine             V9                            Indexes               0
❹ Created            13:36 Monday, May 12, 2003    Observation Length    40
  Last Modified      13:36 Monday, May 12, 2003    Deleted Observations  0
  Protection                                       Compressed            NO
  Data Set Type                                    Sorted                NO
  Label
  Data Representation WINDOWS
  Encoding wlatin1    wlatin1 Western (Windows)

             -----Engine/Host Dependent Information-----
  Data Set Page Size          4096
  Number of Data Set Pages    1
  First Data Page             1
  Max Obs per Page            101
  Obs in First Data Page      4
  Number of Data Set Repairs  0
  File Name                   C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\SAS
                              Temporary Files\_TD832\funnies.sas7bdat
  Release Created             9.0000M0
  Host Created                XP_HOME

          -----Alphabetic List of Variables and Attributes-----
  #   Variable ❶Type ❷Len   ❸Format      ❹Informat ❺Label

  5   DoB        Num    8   WORDDATE18.  MMDDYY8.  Date of birth
  3   Height     Num    8                          Height in inches
  1   Id         Num    8                          Identification no.
  2   Name       Char   8
  4   Weight     Num    8                          Weight in pounds
```

The output starts with information about your data set and then describes each variable.

For the data set

❶ Data set name
❷ Number of observations
❸ Number of variables
❹ Date created

For each variable

❶ Type (numeric or character)
❷ Length (storage size in bytes)
❸ Format for printing (if any)
❹ Informat for input (if any)
❺ Label (if any)

"Contrariwise," continued Tweedledee," if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

LEWIS CARROLL

From *Alice Through the Looking Glass* by Lewis Carroll. Public domain.

CHAPTER 3

Working with Your Data

3.1 Creating and Redefining Variables

If someone were to compile a list of the most popular things to do with SAS software, creating and redefining variables would surely be on it. Fortunately, SAS is flexible and uses a common sense approach to these tasks. You create and redefine variables with assignment statements using this basic form:

```
variable = expression;
```

On the left side of the equal sign is a variable name, either new or old. On the right side of the equal sign may appear a constant, another variable, or a mathematical expression. Here are examples of these basic types of assignment statements:

Type of expression	Assignment statement
numeric constant	`Qwerty = 10;`
character constant	`Qwerty = 'ten';`
a variable	`Qwerty = OldVar;`
addition	`Qwerty = OldVar + 10;`
subtraction	`Qwerty = OldVar - 10;`
multiplication	`Qwerty = OldVar * 10;`
division	`Qwerty = OldVar / 10;`
exponentiation	`Qwerty = OldVar ** 10;`

Whether the variable Qwerty is numeric or character depends on the expression that defines it. When the expression is numeric, Qwerty will be numeric; when it is character, Qwerty will be character.

When deciding how to interpret your expression, SAS follows the standard mathematical rules of precedence: SAS performs exponentiation first, then multiplication and division, followed by addition and subtraction. You can use parentheses to override that order. Here are two similar SAS statements showing that a couple of parentheses can make a big difference:

Assignment statement	Result
`x = 10 * 4 + 3 ** 2;`	`x = 49`
`x = 10 * (4 + 3 ** 2);`	`x = 130`

While SAS can read expressions with or without parentheses, people often can't. If you use parentheses, your programs will be a lot easier to read.

Example The following raw data are from a survey of home gardeners. Gardeners were asked to estimate the number of pounds they harvested for four crops: tomatoes, zucchini, peas, and grapes.

```
Gregor  10  2  40    0
Molly   15  5  10 1000
Luther  50 10  15   50
Susan   20  0   .   20
```


This program reads the data from a file called Garden.dat and then modifies the data:

```
* Modify homegarden data set with assignment statements;
DATA homegarden;
   INFILE 'c:\MyRawData\Garden.dat';
   INPUT Name $ 1-7 Tomato Zucchini Peas Grapes;
   Zone = 14;
   Type = 'home';
   Zucchini = Zucchini * 10;
   Total = Tomato + Zucchini + Peas + Grapes;
   PerTom = (Tomato / Total) * 100;
PROC PRINT DATA = homegarden;
   TITLE 'Home Gardening Survey';
RUN;
```

This program contains five assignment statements. The first creates a new variable, Zone, equal to a numeric constant, 14. The variable Type is set equal to a character constant, home. The variable Zucchini is multiplied by 10 because that just seems natural for zucchini. Total is the sum for all the types of plants. PerTom is not a genetically engineered tomato but the percentage of harvest which were tomatoes. The report from PROC PRINT contains all the variables, old and new:

Home Gardening Survey 1

Obs	Name	Tomato	Zucchini	Peas	Grapes	Zone	Type	Total	PerTom
1	Gregor	10	20	40	0	14	home	70	14.2857
2	Molly	15	50	10	1000	14	home	1075	1.3953
3	Luther	50	100	15	50	14	home	215	23.2558
4	Susan	20	0	.	20	14	home	.	.

Notice that the variable Zucchini appears only once because the new value replaced the old value. The other four assignment statements each created a new variable. When a variable is new, SAS adds it to the data set you are creating. When a variable already exists, SAS replaces the original value with the new one. Using an existing name has the advantage of not cluttering your data set with a lot of similar variables. However, you don't want to overwrite a variable unless you are really sure you won't need the original value later.

The variable Peas had a missing value for the last observation. Because of this, the variables Total and PerTom, which are calculated from Peas, were also set to missing and this message appeared in the log:

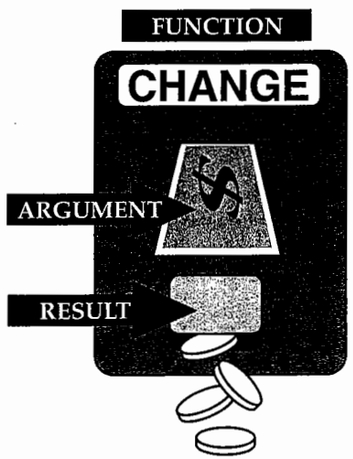
```
NOTE: Missing values were generated as a result of performing an operation on
      missing values.
```

This message is a flag that often indicates an error. However, in this case it is not an error but simply the result of incomplete data collection.[1]

[1] If you want to add only non-missing values, you can use the SUM function discussed in section 10.7.

3.2 Using SAS Functions

Sometimes a simple expression, using only arithmetic operators, does not give you the new value you are looking for. This is where functions are handy, simplifying your task because SAS has already done the programming for you. All you need to do is plug the right values into the function and out comes the result—like putting a dollar in a change machine and getting back four quarters.



SAS has over 400 functions in the following general areas:

Character	Probability
Date and Time	Random Number
Financial	Sample Statistics
Macro	State and ZIP Code
Mathematical	

Section 3.3 gives a sample of the most common SAS functions.

Functions perform a calculation on, or a transformation of, the arguments given in parentheses following the function name. SAS functions have the following general form:

```
function-name(argument, argument, ...)
```

All functions must have parentheses even if they don't require any arguments. Arguments are separated by commas and can be variable names, constant values such as numbers or characters enclosed in quotation marks, or expressions. The following statement computes Birthday as a SAS date value using the function MDY and the variables MonthBorn, DayBorn, and YearBorn. The MDY function takes three arguments, one each for the month, day, and year:

```
Birthday = MDY(MonthBorn, DayBorn, YearBorn);
```

Functions can be nested, where one function is the argument of another function. For example, the following statement calculates NewValue using two nested functions, INT and LOG:

```
NewValue = INT(LOG(10));
```

The result for this example is 2, the integer portion of the natural log of the numeric constant 10 (2.3026). Just be careful when nesting functions that each parenthesis has a mate.

Example Data from a pumpkin carving contest illustrate the use of several functions. The contestants' names are followed by their age, type of pumpkin (carved or decorated), date of entry, and the scores from five judges:


```
Alicia Grossman  13 c 10-28-2003 7.8 6.5 7.2 8.0 7.9
Matthew Lee       9 D 10-30-2003 6.5 5.9 6.8 6.0 8.1
Elizabeth Garcia 10 C 10-29-2003 8.9 7.9 8.5 9.0 8.8
Lori Newcombe     6 D 10-30-2003 6.7 5.6 4.9 5.2 6.1
Jose Martinez     7 d 10-31-2003 8.9 9.510.0 9.7 9.0
Brian Williams   11 C 10-29-2003 7.8 8.4 8.5 7.9 8.0
```

The following program reads the data, creates two new variables (AvgScore and DayEntered) and transforms another (Type):

```
DATA contest;
   INFILE 'c:\MyRawData\Pumpkin.dat';
   INPUT Name $16. Age 3. +1 Type $1. +1 Date MMDDYY10.
         (Scr1 Scr2 Scr3 Scr4 Scr5) (4.1);
   AvgScore = MEAN(Scr1, Scr2, Scr3, Scr4, Scr5);
   DayEntered = DAY(Date);
   Type = UPCASE(Type);
PROC PRINT DATA = contest;
   TITLE 'Pumpkin Carving Contest';
RUN;
```

The variable AvgScore is created using the MEAN function, which returns the mean of the non-missing arguments. This differs from simply adding the arguments together and dividing by their number, which would return a missing value if any of the arguments were missing.

The variable DayEntered is created using the DAY function, which returns the day of the month. SAS has all sorts of functions for manipulating dates, and what's great about them is that you don't have to worry about things like leap year—SAS takes care of that for you.

The variable Type is transformed using the UPCASE function. SAS is case sensitive when it comes to variable values; a 'd' is not the same as 'D'. The data file has both lowercase and uppercase letters for the variable Type, so the function UPCASE is used to make all the values uppercase.

Here are the results:

Pumpkin Carving Contest 1

Obs	Name	Age	Type	Date[1]	Scr1	Scr2	Scr3	Scr4	Scr5	Avg Score	Day Entered
1	Alicia Grossman	13	C	16006	7.8	6.5	7.2	8.0	7.9	7.48	28
2	Matthew Lee	9	D	16008	6.5	5.9	6.8	6.0	8.1	6.66	30
3	Elizabeth Garcia	10	C	16007	8.9	7.9	8.5	9.0	8.8	8.62	29
4	Lori Newcombe	6	D	16008	6.7	5.6	4.9	5.2	6.1	5.70	30
5	Jose Martinez	7	D	16009	8.9	9.5	10.0	9.7	9.0	9.42	31
6	Brian Williams	11	C	16007	7.8	8.4	8.5	7.9	8.0	8.12	29

[1] Notice that these dates are printed as the number of days since January 1, 1960. Section 4.5 discusses how to format these values into readable dates.

3.3 Selected SAS Functions

The following table lists definitions and syntax of commonly used functions.[1]

Function name	Syntax[2]	Definition
Numeric		
INT	INT(*arg*)	Returns the integer portion of argument
LOG	LOG(*arg*)	Natural logarithm
LOG10	LOG10(*arg*)	Logarithm to the base 10
MAX	MAX(*arg,arg,...*)	Largest non-missing value
MEAN	MEAN(*arg,arg,...*)	Arithmetic mean of non-missing values
MIN	MIN(*arg,arg,...*)	Smallest non-missing value
ROUND	ROUND(*arg, round-off-unit*)	Rounds to nearest round-off unit
SUM	SUM(*arg,arg,...*)	Sum of non-missing values
Character		
LEFT	LEFT(*arg*)	Left aligns a SAS character expression
LENGTH	LENGTH(*arg*)	Returns the length of an argument not counting trailing blanks (missing values have a length of 1)
SUBSTR	SUBSTR(*arg,position,n*)	Extracts a substring from an argument starting at '*position*' for '*n*' characters or until end if no '*n*'[3]
TRANSLATE	TRANSLATE(*source,to-1, from-1,...to-n,from-n*)	Replaces '*from*' characters in '*source*' with '*to*' characters (one to one replacement only—you can't replace one character with two, for example)
TRIM	TRIM(*arg*)	Removes trailing blanks from character expression
UPCASE	UPCASE(*arg*)	Converts all letters in argument to uppercase
Date		
DATEJUL	DATEJUL(*julian-date*)	Converts a Julian date to a SAS date value[4]
DAY	DAY(*date*)	Returns the day of the month from a SAS date value
MDY	MDY(*month,day,year*)	Returns a SAS date value from month, day, and year values
MONTH	MONTH(*date*)	Returns the month (1-12) from a SAS date value
QTR	QTR(*date*)	Returns the yearly quarter (1-4) from a SAS date value
TODAY	TODAY()	Returns the current date as a SAS date value

[1] Check the SAS Help and Documentation for a complete list of functions.

[2] *arg* is short for argument, which means a literal value, variable name, or expression.

[3] SUBSTR has a different function when on the left side of an equal sign.

[4] A SAS date value is the number of days since January 1, 1960.

Here are examples using the selected functions.

Function name	Example	Result	Example	Result
Numeric				
INT	`x=INT(4.32);`	`x=4`	`y=INT(5.789);`	`y=5`
LOG	`x=LOG(1);`	`x=0.0`	`y=LOG(10);`	`y=2.30259`
LOG10	`x=LOG10(1);`	`x=0.0`	`y=LOG10(10);`	`y=1.0`
MAX	`x=MAX(9.3,8,7.5);`	`x=9.3`	`y=MAX(-3,.,5);`	`y=5`
MEAN	`x=MEAN(1,4,7,2);`	`x=3.5`	`y=MEAN(2,.,3);`	`y=2.5`
MIN	`x=MIN(9.3,8,7.5);`	`x=7.5`	`y=MIN(-3,.,5);`	`y=-3`
ROUND	`x=ROUND(12.65);`	`x=13`	`y=ROUND(12.65,.1);`	`y=12.7`
SUM	`x=SUM(3,5,1);`	`x=9.0`	`y=SUM(4,7,.);`	`y=11`
Character				
LEFT	`a=' cat';` `x=LEFT(a);`	`x='cat '`	`a=' my cat';` `y=LEFT(a);`	`y='my cat '`
LENGTH	`a='my cat';` `x=LENGTH(a);`	`x=6`	`a=' my cat ';` `y=LENGTH(a);`	`y=7`
SUBSTR	`a='(916)734-6281';` `x=SUBSTR(a,2,3);`	`x='916'`	`y=SUBSTR('1cat',2);`	`y='cat'`
TRANSLATE	`a='6/16/99';` `x=TRANSLATE (a,'-','/');`	`x='6-16-99'`	`a='my cat can';` `y=TRANSLATE (a, 'r','c');`	`y='my rat ran'`
TRIM	`a='my ';b='cat';` `x=TRIM(a)||b;`[5]	`x='mycat '`	`a='my cat ';b='s';` `y=TRIM(a)||b;`	`y='my cats '`
UPCASE	`a='MyCat';` `x=UPCASE(a);`	`x='MYCAT'`	`y=UPCASE('Tiger');`	`y='TIGER'`
Date				
DATEJUL	`a=60001;` `x=DATEJUL(a);`	`x=0`	`a=60365;` `y=DATEJUL(a);`	`y=364`
DAY	`a=MDY(4,18,1999);` `x=DAY(a);`	`x=18`	`a=MDY(9,3,60);` `y=DAY(a);`	`y=3`
MDY	`x=MDY(1,1,1960);`	`x=0`	`m=2; d=1; y=60;` `Date=MDY(m,d,y);`	`Date=31`
MONTH	`a=MDY(4,18,1999);` `x=MONTH(a);`	`x=4`	`a=MDY(9,3,60);` `y=MONTH(a);`	`y=9`
QTR	`a=MDY(4,18,1999);` `x=QTR(a);`	`x=2`	`a=MDY(9,3,60);` `y=QTR(a);`	`y=3`
TODAY	`x=TODAY();`	`x=today's date`	`x=TODAY()-1;`	`x=yesterday's date`

[5] The concatenation operator || concatenates character strings.

3.4 Using IF-THEN Statements

Frequently, you want an assignment statement to apply to some observations but not all—under some conditions, but not others. This is called conditional logic, and you do it with IF-THEN statements:

```
IF condition THEN action;
```

The *condition* is an expression comparing one thing to another, and the *action* is what SAS should do when the expression is true, often an assignment statement. For example

```
IF Model = 'Mustang' THEN Make = 'Ford';
```

This statement tells SAS to set the variable Make equal to Ford whenever the variable Model equals Mustang. The terms on either side of the comparison may be constants, variables, or expressions. Those terms are separated by a comparison operator, which may be either symbolic or mnemonic. The decision of whether to use symbolic or mnemonic operators depends on your personal preference and the symbols available on your keyboard. Here are the basic comparison operators:

Symbolic	Mnemonic	Meaning
=	EQ	equals
¬ =, ^ =, or ~ =	NE	not equal
>	GT	greater than
<	LT	less than
> =	GE	greater than or equal
< =	LE	less than or equal

The IN operator also makes comparisons, but it works a bit differently. IN compares the value of a variable to a list of values. Here is an example:

```
IF Model IN ('Corvette', 'Camaro') THEN Make = 'Chevrolet';
```

This statement tells SAS to set the variable Make equal to Chevrolet whenever the value of Model is Corvette or Camaro.

A single IF-THEN statement can only have one action. If you add the keywords DO and END, then you can execute more than one action. For example

```
IF condition THEN DO;
   action;
   action;
END;
```

```
IF Model = 'Mustang' THEN DO;
   Make = 'Ford';
   Size = 'compact';
END;
```

The DO statement causes all SAS statements coming after it to be treated as a unit until a matching END statement appears. Together, the DO statement, the END statement, and all the statements in between are called a DO group.

You can also specify multiple conditions with the keywords AND and OR:

```
IF condition AND condition THEN action;
```


For example

```
IF Model = 'Mustang' AND Year < 1975 THEN Status = 'classic';
```

Like the comparison operators, AND and OR may be symbolic or mnemonic:

Symbolic	Mnemonic	Meaning
&	AND	all comparisons must be true
|, ¦, or !	OR	only one comparison must be true

Be careful with long strings of comparisons; they can be a logical maze.

Example The following data about used cars contain values for model, year, make, number of seats, and color:

```
Corvette 1955 .      2 black
XJ6      1995 Jaguar 2 teal
Mustang  1966 Ford   4 red
Miata    2002 .      . silver
CRX      2001 Honda  2 black
Camaro   2000 .      4 red
```

This program reads the data from a file called Cars.dat, uses a series of IF-THEN statements to fill in missing data, and creates a new variable, Status:

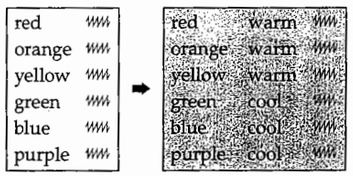
```
DATA sportscars;
   INFILE 'c:\MyRawData\Cars.dat';
   INPUT Model $ Year Make $ Seats Color $;
   IF Year < 1975 THEN Status = 'classic';
   IF Model = 'Corvette' OR Model = 'Camaro' THEN Make = 'Chevy';
   IF Model = 'Miata' THEN DO;
      Make = 'Mazda';
      Seats = 2;
   END;
PROC PRINT DATA = sportscars;
   TITLE "Eddy's Excellent Emporium of Used Sports Cars";
RUN;
```

This program contains three IF-THEN statements. The first is a simple IF-THEN that creates the new variable Status based on the value of Year. That is followed by a compound IF-THEN using an OR. The last IF-THEN uses DO and END. The output looks like this:

Eddy's Excellent Emporium of Used Sports Cars 1

Obs	Model	Year	Make	Seats	Color	Status
1	Corvette	1955	Chevy	2	black	classic
2	XJ6	1995	Jaguar	2	teal	
3	Mustang	1966	Ford	4	red	classic
4	Miata	2002	Mazda	2	silver	
5	CRX	2001	Honda	2	black	
6	Camaro	2000	Chevy	4	red	

3.5 Grouping Observations with IF-THEN/ELSE Statements



One of the most common uses of IF-THEN statements is for grouping observations. Perhaps a variable has too many different values and you want to print a more compact report, or perhaps you are going to run an analysis based on specific groups of interest. There are many possible reasons for grouping data, so sooner or later you'll probably need to do it.

The simplest and most common way to create a grouping variable is with a series of IF-THEN statements.[1] By adding the keyword ELSE to your IF statements, you can tell SAS that these statements are related.

IF-THEN/ELSE logic takes this basic form:

```
IF condition THEN action;
   ELSE IF condition THEN action;
   ELSE IF condition THEN action;
```

Notice that the ELSE statement is simply an IF-THEN statement with an ELSE tacked onto the front. You can have any number of these statements.

IF-THEN/ELSE logic has two advantages when compared to a simple series of IF-THEN statements without any ELSE statements. First, it is more efficient, using less computer time; once an observation satisfies a condition, SAS skips the rest of the series. Second, ELSE logic ensures that your groups are mutually exclusive so you don't accidentally have an observation fitting into more than one group.

Sometimes the last ELSE statement in a series is a little different, containing just an action, with no IF or THEN. Note the final ELSE statement in this series:

```
IF condition THEN action;
   ELSE IF condition THEN action;
   ELSE action;
```

An ELSE of this kind becomes a default which is automatically executed for all observations failing to satisfy any of the previous IF statements. You can only have one of these statements, and it must be the last in the IF-THEN/ELSE series.

Example Here are data from a survey of home improvements. Each record contains three data values: owner's name, description of the work done, and cost of the improvements in dollars:

```
Bob     kitchen cabinet face-lift   1253.00
Shirley bathroom addition          11350.70
Silvia  paint exterior                    .
Al      backyard gazebo             3098.63
Norm    paint interior               647.77
Kathy   second floor addition      75362.93
```

[1] Other ways to create grouping variables include using a SELECT statement, or using a PUT function with a user-defined format from PROC FORMAT.

This program reads the raw data from a file called Home.dat and then assigns a grouping variable called CostGroup. This variable has a value of high, medium, low, or missing, depending on the value of Cost:

```
* Group observations by cost;
DATA homeimprovements;
   INFILE 'c:\MyRawData\Home.dat';
   INPUT Owner $ 1-7 Description $ 9-33 Cost;
   IF Cost = . THEN CostGroup = 'missing';
     ELSE IF Cost < 2000 THEN CostGroup = 'low';
     ELSE IF Cost < 10000 THEN CostGroup = 'medium';
     ELSE CostGroup = 'high';
PROC PRINT DATA = homeimprovements;
   TITLE 'Home Improvement Cost Groups';
RUN;
```

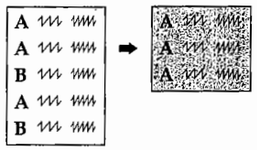
Notice that there are four statements in this IF-THEN/ELSE series, one for each possible value of the variable CostGroup. The first statement deals with observations that have missing data for the variable Cost. Without this first statement, observations with a missing value for Cost would be incorrectly assigned a CostGroup of low. SAS considers missing values to be smaller than non-missing values, smaller than any printable character for character variables, and smaller than negative numbers for numeric variables. Unless you are sure that your data contain no missing values, you should allow for missing values when you write IF-THEN/ELSE statements.

The results look like this:

Home Improvement Cost Groups 1

Obs	Owner	Description	Cost	Cost Group
1	Bob	kitchen cabinet face-lift	1253.00	low
2	Shirley	bathroom addition	11350.70	high
3	Silvia	paint exterior	.	missing
4	Al	backyard gazebo	3098.63	medium
5	Norm	paint interior	647.77	low
6	Kathy	second floor addition	75362.93	high

3.6 Subsetting Your Data



Often programmers find that they want to use some of the observations in a data set and exclude the rest. The most common way to do this is with a subsetting IF statement in a DATA step.[1] The basic form of a subsetting IF is

```
IF expression;
```

Consider this example:

```
IF Sex = 'f';
```

At first subsetting IF statements may seem odd. People naturally ask, "IF Sex = 'f', then what?" The subsetting IF looks incomplete, as if a careless typist pressed the delete key too long. But it is really a special case of the standard IF-THEN statement. In this case the action is merely implied. If the expression is true, then SAS continues with the DATA step. If the expression is false, then no further statements are processed for that observation; that observation is not added to the data set being created; and SAS moves on to the next observation. You can think of the subsetting IF as a kind of on-off switch. If the condition is true, then the switch is on and the observation is processed. If the condition is false, then that observation is turned off.

If you don't like subsetting IFs, there is another alternative, the DELETE statement. DELETE statements do the opposite of subsetting IFs. While the subsetting IF statement tells SAS which observations to include, the DELETE statement tells SAS which observations to exclude:

```
IF expression THEN DELETE;
```

The following two statements are equivalent (assuming there are only two values for the variable Sex, and no missing data):

```
IF Sex = 'f';        IF Sex = 'm' THEN DELETE;
```

Example The members of a local amateur playhouse want to choose a Shakespearean comedy for this spring's play. You volunteer to compile a list of titles using an online encyclopedia. For each play your data file contains title, approximate year of first performance, and type of play:

```
A Midsummer Night's Dream 1595 comedy
Comedy of Errors          1590 comedy
Hamlet                    1600 tragedy
Macbeth                   1606 tragedy
Richard III               1594 history
Romeo and Juliet          1596 tragedy
Taming of the Shrew       1593 comedy
Tempest                   1611 romance
```

[1] Other ways to subset data include using multiple INPUT statements (discussed in section 2.13), and the WHERE statement (discussed in section 4.2 and appendix F).

This program reads the data from a raw data file called Shakespeare.dat and then uses a subsetting IF statement to select only comedies:

```
* Choose only comedies;
DATA comedy;
   INFILE 'c:\MyRawData\Shakespeare.dat';
   INPUT Title $ 1-26 Year Type $;
   IF Type = 'comedy';
PROC PRINT DATA = comedy;
   TITLE 'Shakespearean Comedies';
RUN;
```

The output looks like this:

Shakespearean Comedies 1

Obs	Title	Year	Type
1	A Midsummer Night's Dream	1595	comedy
2	Comedy of Errors	1590	comedy
3	Taming of the Shrew	1593	comedy

These notes appear in the log stating that although eight records were read from the input file, the data set WORK.COMEDY contains only three observations:

```
NOTE: 8 records were read from the infile 'c:\MyRawData\Shakespeare.dat'
NOTE: The data set WORK.COMEDY has 3 observations and 3 variables.
```

It is always a good idea to check the SAS log when you subset observations to make sure that you ended up with what you expected.

In the program above, you could substitute the statement

```
IF Type = 'tragedy' OR Type = 'romance' OR Type = 'history' THEN DELETE;
```

for the statement

```
IF Type = 'comedy';
```

But you would have to do a lot more typing. Generally, you use the subsetting IF when it is easier to specify a condition for including observations, and use the DELETE statement when is easier to specify a condition for excluding observations.

3.7 Working with SAS Dates

Dates can be tricky to work with. Some months have 30 days, some 31, some 28, and don't forget leap year. SAS dates simplify all this. A SAS date is a numeric value equal to the number of days since January 1, 1960.[1] The table below lists four dates and their values as SAS dates:

Date	SAS date value
January 1, 1959	-365
January 1, 1960	0
January 1, 1961	366
January 1, 2003	15706

SAS has special tools for working with dates: informats for reading dates, functions for manipulating dates, and formats for printing dates.[2] A table of selected date informats, formats, and functions appears in section 3.8.

Informats To read variables that are dates, you use formatted style input. The INPUT statement below tells SAS to read a variable named BirthDate using the MMDDYY10. informat:

```
INPUT BirthDate MMDDYY10.;
```

SAS has a variety of date informats for reading dates in many different forms. All of these informats convert your data to a number equal to the number of days since January 1, 1960.[3]

Setting the default century When SAS sees a date with a two-digit year like 07/04/76, SAS has to decide in which century the year belongs. Is the year 1976, 2076, or perhaps 1776? The system option YEARCUTOFF= specifies the first year of a hundred-year span for SAS to use. The default value for this option is 1920, but you can change this value with the OPTIONS statement. To avoid problems, you may want to specify the YEARCUTOFF= option whenever you have data containing two-digit years. This statement tells SAS to interpret two-digit dates as occurring between 1950 and 2049:

```
OPTIONS YEARCUTOFF = 1950;
```

Dates in SAS expressions Once a variable has been read with a SAS date informat, it can be used in arithmetic expressions like other numeric variables. For example, if a library book is due in three weeks, you could find the due date by adding 21 days to the date it was checked out:

```
DateDue = DateCheck + 21;
```

You can use a date as a constant in a SAS expression by adding quotation marks and a letter D. The assignment statement below creates a variable named EarthDay05, which is equal to the SAS date value for April 22, 2005:

```
EarthDay05 = '22APR2005'D;
```

[1] We don't know why this date was chosen, but since SAS dates are relative, January 1, 1960, is as good as any other date.

[2] SAS also has informats, functions, and formats for working with time values (the number of seconds since midnight), and datetime values (the number of seconds since midnight, you guessed it, January 1, 1960).

[3] For more information about informats, see section 2.7; for functions, see section 3.2; and for formats, see section 4.5.

Functions SAS date functions perform a number of handy operations. For example, the TODAY function returns a SAS date value equal to today's date. This statement

```
DaysOverDue = TODAY() - DateDue;
```

subtracts the date a book was due from today's date to compute the number of days a book is overdue.

Formats If you print a SAS date value, SAS will by default print the actual value—the number of days since January 1, 1960. Since this is not very meaningful to most people, SAS has a variety of formats for printing dates in different forms. The FORMAT statement below tells SAS to print the variable BirthDate using the WEEKDATE17. format:

```
FORMAT BirthDate WEEKDATE17.;
```

Example A local library has a data file containing details about library cards. Each record contains three data values—the card holder's name, birthdate, and the date that card was issued:

```
A. Jones     1jan60     9-15-03
M. Rincon    05OCT1949 02-29-2000
Z. Grandage 18mar1988 10-10-2002
K. Kaminaka 29may2001 01-24-2003
```

The program below reads the raw data, and then computes the variable ExpireDate (for expiration date) by adding three years to the variable IssueDate. The variable ExpireQuarter (the quarter the card expires) is computed using the QTR function and the variable ExpireDate. Then an IF statement uses a date constant to identify cards issued after January 1, 2003:

```
DATA librarycards;
   INFILE 'c:\MyRawData\Dates.dat' TRUNCOVER;
   INPUT Name $11. +1 BirthDate DATE9. +1 IssueDate MMDDYY10.;
   ExpireDate = IssueDate + (365.25 * 3);
   ExpireQuarter = QTR(ExpireDate);
   IF IssueDate > '01JAN2003'D THEN NewCard = 'yes';
PROC PRINT DATA = librarycards;
   FORMAT IssueDate MMDDYY8. ExpireDate WEEKDATE17.;
   TITLE 'SAS Dates without and with Formats.';
RUN;
```

Here is the output from PROC PRINT. Notice that the variable BirthDate is printed without a date format, while IssueDate and ExpireDate use formats:

SAS Dates without and with Formats 1

Obs	Name	Birth Date	Issue Date	ExpireDate	Expire Quarter	New Card
1	A. Jones	0	09/15/03	Thu, Sep 14, 2006	3	yes
2	M. Rincon	-3740	02/29/00	Fri, Feb 28, 2003	1	
3	Z. Grandage	10304	10/10/02	Sun, Oct 9, 2005	4	
4	K. Kaminaka	15124	01/24/03	Mon, Jan 23, 2006	1	yes

3.8 Selected Date Informats, Functions, and Formats

Here are definitions for some of the most commonly used date informats, functions, and formats.[1]

Informats	Definition	Width range	Default width
DATE*w.*	Reads dates in form: *ddmmmyy* or *ddmmmyyyy*	7-32	7
DDMMYY*w.*	Reads dates in form: *ddmmyy* or *ddmmyyyy*	6-32	6
JULIAN*w.*	Reads Julian dates in form: *yyddd* or *yyyyddd*	5-32	5
MMDDYY*w.*	Reads dates in form: *mmddyy* or *mmddyyyy*	6-32	6

Functions	Syntax	Definition
DATEJUL	DATEJUL(*julian-date*)	Converts a Julian date to a SAS date value[2]
DAY	DAY(*date*)	Returns the day of the month from a SAS date value
MDY	MDY(*month,day,year*)	Returns a SAS date value from month, day, and year values
MONTH	MONTH(*date*)	Returns the month (1-12) from a SAS date value
QTR	QTR(*date*)	Returns the yearly quarter (1-4) from a SAS date value
TODAY	TODAY()	Returns the current date as a SAS date value

Formats	Definition	Width range	Default width
DATE*w.*	Writes SAS date values in form: *ddmmmyy*	5-9	7
DAY*w.*	Writes the day of the month from a SAS date value	2-32	2
EURDFDD*w.*	Writes SAS date values in form: *dd.mm.yy*	2-10	8
JULIAN*w.*	Writes a Julian date from a SAS date value	5-7	5
MMDDYY*w.*	Writes SAS date values in form: *mmddyy* or *mmddyyyy*	2-10	8
WEEKDATE*w.*	Writes SAS date values in form: *day-of-week, month-name dd, yy or yyyy*	3-37	29
WORDDATE*w.*	Writes SAS date values in form: *month-name dd, yyyy*	3-32	18

[1] For a complete list see the SAS Help and Documentation.

[2] A SAS date value is the number of days since January 1, 1960.

Here are examples using the selected date informats, functions, and formats.

Informats	Input data	INPUT statement	Results
DATE*w.*	`1jan1961`	`INPUT Day DATE10.;`	`366`
DDMMYY*w.*	`01.01.61` `02/01/61`	`INPUT Day DDMMYY8.;`	`366` `367`
JULIAN*w.*	`61001`	`INPUT Day JULIAN7.;`	`366`
MMDDYY*w.*	`01-01-61`	`INPUT Day MMDDYY8.;`	`366`

Functions	Example	Result	Example	Results
DATEJUL	`a=60001;` `x=DATEJUL(a);`	`x=0`	`a=60365;` `y=DATEJUL(a);`	`y=364`
DAY	`a=MDY(4,18,99);` `x=DAY(a);`	`x=18`	`a=MDY(9,3,60);` `y=DAY(a);`	`y=3`
MDY	`x=MDY(1,1,60);`	`x=0`	`m=2; d=1; y=60;` `Date=MDY(m,d,y);`	`Date=31`
MONTH	`a=MDY(4,18,1999)` `x=MONTH(a);`	`x=4`	`a=MDY(9,3,60);` `y=MONTH(a);`	`y=9`
QTR	`a=MDY(4,18,99);` `x=QTR(a);`	`x=2`	`a=MDY(9,3,60);` `y=QTR(a);`	`y=3`
TODAY	`x=TODAY();`	`x=today's date`	`x=TODAY()-1;`	`x=yesterday's date`

Formats	Input data	PUT statement[3]	Results
DATE*w.*	`8966`	`PUT Birth DATE7.;` `PUT Birth DATE9.;`	`19JUL84` `19JUL1984`
DAY*w.*	`8966`	`PUT Birth DAY2.;` `PUT Birth DAY7.;`	`19` `19`
EURDFDD*w.*	`8966`	`PUT Birth EURDFDD8.` `PUT Birth EURDFDD10.;`	`19.07.84` `19.07.1984`
JULIAN*w.*	`8966`	`PUT Birth JULIAN5.;` `PUT Birth JULIAN7.;`	`84201` `1984201`
MMDDYY*w.*	`8966`	`PUT Birth MMDDYY8.;` `PUT Birth MMDDYY6.;`	`07/19/84` `071984`
WEEKDATE*w.*	`8966`	`PUT Birth WEEKDATE15.;` `PUT Birth WEEKDATE29.;`	`Thu, Jul 19, 84` `Thursday, July 19, 1984`
WORDDATE*w.*	`8966`	`PUT Birth WORDDATE12.;` `PUT Birth WORDDATE18.;`	`Jul 19, 1984` `July 19, 1984`

[3] Formats can be used in PUT statements and PUT functions in DATA steps, and in FORMAT statements in either DATA or PROC steps.

3.9 Using the RETAIN and Sum Statements

When reading raw data, SAS sets the values of all variables equal to missing at the start of each iteration of the DATA step. These values may be changed by INPUT or assignment statements, but they are set back to missing again when SAS returns to the top of the DATA step to process the next observation. RETAIN and sum statements change this. If a variable appears in a RETAIN statement, then its value will be retained from one iteration of the DATA step to the next. A sum statement also retains values from the previous iteration of the DATA step, but then adds to it the value of an expression.

RETAIN statement Use the RETAIN statement when you want SAS to preserve a variable's value from the previous iteration of the DATA step. The RETAIN statement can appear anywhere in the DATA step and has the following form, where all variables to be retained are listed after the RETAIN keyword:

```
RETAIN variable-list;
```

You can also specify an initial value, instead of missing, for the variables. All variables listed before an initial value will start the first iteration of the DATA step with that value:

```
RETAIN variable-list initial-value;
```

Sum statement A sum statement also retains values from the previous iteration of the DATA step, but you use it for the special cases where you simply want to cumulatively add the value of an expression to a variable. A sum statement, like an assignment statement, contains no keywords. It has the following form:

```
variable + expression;
```

No, there is no typo here and no equal sign either. This statement adds the value of the expression to the variable while retaining the variable's value from one iteration of the DATA step to the next. The variable must be numeric and has the initial value of zero. This statement can be re-written using the RETAIN statement and SUM function as follows:

```
RETAIN variable 0;
variable = SUM(variable, expression);
```

As you can see, a sum statement is really a special case of using RETAIN.

Example This example illustrates the use of both the RETAIN and sum statements. The minor league baseball team, the Walla Walla Sweets, has the following data about their games. The date the game was played and the team played are followed by the number of hits and runs for the game:

```
6-19 Columbia Peaches      8  3
6-20 Columbia Peaches     10  5
6-23 Plains Peanuts        3  4
6-24 Plains Peanuts        7  2
6-25 Plains Peanuts       12  8
6-30 Gilroy Garlics        4  4
7-1  Gilroy Garlics        9  4
7-4  Sacramento Tomatoes  15  9
7-4  Sacramento Tomatoes  10 10
7-5  Sacramento Tomatoes   2  3
```


The team wants two additional variables in their data set. One shows the cumulative number of runs for the season, and the other shows the maximum number of runs in a game to date. The following program uses a sum statement to compute the cumulative number of runs, and the RETAIN statement and MAX function to determine the maximum number of runs in a game to date:

```
* Using RETAIN and sum statements to find most runs and total runs;
DATA gamestats;
   INFILE 'c:\MyRawData\Games.dat';
   INPUT Month 1 Day 3-4 Team $ 6-25 Hits 27-28 Runs 30-31;
   RETAIN MaxRuns;
   MaxRuns = MAX(MaxRuns, Runs);
   RunsToDate + Runs;
PROC PRINT DATA = gamestats;
   TITLE "Season's Record to Date";
RUN;
```

The variable MaxRuns is set equal to the maximum of its value from the previous iteration of the DATA step (since it appears in the RETAIN statement) or the value of the variable Runs. The variable RunsToDate adds the number of runs per game, Runs, to itself while retaining its value from one iteration of the DATA step to the next. This produces a cumulative record of the number of runs.

Here are the results:

Season's Record to Date 1

Obs	Month	Day	Team	Hits	Runs	Max Runs	Runs ToDate
1	6	19	Columbia Peaches	8	3	3	3
2	6	20	Columbia Peaches	10	5	5	8
3	6	23	Plains Peanuts	3	4	5	12
4	6	24	Plains Peanuts	7	2	5	14
5	6	25	Plains Peanuts	12	8	8	22
6	6	30	Gilroy Garlics	4	4	8	26
7	7	1	Gilroy Garlics	9	4	8	30
8	7	4	Sacramento Tomatoes	15	9	9	39
9	7	4	Sacramento Tomatoes	10	10	10	49
10	7	5	Sacramento Tomatoes	2	3	10	52

3.10 Simplifying Programs with Arrays

Sometimes you want to do the same thing to many variables. You may want to take the log of every numeric variable or change every occurrence of zero to a missing value. You could write a series of assignment statements or IF statements, but if you have a lot of variables to transform, using arrays will simplify and shorten your program.

An array is an ordered group of similar items. You might think your local mall has a nice array of stores to choose from. In SAS, an array is a group of variables. You can define an array to be any group of variables you like, as long as they are either all numeric or all character. The variables can be ones that already exist in your data set, or they can be new variables that you want to create.

Arrays are defined using the ARRAY statement in the DATA step. The ARRAY statement has the following general form:

```
ARRAY name (n) $ variable-list;
```

In this statement, *name* is a name you give to the array, and *n* is the number of variables in the array. Following the (*n*) is a list of variable names. The number of variables in the list must equal the number given in parentheses. (You may use {} or [ ] instead of parentheses if you like.) This is called an explicit array, where you explicitly state the number of variables in the array. The $ is needed if the variables are character and is only necessary if the variables have not previously been defined.

The array itself is not stored with the data set; it is defined only for the duration of the DATA step. You can give the array any name, as long as it does not match any of the variable names in your data set or any SAS keywords. The rules for naming arrays are the same as those for naming variables (must be 32 characters or fewer and start with a letter or underscore followed by letters, numerals, or underscores).

To reference a variable using the array name, give the array name and the subscript for that variable. The first variable in the variable list has subscript 1, the second has subscript 2, and so forth. So if you have an array defined as

```
ARRAY store (4) Macys Penneys Sears Target;
```

STORE(1) is the variable Macys, STORE(2) is the variable Penneys, STORE(3) is the variable Sears, and STORE(4) is the variable Target. This is all just fine, but simply defining an array doesn't do anything for you. You want to be able to use the array to make things easier for you.

Example The radio station WBRK is conducting a survey asking people to rate ten different songs. Songs are rated on a scale of 1 to 5, where 1 = change the station when it comes on, and 5 = turn up the volume when it comes on. If listeners had not heard the song or didn't care to comment on it, a 9 was entered for that song. The following are the data collected:

```
Albany      54 4 3 5 9 9 2 1 4 4 9
Richmond    33 5 2 4 3 9 2 9 3 3 3
Oakland     27 1 3 2 9 9 9 3 4 2 3
Richmond    41 4 3 5 5 5 2 9 4 5 5
Berkeley    18 3 4 9 1 4 9 3 9 3 2
```


The listener's city of residence, age, and their responses to all ten songs are listed. The following program changes all the 9s to missing values. (The variables are named using the first letters of the words in the song's title.)

```
* Change all 9s to missing values;
DATA songs;
   INFILE 'c:\MyRawData\WBRK.dat';
   INPUT City $ 1-15 Age domk wj hwow simbh kt aomm libm tr filp ttr;
   ARRAY song (10) domk wj hwow simbh kt aomm libm tr filp ttr;
   DO i = 1 TO 10;
      IF song(i) = 9 THEN song(i) = .;
   END;
PROC PRINT DATA = songs;
   TITLE 'WBRK Song Survey';
RUN;
```

An array, SONG, is defined as having ten variables, the same ten variables that appear in the INPUT statement representing the ten songs. Next comes an iterative DO statement. All statements between the DO statement and the END statement are executed, in this case, ten times, once for each variable in the array.

The variable I is used as an index variable and is incremented by 1 each time through the DO loop. The first time through the DO loop, the variable I has a value of 1 and the IF statement would read `IF song(1)=9 THEN song(1)=.;`, which is the same as `IF domk=9 THEN domk=.;`. The second time through, I has a value of 2 and the IF statement would read `IF song(2)=9 THEN song(2)=.;`, which is the same as `IF wj=9 THEN wj=.;`. This continues through all 10 variables in the array.

Here are the results:

WBRK Song Survey 1

Obs	City	Age	domk	wj	hwow	simbh	kt	aomm	libm	tr	filp	ttr	i
1	Albany	54	4	3	5	.	.	2	1	4	4	.	11
2	Richmond	33	5	2	4	3	.	2	.	3	3	3	11
3	Oakland	27	1	3	2	.	.	.	3	4	2	3	11
4	Richmond	41	4	3	5	5	5	2	.	4	5	5	11
5	Berkeley	18	3	4	.	1	4	.	3	.	3	2	11

Notice that the array members SONG(1) to SONG(10) did not become part of the data set, but the variable I did. You could have written ten IF statements instead of using arrays and accomplished the same result. In this program it would not have made a big difference, but if you had 100 songs in your survey instead of ten, then using arrays would clearly be a better solution.

3.11 Using Shortcuts for Lists of Variable Names

As the title states, this section is about shortcuts, shorthand ways of writing lists of variable names. While writing SAS programs, you will often need to write a list of variable names. When defining ARRAYS, using functions like MEAN or SUM, or using SAS procedures, you must specify which variables to use. Now, if you only have a handful of variables, you might not feel a need for a shortcut. But if, for example, you need to define an array with 100 elements, you might be a little grumpy after typing in the 49th variable name knowing you still have 51 more to go. You might even think, "There must be an easier way." Well, there is.

You can use an abbreviated list of variable names anywhere you can use a regular variable list. In functions, abbreviated lists must be preceded by the keyword OF (for example, SUM(OF Cat8 - Cat12)). Otherwise, you simply replace the regular list of variables with the abbreviated one.

Numbered range lists Variables which start with the same characters and end with consecutive numbers can be part of a numbered range list. The numbers can start and end anywhere as long as the number sequence between is complete. For example, the following INPUT statement shows a variable list and its abbreviated form:

Variable list	Abbreviated list
`INPUT Cat8 Cat9 Cat10 Cat11 Cat12;`	`INPUT Cat8 - Cat12;`

Name range lists Name range lists depend on the internal order, or position, of the variables in the SAS data set. This is determined by the order of appearance of the variables in the DATA step. For example, if you had the following DATA step, then the internal variable order would be Y A C H R B:

```
DATA example;
   INPUT y a c h r;
   b = c + r;
RUN;
```

To specify a name range list, put the first variable, then two hyphens, then the last variable. The following PUT statements show the variable list and its abbreviated form using a named range:

Variable list	Abbreviated list
`PUT y a c h r b;`	`PUT y -- b;`

If you are not sure of the internal order, you can find out using PROC CONTENTS with the POSITION option. The following program will list the variables in the permanent SAS data set DISTANCE sorted by position:

```
LIBNAME mydir 'c:\MySASLib';
PROC CONTENTS DATA = mydir.distance POSITION;
RUN;
```

Use caution when including name range lists in your programs. Although they can save on typing, they may also make your programs more difficult to understand and debug.

Special SAS name lists The special name lists, _ALL_, _CHARACTER_, and _NUMERIC_ can also be used any place you want either all the variables, all the character variables, or all the numeric variables in a SAS data set. These name lists are useful when you want to do something like compute the mean of all the numeric variables for an observation (MEAN(OF _NUMERIC_)), or list the values of all variables in an observation (PUT _ALL_;).

Example The radio station WBRK wants to modify the program from the previous section, which changes all 9s to missing values. Now, instead of changing the original variables, they use the following program to create new variables (Song1 through Song10) which will have the new missing values. This program also computes the average score using the MEAN function.

```
DATA songs;
   INFILE 'c:\MyRawData\WBRK.dat';
   INPUT City $ 1-15 Age domk wj hwow simbh kt aomm libm tr filp ttr;
   ARRAY new (10) Song1 - Song10;
   ARRAY old (10) domk - ttr;
   DO i = 1 TO 10;
      IF old(i) = 9 THEN new(i) = .;
         ELSE new(i) = old(i);
   END;
   AvgScore = MEAN(OF Song1 - Song10);
PROC PRINT DATA = songs;
   TITLE 'WBRK Song Survey';
RUN;
```

Note that both ARRAY statements use abbreviated variable lists; array NEW uses a numbered range list and array OLD uses a name range list. Inside the iterative DO loop, the Song variables (array NEW) are set equal to missing if the original variable (array OLD) had a value of 9. Otherwise, they are set equal to the original values. After the DO loop, a new variable, AvgScore, is created using an abbreviated variable list in the function MEAN. The output includes variables from both the OLD array (domk, wj, ... ttr) and NEW array (Song1 - Song10):

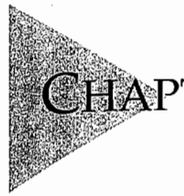
WBRK Song Survey 1

O	City	Age	domk	wj	hwow	simbh	kt	aomm	libm	tr	filp	ttr	Song1	Song2	Song3	Song4	Song5	Song6	Song7	Song8	Song9	Song10	i	AvgScore
1	Albany	54	4	3	5	9	9	2	1	4	4	9	4	3	5	.	.	2	1	4	4	.	11	3.28571
2	Richmond	33	5	2	4	3	9	2	9	3	3	3	5	2	4	3	.	2	.	3	3	3	11	3.12500
3	Oakland	27	1	3	2	9	9	9	3	4	2	3	1	3	2	.	.	.	3	4	2	3	11	2.57143
4	Richmond	41	4	3	5	5	5	2	9	4	5	5	4	3	5	5	5	2	.	4	5	5	11	4.22222
5	Berkeley	18	3	4	9	1	4	9	3	9	3	2	3	4	.	1	4	.	3	.	3	2	11	2.85714

"Once in a while the simple things work right off."

PHIL GALLAGHER

From the SAS L Listserve, 1994. Reprinted by permission of the author.

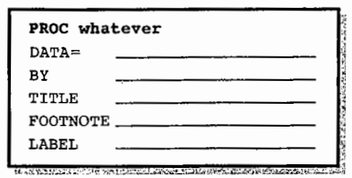


CHAPTER 4

Sorting, Printing, and Summarizing Your Data

4.1 Using SAS Procedures

Using a procedure, or PROC, is like filling out a form. Someone else designed the form, and all you have to do is fill in the blanks and choose from a list of options. Each PROC has its own unique form with its own list of options. But while each procedure is unique, there are similarities too. This section discusses some of those similarities.



All procedures have required statements, and most have optional statements. PROC PRINT, for example, requires only two words:

```
PROC PRINT;
```

However, by adding optional statements you could make this procedure a dozen lines or even longer.

PROC statement All procedures start with the keyword PROC followed by the name of the procedure, such as PRINT or CONTENTS. Options, if there are any, follow the procedure name. The DATA= option tells SAS which data set to use as input for that procedure. In this case, SAS will use a temporary SAS data set named BANANA:

```
PROC CONTENTS DATA = banana;
```

The DATA= option is, of course, optional. If you skip it, then SAS will use the most recently created data set, which is not necessarily the same as the most recently used. Sometimes it is easier to specify the data set you want than to figure out which data set SAS will use by default. To use a permanent SAS data set, issue a LIBNAME statement to set up a libref pointing to the location of your data set, and put the data set's two-level name in the DATA= option, as discussed in section 2.20,

```
LIBNAME tropical 'c:\MySASLib';
PROC CONTENTS DATA = tropical.banana;
```

or refer to it directly by placing your operating environment's name for the permanent SAS data set between quotation marks, as discussed in section 2.21.

```
PROC CONTENTS DATA = 'c:\MySASLib\banana';
```

BY statement The BY statement is required for only one procedure, PROC SORT. In PROC SORT the BY statement tells SAS how to arrange the observations. In all other procedures, the BY statement is optional, and tells SAS to perform a separate analysis for each combination of values of the BY variables rather than treating all observations as one group. For example, this statement tells SAS to run a separate analysis for each state:

```
BY State;
```

All procedures, except PROC SORT, assume that your data are already sorted by the variables in your BY statement. If your observations are not already sorted, then use PROC SORT to do the job.

TITLE and FOOTNOTE statements You have seen TITLE statements many times in this book. FOOTNOTE works the same way, but prints at the bottom of the page. These global statements are not technically part of any step. You can put them anywhere in your program, but since they apply to the procedure output it generally makes sense to put them with the procedure.

The most basic TITLE statement consists of the keyword TITLE followed by your title enclosed in quotation marks. SAS doesn't care if the two quotation marks are single or double as long as they are the same:

```
TITLE 'This is a title';
```

If you find that your title contains an apostrophe, use double quotation marks around the title, or replace the single apostrophe with two:

```
TITLE "Here's another title";
TITLE 'Here''s another title';
```

You can specify up to ten titles or footnotes by adding numbers to the keywords TITLE and FOOTNOTE:

```
FOOTNOTE3 'This is the third footnote';
```

Titles and footnotes stay in effect until you replace them with new ones or cancel them with a null statement. The following null statement would cancel all current titles:

```
TITLE;
```

When you specify a new title or footnote, it replaces the old title or footnote with the same number and cancels those with a higher number. For example, a new TITLE2 cancels an existing TITLE3, if there is one.

LABEL statement By default, SAS uses variable names to label your output, but with the LABEL statement you can create more descriptive labels, up to 256 characters long, for each variable. This statement creates labels for the variables ReceiveDate and ShipDate:

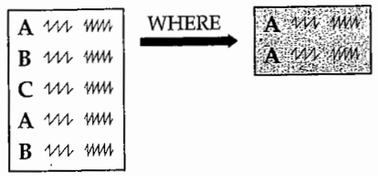
```
LABEL ReceiveDate  = 'Date order was received'
      ShipDate = 'Date merchandise was shipped';
```

When a LABEL statement is used in a DATA step, the labels become part of the data set; but when used in a PROC, the labels stay in effect only for the duration of that step.

Customizing output You have a lot of control over the output produced by procedures. Using system options, you can set many features such as centering, dates, line size, and page size (section 1.13). With the Output Delivery System, you can you can also change the overall style of your output, produce output in different formats (such as HTML or RTF), or change almost any detail of your output (section 1.10 and chapter 5).

Output data sets Most procedures produce some kind of report, but sometimes you would like the results of the procedure saved as a SAS data set so you can perform further analysis. You can create SAS data sets from any procedure output using the ODS OUTPUT statement (section 5.3). Some procedures can also write a SAS data set using an OUTPUT statement or OUT= option.

4.2 Subsetting in Procedures with the WHERE Statement



One optional statement for any PROC that reads a SAS data set is the WHERE statement. The WHERE statement tells a procedure to use a subset of the data. There are other ways to subset data, as you probably remember, so you could get by without ever using the WHERE statement.[1] However, the WHERE statement is a shortcut. While the other methods of subsetting work only in DATA steps, the WHERE statement works in PROC steps too.

Unlike subsetting in a DATA step, using a WHERE statement in a procedure does not create a new data set. That is one of the reasons why WHERE statements are sometimes more efficient than other ways of subsetting.

The basic form of a WHERE statement is

```
WHERE condition;
```

Only observations satisfying the condition will be used by the PROC. This may look familiar since it is similar to a subsetting IF. The left side of that condition is a variable name, and the right side is a variable name, a constant, or a mathematical expression. Mathematical expressions can contain the standard arithmetic symbols for addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (**). Between the two sides of the expression, you can use comparison and logical operators; those operators may be symbolic or mnemonic. Here are the most frequently used operators:

Symbolic	Mnemonic	Example
=	EQ	`WHERE Region = 'Spain';`
¬=, ~=, ^=	NE	`WHERE Region ~= 'Spain';`
>	GT	`WHERE Rainfall > 20;`
<	LT	`WHERE Rainfall < AvgRain;`
>=	GE	`WHERE Rainfall >= AvgRain + 5;`
<=	LE	`WHERE Rainfall <= AvgRain / 1.25;`
&	AND	`WHERE Rainfall > 20 AND Temp < 90;`
|,!,!	OR	`WHERE Rainfall > 20 OR Temp < 90;`
	IS NOT MISSING	`WHERE Region IS NOT MISSING;`
	BETWEEN AND	`WHERE Region BETWEEN 'Plain' AND 'Spain';`
	CONTAINS	`WHERE Region CONTAINS 'ain';`
	IN (*list*)	`WHERE Region IN ('Rain', 'Spain', 'Plain');`

[1] Subsetting while reading a raw data file is discussed in section 2.13, and the subsetting IF statement is discussed in section 3.6.

Example You have a database containing information about well-known painters. A subset of the data appears below. For each artist, the data include the painter's name, primary style, and nation of origin:

```
Mary Cassatt           Impressionism      U
Paul Cezanne           Post-impressionism F
Edgar Degas            Impressionism      F
Paul Gauguin           Post-impressionism F
Claude Monet           Impressionism      F
Pierre Auguste Renoir  Impressionism      F
Vincent van Gogh       Post-impressionism N
```

To make this example more realistic, it has two steps: one to create a permanent SAS data set, the other to subset the data. The first DATA step reads the data from a file named Artists.dat, and uses direct referencing (you could use a LIBNAME statement instead) to create a permanent SAS data set named STYLE in a directory named MySASLib (Windows).

```
DATA 'c:\MySASLib\style';
   INFILE 'c:\MyRawData\Artists.dat';
   INPUT Name $ 1-21 Genre $ 23-40 Origin $ 42;
RUN;
```

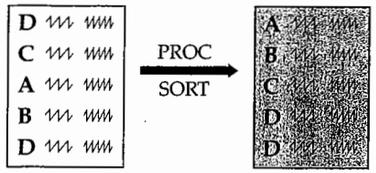
Suppose a day later you wanted to print a list of just the impressionist painters. The quick-and-easy way to do this is with a WHERE statement and PROC PRINT. The quotation marks around the data set name tell SAS that this is a permanent SAS data set.

```
PROC PRINT DATA = 'c:\MySASLib\style';
   WHERE Genre = 'Impressionism';
   TITLE 'Major Impressionist Painters';
   FOOTNOTE 'F = France N = Netherlands U = US';
RUN;
```

The output looks like this:

```
                Major Impressionist Painters                1

      Obs    Name                    Genre           Origin

       1     Mary Cassatt            Impressionism   U
       3     Edgar Degas             Impressionism   F
       5     Claude Monet            Impressionism   F
       6     Pierre Auguste Renoir   Impressionism   F

            F = France N = Netherlands U = US
```

4.3 Sorting Your Data with PROC SORT



There are many reasons for sorting your data: to organize data for a report, before combining data sets, or before using a BY statement in another PROC or DATA step. Fortunately, PROC SORT is quite simple. The basic form of this procedure is

```
PROC SORT;
   BY variable-1 ... variable-n;
```

The variables named in the BY statement are called BY variables. You can specify as many BY variables as you wish. With one BY variable, SAS sorts the data based on the values of that variable. With more than one variable, SAS sorts observations by the first variable, then by the second variable within categories of the first, and so on. A BY group is all the observations that have the same values of BY variables. If, for example, your BY variable is State then all the observations for North Dakota form one BY group.

The DATA= and OUT= options specify the input and output data sets. If you don't specify the DATA= option, then SAS will use the most recently created data set. If you don't specify the OUT= option, then SAS will replace the original data set with the newly sorted version. This sample statement tells SAS to sort the data set named MESSY, and then put the sorted data into a data set named NEAT:

```
PROC SORT DATA = messy OUT = neat;
```

The NODUPKEY option tells SAS to eliminate any duplicate observations that have the same values for the BY variables. To use this option, just add NODUPKEY to the PROC SORT statement:

```
PROC SORT DATA = messy OUT = neat NODUPKEY;
```

By default SAS sorts data in ascending order, from lowest to highest or from A to Z. To have your data sorted from highest to lowest, add the keyword DESCENDING to the BY statement before each variable that should be sorted from highest to lowest. This statement tells SAS to sort first by State (from A to Z) and then by City (from Z to A) within State:

```
BY State DESCENDING City;
```

Example The following data show the average length in feet of selected whales and sharks:

```
beluga   whale 15
whale    shark 40
basking  shark 30
gray     whale 50
mako     shark 12
sperm    whale 60
dwarf    shark .5
whale    shark 40
humpback   .   50
blue     whale 100
killer   whale 30
```


This program reads and sorts the data:

```
DATA marine;
   INFILE 'c:\MyRawData\Sealife.dat';
   INPUT Name $ Family $ Length;
* Sort the data;
PROC SORT DATA = marine OUT = seasort NODUPKEY;
   BY Family DESCENDING Length;
PROC PRINT DATA = seasort;
   TITLE 'Whales and Sharks';
RUN;
```

The DATA step reads the raw data from a file called Sealife.dat and creates a SAS data set named MARINE. Then PROC SORT rearranges the observations by family in ascending order, and by length in descending order. The NODUPKEY option of PROC SORT eliminates any duplicates, while the OUT= option writes the sorted data into a new data set named SEASORT. The output from PROC PRINT looks like this:

```
                    Whales and Sharks          1

 Obs   Name         Family    Length

  1    humpback                50.0
  2    whale        shark      40.0
  3    basking      shark      30.0
  4    mako         shark      12.0
  5    dwarf        shark       0.5
  6    blue         whale     100.0
  7    sperm        whale      60.0
  8    gray         whale      50.0
  9    killer       whale      30.0
 10    beluga       whale      15.0
```

Notice that the humpback with a missing value for Family became observation one. That is because missing values are always low for both numeric and character variables. Also, the NODUPKEY option eliminated a duplicate observation for the whale shark. The log contains these notes showing that the sorted data set has one fewer observation than the original data set.

```
NOTE: The data set WORK.MARINE has 11 observations and 3 variables.

NOTE: 1 observations with duplicate key values were deleted.
NOTE: The data set WORK.SEASORT has 10 observations and 3 variables.
```

4.4 Printing Your Data with PROC PRINT

The PRINT procedure is perhaps the most widely used SAS procedure. You have seen this procedure used many times in this book to print the contents of a SAS data set. In its simplest form, PROC PRINT prints all variables for all observations in the SAS data set. SAS decides the best way to format the output, so you don't have to worry about things like how many variables will fit on a page. But there are a few more features of PROC PRINT that you might want to use.

The PRINT procedure requires just one statement:

```
PROC PRINT;
```

By default, SAS uses the SAS data set created most recently. If you do not want to print the most recent data set, then use the DATA= option to specify the data set. We recommend always using the DATA= option for clarity in your programs as it is not always easy to quickly determine which data set was created last.

```
PROC PRINT DATA = data-set;
```

Also, SAS prints the observation numbers along with the variables' values. If you don't want observation numbers, use the NOOBS option in the PROC PRINT statement. If you define variable labels with a LABEL statement, and you want to print the labels instead of the variable names, then add the LABEL option as well. The following statement shows all of these options together:

```
PROC PRINT DATA = data-set NOOBS LABEL;
```

The following are optional statements that sometimes come in handy:

`BY variable-list;`	The BY statement starts a new section in the output for each new value of the BY variables and prints the values of the BY variables at the top of each section. The data must be presorted by the BY variables.
`ID variable-list;`	When you use the ID statement, the observation numbers are not printed. Instead, the variables in the ID variable list appear on the left-hand side of the page.
`SUM variable-list;`	The SUM statement prints sums for the variables in the list.
`VAR variable-list;`	The VAR statement specifies which variables to print and the order. Without a VAR statement, all variables in the SAS data set are printed in the order that they occur in the data set.

Example Students from two fourth-grade classes are selling candy to earn money for a special field trip. The class earning more money gets a free box of candy. The following are the data for the results of the candy sale. The students' names are followed by their classroom number, the date they turned in their money, the type of candy: mint patties or chocolate dinosaurs, and the number of boxes sold:


```
Adriana    21  3/21/2000 MP  7
Nathan     14  3/21/2000 CD 19
Matthew    14  3/21/2000 CD 14
Claire     14  3/22/2000 CD 11
Caitlin    21  3/24/2000 CD  9
Ian        21  3/24/2000 MP 18
Chris      14  3/25/2000 CD  6
Anthony    21  3/25/2000 MP 13
Stephen    14  3/25/2000 CD 10
Erika      21  3/25/2000 MP 17
```

The class earns $1.25 for each box of candy sold. The teachers want a report giving the money earned for each classroom, the money earned by each student, the type of candy sold, and the date the students returned their money. The following program reads the data, computes money earned (Profit), and sorts the data by classroom using PROC SORT. Then, the PROC PRINT step uses a BY statement to print the data by Class and a SUM statement to give the totals for Profit. The VAR statement lists the variables to be printed:

```
DATA sales;
   INFILE 'c:\MyRawData\Candy.dat';
   INPUT Name $ 1-11 Class @15 DateReturned MMDDYY10. CandyType $
      Quantity;
   Profit = Quantity * 1.25;
PROC SORT DATA = sales;
   BY Class;
PROC PRINT DATA = sales;
   BY Class;
   SUM Profit;
   VAR Name DateReturned CandyType Profit;
   TITLE 'Candy Sales for Field Trip by Class';
RUN;
```

Here are the results. Notice that the values for the variable DateReturned are printed as their SAS date values. You can use formats, covered in section 4.5, to print dates in readable forms.

```
                Candy Sales for Field Trip by Class                    1
------------------------------ Class=14 -------------------------------

                                   Date      Candy
              Obs    Name        Returned    Type      Profit

                1    Nathan         14690     CD         23.75
                2    Matthew        14690     CD         17.50
                3    Claire         14691     CD         13.75
                4    Chris          14694     CD          7.50
                5    Stephen        14694     CD         12.50
             -----                                      ------
             Class                                      75.00

------------------------------ Class=21 -------------------------------

                                   Date      Candy
              Obs    Name        Returned    Type      Profit

                6    Adriana        14690     MP          8.75
                7    Caitlin        14693     CD         11.25
                8    Ian            14693     MP         22.50
                9    Anthony        14694     MP         16.25
               10    Erika          14694     MP         21.25
             -----                                      ------
             Class                                      80.00
                                                        ======
                                                       155.00
```

4.5 Changing the Appearance of Printed Values with Formats

0	1002
2	2012
31	4336

→

Obs	Date	Sales
1	01/01/60	1,002
2	01/03/60	2,012
3	02/01/60	4,336

When SAS prints your data, it decides which format is best—how many decimal places to print, how much space to allow for each value, and so on. This is very convenient and makes your job much easier, but SAS doesn't always do what you want. Fortunately you're not stuck with the format SAS thinks is best. You can change the appearance of printed values using SAS formats.

SAS has many formats for character, numeric, and date values. For example, you can use the COMMA*w.d* format to print numbers with embedded commas, the $*w*. format to control the number of characters printed, and the MMDDYY*w*. format to print SAS date values (the number of days since January 1, 1960) in a readable form like 12/03/2003. You can even print your data in more obscure formats like hexadecimal, zoned decimal, and packed decimal, if you like.[1]

The general forms of a SAS format are

Character	Numeric	Date
`$formatw.`	`formatw.d`	`formatw.`

where the $ indicates character formats, *format* is the name of the format, *w* is the total width including any decimal point, and *d* is the number of decimal places. The period in the format is very important because it distinguishes a format from a variable name, which cannot, by default, contain any special characters except the underscore.

FORMAT statement You can associate formats with variables in a FORMAT statement. The FORMAT statement starts with the keyword FORMAT, followed by the variable name (or names if more than one variable is to be associated with the same format), followed by the format. For example, the following FORMAT statement associates the DOLLAR8.2 format with the variables Profit and Loss and associates the MMDDYY8. format with the variable SaleDate:

```
FORMAT Profit Loss DOLLAR8.2 SaleDate MMDDYY8.;
```

FORMAT statements can go in either DATA steps or PROC steps. If the FORMAT statement is in a DATA step, then the format association is permanent and is stored with the SAS data set. If the FORMAT statement is in a PROC step, then it is temporary—affecting only the results from that procedure.

PUT statement You can also use formats in PUT statements when writing raw data files or reports. Place a format after each variable name, as in the following example:

```
PUT Profit DOLLAR8.2 Loss DOLLAR8.2 SaleDate MMDDYY8.;
```

Example In section 4.4, results from the fourth-grade candy sale were printed using the PRINT procedure. The names of the students were printed along with the date they turned in their money, the type of candy sold, and the profit. You may have noticed that the dates printed

[1] You can also create your own formats using the FORMAT procedure covered in section 4.7.

were numbers like 14690 and 14694. Using the FORMAT statement in the PRINT procedure, we can print the dates in a readable form. At the same time, we can print the variable Profit using the DOLLAR6.2 format so dollar signs appear before the numbers.

Here are the data, where the students' names are followed by their classroom, the date they turned in their money, the type of candy sold: mint patties or chocolate dinosaurs, and the number of boxes sold:

```
Adriana     21  3/21/2000 MP  7
Nathan      14  3/21/2000 CD 19
Matthew     14  3/21/2000 CD 14
Claire      14  3/22/2000 CD 11
Caitlin     21  3/24/2000 CD  9
Ian         21  3/24/2000 MP 18
Chris       14  3/25/2000 CD  6
Anthony     21  3/25/2000 MP 13
Stephen     14  3/25/2000 CD 10
Erika       21  3/25/2000 MP 17
```

The following program reads the raw data and computes Profit. The FORMAT statement in the PRINT procedure associates the DATE9. format with the variable DateReturned and the DOLLAR6.2 format with the variable Profit:

```
DATA sales;
   INFILE 'c:\MyRawData\Candy.dat';
   INPUT Name $ 1-11 Class @15 DateReturned MMDDYY10. CandyType $
         Quantity;
   Profit = Quantity * 1.25;
PROC PRINT DATA = sales;
   VAR Name DateReturned CandyType Profit;
   FORMAT DateReturned DATE9. Profit DOLLAR6.2;
   TITLE 'Candy Sale Data Using Formats';
RUN;
```

Here are the results:

Candy Sale Data Using Formats 1

Obs	Name	Date Returned	Candy Type	Profit
1	Adriana	21MAR2000	MP	$8.75
2	Nathan	21MAR2000	CD	$23.75
3	Matthew	21MAR2000	CD	$17.50
4	Claire	22MAR2000	CD	$13.75
5	Caitlin	24MAR2000	CD	$11.25
6	Ian	24MAR2000	MP	$22.50
7	Chris	25MAR2000	CD	$7.50
8	Anthony	25MAR2000	MP	$16.25
9	Stephen	25MAR2000	CD	$12.50
10	Erika	25MAR2000	MP	$21.25

4.6 Selected Standard Formats

Here are definitions of commonly used formats[1] along with the width range and default width.

Format	Definition	Width range	Default width
Character			
$HEX*w*.	Converts character data to hexidecimal (specify *w* twice the length of the variable)	1-32767	4
$*w*.	Writes standard character data—does not trim leading blanks (same as $CHAR*w*.)	1-32767	Length of variable or 1
Date, Time, and Datetime			
DATE*w*.	Writes SAS date values in form *ddmmmyy* or *ddmmmyyyy*	5-9	7
DATETIME*w.d*	Writes SAS datetime values in form *ddmmmyy:hh:mm:ss.ss*	7-40	16
DAY*w*.	Writes day of month from a SAS date value	2-32	2
EURDFDD*w*.	Writes a SAS date value in form: *dd.mm.yy*	2-10	8
JULIAN*w*.	Writes a Julian date from a SAS date value in form *yyddd* or *yyyyddd*	5-7	5
MMDDYY*w*.	Writes SAS date values in form *mmddyy* or *mmddyyyy*	2-10	8
TIME*w.d*	Writes SAS time values in form *hh:mm:ss.ss*	2-20	8
WEEKDATE*w*.	Writes SAS date values in form *day-of-week, month-name dd, yy* or *yyyy*	3-37	29
WORDDATE*w*.	Writes SAS date values in form *month-name dd, yyyy*	3-32	18
Numeric			
BEST*w*.	SAS chooses best format—this is the default format for writing numeric data	1-32	12
COMMA*w.d*	Writes numbers with commas separating every three digits	2-32	6
DOLLAR*w.d*	Writes numbers with a leading $ and commas separating every three digits	2-32	6
E*w*.	Writes numbers in scientific notation	7-32	12
PD*w.d*	Writes numbers in packed decimal—*w* specifies the number of bytes	1-16	1
w.d	Writes standard numeric data	1-32	none

[1] Check your SAS Help and Documentation for a complete list of formats.

[2] SAS date values are the number of days since January 1, 1960. SAS time values are the number of seconds past midnight, and datetime values are the number of seconds since midnight January 1, 1960.

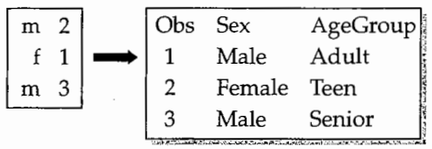
Here are examples using the selected formats.

Format	Input data	PUT statement	Results
Character			
$HEX*w*.	AB	`PUT Name $HEX4.;`	C1C2 (EBCDIC)[3] 4142 (ASCII)
$*w*.	my cat my snake	`PUT Animal $8. '*';`	my cat * my snak*
Date, Time, and Datetime			
DATE*w*.	8966	`PUT Birth DATE7.;` `PUT Birth DATE9.;`	19JUL84 19JUL1984
DATETIME*w*.	12182	`PUT Start DATETIME13.;` `PUT Start DATETIME18.1;`	01JAN60:03:23 01JAN60:03:23:02.0
DAY*w*.	8966	`PUT Birth DAY2.;` `PUT Birth DAY7.;`	19 19
EURDFDD*w*.	8966	`PUT Birth EURDFDD8.;`	19.07.84
JULIAN*w*.	8966	`PUT Birth JULIAN5.;` `PUT Birth JULIAN7.;`	84201 1984201
MMDDYY*w*.	8966	`PUT Birth MMDDYY8.;` `PUT Birth MMDDYY6.;`	7/19/84 071984
TIME*w.d*	12182	`PUT Start TIME8.;` `PUT Start TIME11.2;`	3:23:02 3:23:02.00
WEEKDATE*w*.	8966	`PUT Birth WEEKDATE15.;` `PUT Birth WEEKDATE29.;`	Thu, Jul 19, 84 Thursday, July 19, 1984
WORDDATE*w*.	8966	`PUT Birth WORDDATE12.;` `PUT Birth WORDDATE18.;`	Jul 19, 1984 July 19, 1984
Numeric			
BEST*w*.	1200001	`PUT Value BEST6.;` `PUT Value BEST8.;`	1.20E6 1200001
COMMA*w.d*	1200001	`PUT Value COMMA9.;` `PUT Value COMMA12.2;`	1,200,001 1,200,001.00
DOLLAR*w.d*	1200001	`PUT Value DOLLAR10.;` `PUT Value DOLLAR13.2;`	$1,200,001 $1,200,001.00
E*w*.	1200001	`PUT Value E7.;`	1.2E+06
PD*w.d*	128	`PUT Value PD4.;`	[4]
w.d	23.635	`PUT Value 6.3;` `PUT Value 5.2;`	23.635 23.64

[3] The EBCDIC character set is used on most IBM mainframe computers while the ASCII character set is used on most other computers. So, depending on the computer you are using, you will get one or the other.

[4] These values cannot be printed.

4.7 Creating Your Own Formats Using PROC FORMAT



m	2
f	1
m	3

Obs	Sex	AgeGroup
1	Male	Adult
2	Female	Teen
3	Male	Senior

At some time you will probably want to create your own custom formats—especially if you use a lot of coded data. Imagine that you have just completed a survey for your company and to save disk space and time, all the responses to the survey questions are coded. For example, the age categories teen, adult, and senior are coded as numbers 1, 2, and 3. This is convenient for data entry and analysis but bothersome when it comes time to interpret the results. You could present your results along with a code book, and your company directors could look up the codes as they read the results. But this will probably not get you that promotion you've been looking for. A better solution is to create user-defined formats using PROC FORMAT and print the formatted values instead of the coded values.

The FORMAT procedure creates formats that will later be associated with variables in a FORMAT statement. The procedure starts with the statement PROC FORMAT and continues with one or more VALUE statements (other optional statements are available):

```
PROC FORMAT;
     VALUE name range-1 = 'formatted-text-1'
                range-2 = 'formatted-text-2'
                        .
                        .
                        .
                range-n = 'formatted-text-n';
```

The *name* in the VALUE statement is the name of the format you are creating. If the format is for character data, the *name* must start with a $. The *name* can't be longer than 32 characters (including the $ for character data), it must not start or end with a number, and cannot contain any special characters except the underscore. In addition, the *name* can't be the name of an existing format. Each *range* is the value of a variable that is assigned to the text given in quotation marks on the right side of the equal sign. The text can be up to 32,767 characters long, but some procedures print only the first 8 or 16 characters. The following are examples of valid range specifications:

```
          'A' = 'Asia'
1, 3, 5, 7, 9 = 'Odd'
500000 - HIGH = 'Not Affordable'
     13 -< 20 = 'Teenager'
    0 <- HIGH = 'Positive Non Zero'
        OTHER = 'Bad Data'
```

Character values must be enclosed in quotation marks ('A' for example). If there is more than one value in the range, then separate the values with a comma or use the hyphen (-) for a continuous range. The keywords LOW and HIGH can be used in ranges to indicate the lowest and the highest non-missing value for the variable. You can also use the less than symbol (<) in ranges to exclude either end point of the range. The OTHER keyword can be used to assign a format to any values not listed in the VALUE statement.

Example Universe Cars is surveying its customers as to their preferences for car colors. They have information about the customer's age, sex (coded as 1 for male and 2 for female), annual income, and preferred car color (yellow, gray, blue, or white). Here are the data:

```
19 1 14000 Y
45 1 65000 G
72 2 35000 B
31 1 44000 Y
58 2 83000 W
```

The following program reads the data; creates formats for age, sex, and car color using the FORMAT procedure; then prints the data using the new formats:

```
DATA carsurvey;
   INFILE 'c:\MyRawData\Cars.dat';
   INPUT Age Sex Income Color $;
PROC FORMAT;
   VALUE gender 1 = 'Male'
                2 = 'Female';
   VALUE agegroup 13 -< 20 = 'Teen'
                  20 -< 65 = 'Adult'
                  65 - HIGH = 'Senior';
   VALUE $col  'W' = 'Moon White'
               'B' = 'Sky Blue'
               'Y' = 'Sunburst Yellow'
               'G' = 'Rain Cloud Gray';
* Print data using user-defined and standard (DOLLAR8.) formats;
PROC PRINT DATA = carsurvey;
   FORMAT Sex gender. Age agegroup. Color $col. Income DOLLAR8.;
   TITLE 'Survey Results Printed with User-Defined Formats';
RUN;
```

This program creates two numeric formats: GENDER for the variable Sex and AGEGROUP for the variable Age. The program creates a character format, $COL, for the variable Color. Notice that the format names do not end with periods in the VALUE statement, but they do in the FORMAT statement.

Here is the output:

Survey Results Printed with User-Defined Formats 1

Obs	Age	Sex	Income	Color
1	Teen	Male	$14,000	Sunburst Yellow
2	Adult	Male	$65,000	Rain Cloud Gray
3	Senior	Female	$35,000	Sky Blue
4	Adult	Male	$44,000	Sunburst Yellow
5	Adult	Female	$83,000	Moon White

This example creates temporary formats that exist only for the current job or session. Creating and using permanent formats is discussed under the FORMAT Procedure in the SAS Help and Documentation.

4.8 Writing Simple Custom Reports

PROC PRINT is flexible and easy to use. Still, there are times when PROC PRINT just won't do: when your report to a state agency has to be spaced just like their fill-in-the-blank form, or when your client insists that the report contain complete sentences, or when you want one page per observation. At those times you can use the flexibility of the DATA step, and format to your heart's content.

You can write data in a DATA step the same way you read data—but in reverse. Instead of using an INFILE statement, you use a FILE statement; instead of INPUT statements, you use PUT statements. This is similar to writing a raw data file in a DATA step (section 9.5), but to write a report you use the PRINT option telling SAS to include the carriage returns and page breaks needed for printing. Here is the general form of a FILE statement for creating a report:

```
FILE 'file-specification' PRINT;
```

Like INPUT statements, PUT statements can be in list, column, or formatted style, but since SAS already knows whether a variable is numeric or character, you don't have to put a $ after character variables. If you use list format, SAS will automatically put a space between each variable. If you use column or formatted styles of PUT statements, SAS will put the variables wherever you specify. You can control spacing with the same pointer controls that INPUT statements use: *@n* to move to column *n*, *+n* to move *n* columns, / to skip to the next line, *#n* to skip to line *n*, and the trailing @ to hold the current line. In addition to printing variables, you can insert a text string by simply enclosing it in quotation marks.

Example To show how this differs from PROC PRINT, we'll use the candy sales data again. Two fourth-grade classes have sold candy to raise money for a field trip. Here are the data with each student's name, classroom number, the date they turned in their money, the type of candy: mint patties or chocolate dinosaurs, and the number of boxes sold:

```
Adriana     21 3/21/2000 MP  7
Nathan      14 3/21/2000 CD 19
Matthew     14 3/21/2000 CD 14
Claire      14 3/22/2000 CD 11
Caitlin     21 3/24/2000 CD  9
Ian         21 3/24/2000 MP 18
Chris       14 3/25/2000 CD  6
Anthony     21 3/25/2000 MP 13
Stephen     14 3/25/2000 CD 10
Erika       21 3/25/2000 MP 17
```

The teachers want a report for each student showing how much money that student earned. They want each student's report on a separate page so it is easy to hand out. Lastly, they want it to be easy for fourth graders to understand, with complete sentences. Here is the program:

```
* Write a report with FILE and PUT statements;
DATA _NULL_;
   INFILE 'c:\MyRawData\Candy.dat';
   INPUT Name $ 1-11 Class @15 DateReturned MMDDYY10.
         CandyType $ Quantity;
   Profit = Quantity * 1.25;
   FILE 'c:\MyRawData\Student.rep' PRINT;
   TITLE;
   PUT @5 'Candy sales report for ' Name 'from classroom ' Class /
     / @5 'Congratulations!  You sold ' Quantity 'boxes of candy'
     / @5 'and earned  ' Profit DOLLAR6.2 ' for our field trip.' /;
   PUT _PAGE_;
RUN;
```


Notice that the keyword _NULL_ appears in the DATA statement instead of a data set name. _NULL_ tells SAS not to bother writing a SAS data set (since the goal is to create a report not a data set), and makes the program run slightly faster. The FILE statement creates the output file for the report, and the PRINT option tells SAS to include carriage returns and page breaks. The null TITLE statement tells SAS to eliminate all automatic titles.

The first PUT statement in this program starts with a pointer, @5, telling SAS to go to column 5. Then it tells SAS to print the words `Candy sales report for` followed by the current value of the variable Name. The variables Name, Class, and Quantity are printed in list style whereas Profit is printed using formatted style and the DOLLAR6.2 format. A slash line pointer tells SAS to skip to the next line; two slashes skips two lines. You could use multiple PUT statements instead of slashes to skip lines because SAS goes to a new line every time there is a new PUT statement. The statement PUT _PAGE_ inserts a page break after each student's report. When the program is run, the log will contain these notes:

```
NOTE: 10 records were read from the infile 'c:\MyRawData\Candy.dat'.

NOTE: 30 records were written to the file 'c:\MyRawData\Student.rep'.
```

The first three pages of the report look like this:

```
Candy sales report for Adriana from classroom 21

Congratulations!  You sold 7 boxes of candy
and earned  $8.75 for our field trip.
```

```
Candy sales report for Nathan from classroom 14

Congratulations!  You sold 19 boxes of candy
and earned $23.75 for our field trip.
```

```
Candy sales report for Matthew from classroom 14

Congratulations!  You sold 14 boxes of candy
and earned $17.50 for our field trip.
```

4.9 Summarizing Your Data Using PROC MEANS

One of the first things people usually want to do with their data, after reading it and making sure it is correct, is look at some simple statistics. Statistics such as the mean value, standard deviation, and minimum and maximum values give you a feel for your data. These types of information can also alert you to errors in your data (a score of 980 in a basketball game, for example, is suspect). The MEANS procedure provides simple statistics on numeric variables.

The MEANS procedure starts with the keywords PROC MEANS, followed by options listing the statistics you want printed:

```
PROC MEANS options;
```

If you do not specify any options, MEANS will print the number of non-missing values, the mean, the standard deviation, and the minimum and maximum values for each variable. There are over 30 different statistics you can request with the MEANS procedure. The following is a list of some of the simple statistics. More options for the MEANS procedure are discussed in section 8.2.

MAX	the maximum value
MIN	the minimum value
MEAN	the mean
MEDIAN	the median
N	number of non-missing values
NMISS	number of missing values
RANGE	the range
STDDEV	the standard deviation
SUM	the sum

If you use the PROC MEANS statement with no other statements, then you will get statistics for all observations and all numeric variables in your data set. Here are some of the optional statements you may want to use:

`BY variable-list;`	The BY statement performs separate analyses for each level of the variables in the list.[1] The data must first be sorted in the same order as the *variable-list*. (You can use PROC SORT to do this.)
`CLASS variable-list;`	The CLASS statement also performs separate analyses for each level of the variables in the list,[1] but its output is more compact than with the BY statement, and the data do not have to be sorted first.
`VAR variable-list;`	The VAR statement specifies which numeric variables to use in the analysis. If it is absent then SAS uses all numeric variables.

[1] By default, observations are excluded if they have missing values for BY or CLASS variables. If you want to include missing values, add the MISSING option to the PROC MEANS statement.

Example A wholesale nursery is selling garden flowers, and they want to summarize their sales figures by month. The data file which follows contains the customer ID, date of sale, and number of petunias, snapdragons, and marigolds sold:

```
756-01  05/04/2001 120  80 110
834-01  05/12/2001  90 160  60
901-02  05/18/2001  50 100  75
834-01  06/01/2001  80  60 100
756-01  06/11/2001 100 160  75
901-02  06/19/2001  60  60  60
756-01  06/25/2001  85 110 100
```

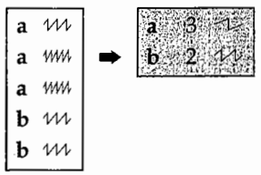
The following program reads the data; computes a new variable, Month, which is the month of the sale; sorts the data by Month using PROC SORT; then summarizes the data by Month using PROC MEANS with a BY statement:

```
DATA sales;
   INFILE 'c:\MyRawData\Flowers.dat';
   INPUT CustomerID $ @9 SaleDate MMDDYY10. Petunia SnapDragon
         Marigold;
   Month = MONTH(SaleDate);
PROC SORT DATA = sales;
   BY Month;
* Calculate means by Month for flower sales;
PROC MEANS DATA = sales;
   BY Month;
   VAR Petunia SnapDragon Marigold;
   TITLE 'Summary of Flower Sales by Month';
RUN;
```

Here are the results of the PROC MEANS:

```
                    Summary of Flower Sales by Month                1

--------------------------------- Month=5 ---------------------------------
                              The MEANS Procedure
Variable     N            Mean         Std Dev         Minimum         Maximum
-------------------------------------------------------------------------------
Petunia      3      86.6666667      35.1188458      50.0000000     120.0000000
SnapDragon   3     113.3333333      41.6333200      80.0000000     160.0000000
Marigold     3      81.6666667      25.6580072      60.0000000     110.0000000
-------------------------------------------------------------------------------

--------------------------------- Month=6 ---------------------------------
Variable     N            Mean         Std Dev         Minimum         Maximum
-------------------------------------------------------------------------------
Petunia      4      81.2500000      16.5201897      60.0000000     100.0000000
SnapDragon   4      97.5000000      47.8713554      60.0000000     160.0000000
Marigold     4      83.7500000      19.7378655      60.0000000     100.0000000
-------------------------------------------------------------------------------
```

4.10 Writing Summary Statistics to a SAS Data Set



Sometimes you want to save summary statistics to a SAS data set for further analysis, or to merge with other data. For example, you might want to plot the hourly temperature in your office to show how it heats up every afternoon causing you to fall asleep, but the instrument you have records data for every minute. The MEANS procedure can condense the data by computing the mean temperature for each hour and then save the results in a SAS data set so it can be plotted.

There are two methods in PROC MEANS for saving summary statistics in a SAS data set. You can use the Output Delivery System (ODS), which is covered in section 5.3, or you can use the OUTPUT statement. The OUTPUT statement has the following form:

```
OUTPUT OUT = data-set output-statistic-list;
```

Here, *data-set* is the name of the SAS data set which will contain the results (this can be either temporary or permanent), and *output-statistic-list* defines which statistics you want and the associated variable names. You can have more than one OUTPUT statement and multiple output statistic lists. The following is one of the possible forms for *output-statistic-list*:

```
statistic(variable-list) = name-list
```

Here, *statistic* can be any of the statistics available in PROC MEANS (SUM, N, MEAN, for example), *variable-list* defines which of the variables in the VAR statement you want to output, and *name-list* defines the new variable names for the statistics. The new variable names must be in the same order as their corresponding variables in *variable-list*. For example, the following PROC MEANS statements produce a new data set called ZOOSUM, which contains one observation with the variables LionWeight, the mean of the lions' weights, and BearWeight, the mean of the bears' weights:

```
PROC MEANS DATA = zoo NOPRINT;
   VAR Lions Tigers Bears;
   OUTPUT OUT = zoosum MEAN(Lions Bears) = LionWeight BearWeight;
RUN;
```

The NOPRINT option in the PROC MEANS statement tells SAS there is no need to produce any printed results since we are saving the results in a SAS data set.[1]

The SAS data set created in the OUTPUT statement will contain all the variables defined in the *output statistic list*; any variables listed in a BY or CLASS statement; plus two new variables, _TYPE_ and _FREQ_. If there is no BY or CLASS statement, then the data set will have just one observation. If there is a BY statement, then the data set will have one observation for each level of the BY group. CLASS statements produce one observation for each level of interaction of the class variables. The value of the _TYPE_ variable depends on the level of interaction. The observation where _TYPE_ has a value of zero is the grand total.[2]

[1] Using PROC MEANS with a NOPRINT option is the same as using PROC SUMMARY.

[2] For a more detailed explanation of the _TYPE_ variable, see the SAS Help and Documentation.

Example The following are sales data for a wholesale nursery with the customer ID; date of sale; and the number of petunias, snapdragons, and marigolds sold:

```
756-01  05/04/2001 120  80 110
834-01  05/12/2001  90 160  60
901-02  05/18/2001  50 100  75
834-01  06/01/2001  80  60 100
756-01  06/11/2001 100 160  75
901-02  06/19/2001  60  60  60
756-01  06/25/2001  85 110 100
```

You want to summarize the data so that you have only one observation per customer containing the sum and mean of the number of plant sets sold, and you want to save the results in a SAS data set for further analysis. The following program reads the data from the file; sorts by the variable, CustomerID; and then uses the MEANS procedure with the NOPRINT option to calculate the sums and means by CustomerID. The results are saved in a SAS data set named TOTALS in the OUTPUT statement. The sums are given the original variable names Petunia, SnapDragon, and Marigold, and the means are given new variable names MeanPetunia, MeanSnapDragon, and MeanMarigold. A PROC PRINT is used to show the TOTALS data set:

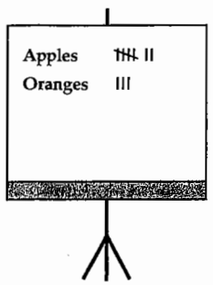
```
DATA sales;
   INFILE 'c:\MyRawData\Flowers.dat';
   INPUT CustomerID $ @9 SaleDate MMDDYY10. Petunia SnapDragon Marigold;
PROC SORT DATA = sales;
   BY CustomerID;
* Calculate means by CustomerID, output sum and mean to new data set;
PROC MEANS NOPRINT DATA = sales;
   BY CustomerID;
   VAR Petunia SnapDragon Marigold;
   OUTPUT OUT = totals  MEAN(Petunia SnapDragon Marigold) =
          MeanPetunia MeanSnapDragon MeanMarigold
          SUM(Petunia SnapDragon Marigold) = Petunia SnapDragon Marigold;
PROC PRINT DATA = totals;
   TITLE 'Sum of Flower Data over Customer ID';
   FORMAT MeanPetunia MeanSnapDragon MeanMarigold 3.;
RUN;
```

Here are the results:

Sum of Flower Data over Customer ID 1

Obs	Customer ID	_TYPE_	_FREQ_	Mean Petunia	Mean Snap Dragon	Mean Marigold	Snap Petunia	Snap Dragon	Snap Marigold
1	756-01	0	3	102	117	95	305	350	285
2	834-01	0	2	85	110	80	170	220	160
3	901-02	0	2	55	80	68	110	160	135

4.11 Counting Your Data with PROC FREQ



A frequency table is a simple list of counts answering the question "How many?" When you have counts for one variable, they are called one-way frequencies. When you combine two or more variables, the counts are called two-way, three-way, and so on up to *n*-way frequencies; or simply cross-tabulations.

The most obvious reason for using PROC FREQ is to create tables showing the distribution of categorical data values, but PROC FREQ can also reveal irregularities in your data. You could get dizzy proofreading a large data set, but data entry errors are often glaringly obvious in a frequency table. The basic form of PROC FREQ is

```
PROC FREQ;
   TABLES variable-combinations;
```

To produce a one-way frequency table, just list the variable name. This statement produces a frequency table listing the number of observations for each value of YearsEducation:

```
TABLES YearsEducation;
```

To produce a cross-tabulation, list the variables separated by an asterisk. This statement produces a cross-tabulation showing the number of observations for each combination of Sex by YearsEducation:

```
TABLES Sex * YearsEducation;
```

You can specify any number of table requests in a single TABLES statement, and you can have as many TABLES statements as you wish. Be careful though; reading cross-tabulations of three or more levels is like playing three-dimensional tic-tac-toe without the benefit of a three-dimensional board.

Options, if any, appear after a slash in the TABLES statement. For a list of statistical options for PROC FREQ see section 8.3. Options for controlling the output of PROC FREQ include

LIST	prints cross-tabulations in list format rather than grid
MISSING	includes missing values in frequency statistics
NOCOL	suppresses printing of column percentages in cross-tabulations
NOROW	suppresses printing of row percentages in cross-tabulations
OUT = *data-set*	writes a data set containing frequencies

The statement below, for instance, tells SAS to include missing values in the frequencies:

```
TABLES Sex * YearsEducation / MISSING;
```

Example The proprietor of a local coffee shop, Cathy's Coffee Cup, keeps a record of all sales. For each drink sold, she records the type of coffee (cappuccino, espresso, kona, or iced coffee), and whether the customer walked in or came to the drive-up window. Here are the data with ten observations per line of raw data:

```
esp w cap d cap w kon w ice w kon d esp d kon w ice d esp d
cap w esp d cap d Kon d .   d kon w esp d cap w ice w kon w
kon w kon w ice d esp d kon w esp d esp w kon w cap w kon w
```


The following program reads the data and produces one-way and two-way frequencies:

```
DATA orders;
   INFILE 'c:\MyRawData\Coffee.dat';
   INPUT Coffee $ Window $ @@;
* Print tables for Window and Window by Coffee;
PROC FREQ DATA = orders;
   TABLES Window Window * Coffee;
   RUN;
```

The output contains two tables. The first is a one-way frequency table for the variable Window. You can see that 13 customers came to the drive-up window while 17 walked into the restaurant.

The FREQ Procedure

Window	Frequency	Percent	Cumulative Frequency	Cumulative Percent
d	13	43.33	13	43.33
w	17	56.67	30	100.00

Table of Window by Coffee

Window Frequency Percent Row Pct Col Pct	Kon	cap	esp	ice	kon	Total
d	1	2	6	2	1	12
	3.45	6.90	20.69	6.90	3.45	41.38
	8.33	16.67	50.00	16.67	8.33	
	100.00	33.33	75.00	50.00	10.00	
w	0	4	2	2	9	17
	0.00	13.79	6.90	6.90	31.03	58.62
	0.00	23.53	11.76	11.76	52.94	
	0.00	66.67	25.00	50.00	90.00	
Total	1	6	8	4	10	29
	3.45	20.69	27.59	13.79	34.48	100.00

Frequency Missing = 1

The second table is a two-way cross-tabulation of Window by Coffee. Inside each cell, SAS prints the frequency, percentage, percentage for that row, and percentage for that column; while cumulative frequencies and percents appear along the right side and bottom. Notice that the missing value is mentioned but not included in the statistics. (Use the MISSING option if you want missing values to be included in the table.) Also, there is one observation with a value of Kon for Coffee. This data entry error should be kon.

4.12 Producing Tabular Reports with PROC TABULATE

Every summary statistic the TABULATE procedure computes can also be produced by other procedures such as PRINT, MEANS, and FREQ, but PROC TABULATE is popular because its reports are pretty. If TABULATE were a box, it would be gift-wrapped.

PROC TABULATE is so powerful that entire books have been written about it, but it is also so concise that you may feel like you're reading hieroglyphics. If you find the syntax of PROC TABULATE a little hard to get used to, that may be because it has roots outside of SAS. PROC TABULATE is based in part on the Table Producing Language, a complex and sophisticated language developed by the U.S. Department of Labor.

The general form of PROC TABULATE is

```
PROC TABULATE;
   CLASS classification-variable-list;
   TABLE page-dimension, row-dimension, column-dimension;
```

The CLASS statement tells SAS which variables contain categorical data to be used for dividing observations into groups, while the TABLE statement tells SAS how to organize your table and what numbers to compute. Each TABLE statement defines only one table, but you may have multiple TABLE statements. If a variable is listed in a CLASS statement, then, by default, PROC TABULATE produces simple counts of the number of observations in each category of that variable. PROC TABULATE offers many other statistics too, and section 4.13 describes how to request those.

Dimensions Each TABLE statement can specify up to three dimensions. Those dimensions, separated by commas, tell SAS which variables to use for the pages, rows, and columns in the report. If you specify only one dimension, then that becomes, by default, the column dimension. If you specify two dimensions, then you get rows and columns, but no page dimension. If you specify three dimensions, then you get pages, rows, and columns.

When you write a TABLE statement, start with the column dimension. Once you have that debugged, add the rows. Once you are happy with your rows and columns, then you are ready to add a page dimension, if you need one. Notice that the order of dimensions in the TABLE statement is page, then row, then column. So, to avoid scrambling your table when you add dimensions, insert the page and row specifications *in front* of the column dimension.

Missing data By default, observations are excluded from tables if they have missing values for variables listed in a CLASS statement. If you want to keep these observations, then simply add the MISSING option to your PROC statement like this:

```
PROC TABULATE MISSING;
```

Example Here are data about pleasure boats including the name of each boat, its home port, whether it is a sailing or power vessel, the type of boat (schooner, catamaran, or yacht), and the price of an excursion.


```
Silent Lady  Maalea  sail  sch 75.00
America II   Maalea  sail  yac 32.95
Aloha Anai   Lahaina sail  cat 62.00
Ocean Spirit Maalea  power cat 22.00
Anuenue      Maalea  sail  sch 47.50
Hana Lei     Maalea  power cat 28.99
Leilani      Maalea  power yac 19.99
Kalakaua     Maalea  power cat 29.50
Reef Runner  Lahaina power yac 29.95
Blue Dolphin Maalea  sail  cat 42.95
```

Suppose you want a report showing the number of boats of each type that are sailing or power vessels in each port. The following DATA step reads the data from a raw data file named Boats.dat. Then PROC TABULATE creates a three-dimensional report with the values of Port for the pages, Locomotion for the rows, and Type for the columns.

```
DATA boats;
   INFILE 'c:\MyRawData\Boats.dat';
   INPUT Name $ 1-12 Port $ 14-20 Locomotion $ 22-26 Type $ 28-30
      Price 32-36;

* Tabulations with three dimensions;
PROC TABULATE DATA = boats;
   CLASS Port Locomotion Type;
   TABLE Port, Locomotion, Type;
   TITLE 'Number of Boats by Port, Locomotion, and Type';
RUN;
```

This report has two pages, one for each value of the page dimension. Here is one page:

Number of Boats by Port, Locomotion, and Type 2

Port Maalea

	Type		
	cat	sch	yac
	N	N	N
Locomotion			
power	3.00	.	1.00
sail	1.00	2.00	1.00

The value of the page dimension appears in the top, left corner of the output. You can see that this is the page for the port of Maalea. The heading N tells you that the numbers in this table are simple counts, the number of boats in each group.

4.13 Adding Statistics to PROC TABULATE Output

By default, PROC TABULATE produces simple counts for variables listed in a CLASS statement, but you can request many other statistics in a TABLE statement. You can also concatenate or cross variables within dimensions. In fact, you can write TABLE statements so complicated that even *you* won't know what the report is going to look like until you run it.

While the CLASS statement lists categorical variables, the VAR statement tells SAS which variables contain continuous data. Here is the general form:

```
PROC TABULATE;
   VAR analysis-variable-list;
   CLASS classification-variable-list;
   TABLE page-dimension, row-dimension, column-dimension;
```

You may have both a CLASS statement and a VAR statement, or just one, but all variables listed in a TABLE statement must also appear in either a CLASS or a VAR statement.

Keywords In addition to variable names, each dimension can contain keywords. These are a few of the values TABULATE can compute.

ALL	adds a row, column, or page showing the total
MAX	highest value
MIN	lowest value
MEAN	the arithmetic mean
MEDIAN	the median
N	number of non-missing values
NMISS	number of missing values
P90	the 90th percentile
PCTN	the percentage of observations for that group
PCTSUM	the percentage of a total sum represented by that group
STDDEV	the standard deviation
SUM	the sum

Concatenating, crossing, and grouping Within a dimension, variables and keywords can be concatenated, crossed, or grouped. To concatenate variables or keywords simply list them separated by a space, to cross variables or keywords separate then with an asterisk (*), and to group them enclose the variables or keywords in parentheses. The keyword ALL is generally concatenated. To request other statistics, however, cross that keyword with the variable name.

Concatenating:	`TABLE Locomotion Type ALL;`
Crossing:	`TABLE MEAN * Price;`
Crossing, grouping, and concatenating:	`TABLE PCTN *(Locomotion Type);`

Example Here again are the boat data containing the name of each boat, its home port, whether it is a sailing or power vessel, the type of boat (schooner, catamaran, or yacht), and the price of an excursion.

```
Silent Lady  Maalea  sail  sch 75.00
America II   Maalea  sail  yac 32.95
Aloha Anai   Lahaina sail  cat 62.00
Ocean Spirit Maalea  power cat 22.00
Anuenue      Maalea  sail  sch 47.50
Hana Lei     Maalea  power cat 28.99
```



```
Leilani      Maalea  power yac 19.99
Kalakaua     Maalea  power cat 29.50
Reef Runner  Lahaina power yac 29.95
Blue Dolphin Maalea  sail  cat 42.95
```

The following program is similar to the one in section 4.12. However, this PROC TABULATE includes a VAR statement. The TABLE statement in this program contains only two dimensions; but it also concatenates, crosses, and groups variables and statistics.

```
DATA boats;
   INFILE 'c:\MyRawData\Boats.dat';
   INPUT Name $ 1-12 Port $ 14-20 Locomotion $ 22-26 Type $ 28-30
      Price 32-36;

* Tabulations with two dimensions and statistics;
PROC TABULATE DATA = boats;
   CLASS Locomotion Type;
   VAR Price;
   TABLE Locomotion ALL, MEAN*Price*(Type ALL);
   TITLE 'Mean Price by Locomotion and Type';
RUN;
```

The row dimension of this table concatenates the classification variable Locomotion with ALL to produce totals. The column dimension, on the other hand, crosses MEAN with the analysis variable Price and with the classification variable Type (which happens to be concatenated and grouped with ALL). Here are the results:

Mean Price by Locomotion and Type 1

	Mean			
	Price			
	Type			
	cat	sch	yac	All
Locomotion				
power	26.83	.	24.97	26.09
sail	52.48	61.25	32.95	52.08
All	37.09	61.25	27.63	39.08

4.14 Enhancing the Appearance of PROC TABULATE Output

When you use PROC TABULATE, SAS wraps your data in tidy little boxes, but there may be times when they just don't look right. Using three simple options, you can enhance the appearance of your output. Think of it as changing the wrapping paper.

FORMAT= option To change the format of all the data cells in your table, use the FORMAT= option in your PROC statement. For example, if you needed the numbers in your table to have commas and no decimal places, you could use this PROC statement

```
PROC TABULATE FORMAT=COMMA10.0;
```

telling SAS to use the COMMA10.0 format for all the data cells in your table.

BOX= and MISSTEXT= options While the FORMAT= option must be used in your PROC statement, the BOX= and MISSTEXT= options go in TABLE statements. The BOX= option allows you to write a brief phrase in the normally empty box that appears in the upper left corner of every TABULATE report. Using this empty space can give your reports a nicely polished look. The MISSTEXT= option, on the other hand, specifies a value for SAS to print in empty data cells. The period that SAS prints, by default, for missing values can seem downright mysterious to someone, perhaps your CEO, who is not familiar with SAS output. You can give them something more meaningful with the MISSTEXT= option. This statement

```
TABLE Region, MEAN*Sales / BOX='Mean Sales by Region' MISSTEXT='No Sales';
```

tells SAS to print the title "Mean Sales by Region" in the upper left corner of the table, and to print the words "No Sales" in any cells of the table that have no data. The BOX= and MISSTEXT= options must be separated from the dimensions of the TABLE statement by a slash.

Example Here again are the boat data containing the name of each boat, its home port, whether it is a sailing or power vessel, the type of boat (schooner, catamaran, or yacht), and the price of an excursion.

```
Silent Lady  Maalea  sail  sch 75.00
America II   Maalea  sail  yac 32.95
Aloha Anai   Lahaina sail  cat 62.00
Ocean Spirit Maalea  power cat 22.00
Anuenue      Maalea  sail  sch 47.50
Hana Lei     Maalea  power cat 28.99
Leilani      Maalea  power yac 19.99
Kalakaua     Maalea  power cat 29.50
Reef Runner  Lahaina power yac 29.95
Blue Dolphin Maalea  sail  cat 42.95
```

The following program is the same as the one in the previous section except that the FORMAT=, BOX=, and MISSTEXT= options have been added. Notice that the FORMAT= option goes in the PROC statement, while the BOX= and MISSTEXT= options go in the TABLE statement following a slash. Because the BOX= option serves as a title, a null TITLE statement is used to remove the usual title.


```
DATA boats;
   INFILE 'c:\MyRawData\Boats.dat';
   INPUT Name $ 1-12 Port $ 14-20 Locomotion $ 22-26 Type $ 28-30
      Price 32-36;

* PROC TABULATE report with options;
PROC TABULATE DATA = boats FORMAT=DOLLAR9.2;
   CLASS Locomotion Type;
   VAR Price;
   TABLE Locomotion ALL, MEAN*Price*(Type ALL)
      /BOX='Full Day Excursions' MISSTEXT='none';
   TITLE;
RUN;
```

Here is the enhanced output:

1

Full Day Excursions	Mean			
	Price			
	Type			
	cat	sch	yac	All
Locomotion				
power	$26.83	none	$24.97	$26.09
sail	$52.48	$61.25	$32.95	$52.08
All	$37.09	$61.25	$27.63	$39.08

Notice that all the data cells now use the DOLLAR9.2 format as specified in the FORMAT= option. The text "Full Day Excursions" now appears in the upper left corner which was empty in the previous section. In addition, the one data cell with no data now shows the word "none" instead of a period.

4.15 Changing Headers in PROC TABULATE Output

The TABULATE procedure produces reports with a lot of headers. Sometimes there are so many headers that your reports look cluttered; at other times you may simply feel that a different header would be more meaningful. Before you can change a header, though, you need to understand what type of header it is. TABULATE reports have two basic types of headers: headers that are the values of variables listed in a CLASS statement, and headers that are the names of variables and keywords. You use different methods to change different types of headers.

CLASS variable values To change headers which are the values of variables listed in a CLASS statement, use the FORMAT procedure to create a user-defined format. Then assign the format to the variable in a FORMAT statement (section 4.7).

Variable names and keywords To change headers which are the names of variables or keywords, put an equal sign after the variable or keyword followed by the new header enclosed in quotation marks.[1] You can eliminate a header entirely by setting it equal to blank (two quotation marks with nothing in between), and SAS will remove the box for that header. This TABLE statement

```
TABLE Region='', MEAN=''*Sales='Mean Sales by Region';
```

tells SAS to remove the headers for Region, and MEAN, and to change the header for the variable Sales to "Mean Sales by Region."

In some cases SAS leaves the empty box when a row header is set to blank. This happens for statistics and analysis variables (but not class variables). To force SAS to remove the empty box, add the ROW=FLOAT option to the end of your TABLE statement like this:

```
TABLE MEAN=''*Sales='Mean Sales by Region', Region='' / ROW=FLOAT;
```

Example Here again are the boat data containing the name of each boat, its home port, whether it is a sailing or power vessel, the type of boat (schooner, catamaran, or yacht), and the price of an excursion.

```
Silent Lady  Maalea  sail  sch 75.00
America II   Maalea  sail  yac 32.95
Aloha Anai   Lahaina sail  cat 62.00
Ocean Spirit Maalea  power cat 22.00
Anuenue      Maalea  sail  sch 47.50
Hana Lei     Maalea  power cat 28.99
Leilani      Maalea  power yac 19.99
Kalakaua     Maalea  power cat 29.50
Reef Runner  Lahaina power yac 29.95
Blue Dolphin Maalea  sail  cat 42.95
```

The following program is the same as the one in the previous section except that the headers have been changed. To start with, a FORMAT procedure creates a user-defined format named $typ.

[1] You can also change variable headers with a LABEL statement (section 4.1), and keyword headers with a KEYLABEL statement. However, the TABLE statement method used in this section is the only way that you can remove a variable header without leaving a blank box behind.

Then the $typ. format is assigned to the variable Type using a FORMAT statement. In the TABLE statement, more headers are changed. The headers for Locomotion, MEAN, and Type are all set to blank, while the header for Price is set to "Mean Price by Type of Boat."

```
DATA boats;
   INFILE 'c:\MyRawData\Boats.dat';
   INPUT Name $ 1-12 Port $ 14-20 Locomotion $ 22-26 Type $ 28-30
      Price 32-36;

* Changing headers;
PROC FORMAT;
   VALUE $typ 'cat' = 'catamaran'
              'sch' = 'schooner'
              'yac' = 'yacht';

PROC TABULATE DATA = boats FORMAT=DOLLAR9.2;
   CLASS Locomotion Type;
   VAR Price;
   FORMAT Type $typ.;
   TABLE Locomotion='' ALL,
      MEAN='' *Price='Mean Price by Type of Boat'*(Type='' ALL)
      /BOX='Full Day Excursions' MISSTEXT='none';
   TITLE;
RUN;
```

This program does not require the ROW=FLOAT option because the only variable being set to blank in the row dimension is a class variable. If you put an analysis variable or statistics keyword in the row dimension and set it equal to blank, then you would need to add the ROW=FLOAT option to remove empty boxes. Here is the output:

1

Full Day Excursions	Mean Price by Type of Boat			
	catamaran	schooner	yacht	All
power	$26.83	none	$24.97	$26.09
sail	$52.48	$61.25	$32.95	$52.08
All	$37.09	$61.25	$27.63	$39.08

This output is the same as the output in the preceding section, except for the new headers. Notice how much cleaner and more compact this report is.

4.16 Specifying Multiple Formats for Data Cells in PROC TABULATE Output

Using the FORMAT= option in a PROC TABULATE statement, you can easily specify a format for the data cells; but you can only specify one format, and it must apply to all the data cells. If you want to use more than one format in your table, you can do that by putting the FORMAT= option in your TABLE statement.

To apply a format to an individual variable, cross it with the variable name. The general form of this is

```
variable-name*FORMAT=formatw.d
```

Then you insert this rather convoluted construction in your TABLE statement.

```
TABLE Region, MEAN*(Sales*FORMAT=COMMA8.0 Profit*FORMAT=DOLLAR10.2);
```

This TABLE statement applies the COMMA8.0 format to a variable named Sales, and the DOLLAR10.2 format to Profit.

Example Here again are the boat data containing the name of each boat, its home port, whether it is a sailing or power vessel, the type of boat (schooner, catamaran, or yacht), and the price of an excursion. A new variable has been added showing the length of each boat in feet.

```
Silent Lady  Maalea  sail  sch 75.00  64
America II   Maalea  sail  yac 32.95  65
Aloha Anai   Lahaina sail  cat 62.00  60
Ocean Spirit Maalea  power cat 22.00  65
Anuenue      Maalea  sail  sch 47.50  52
Hana Lei     Maalea  power cat 28.99 110
Leilani      Maalea  power yac 19.99  45
Kalakaua     Maalea  power cat 29.50  70
Reef Runner  Lahaina power yac 29.95  50
Blue Dolphin Maalea  sail  cat 42.95  65
```

Suppose you want to show the mean price and mean length of boats side-by-side in the same report. Using dollar signs makes sense for price, but not for length. In the program below, the format DOLLAR6.2 is applied to the variable Price, while the format 6.0 is applied to Length. Notice that the FORMAT= options are crossed with the variables using an asterisk.

```
DATA boats;
   INFILE 'c:\MyRawData\Boats.dat';
   INPUT Name $ 1-12 Port $ 14-20 Locomotion $ 22-26 Type $ 28-30
      Price 32-36 Length 38-40;

* Using the FORMAT= option in the TABLE statement;
PROC TABULATE DATA = boats;
   CLASS Locomotion Type;
   VAR Price Length;
   TABLE Locomotion ALL,
      MEAN * (Price*FORMAT=DOLLAR6.2 Length*FORMAT=6.0) * (Type ALL);
   TITLE 'Price and Length by Type of Boat';
RUN;
```


Here is the resulting output:

Price and Length by Type of Boat 1

	Mean							
	Price				Length			
	Type				Type			
	cat	sch	yac	All	cat	sch	yac	All
Locomotion								
power	$26.83	.	$24.97	$26.09	82	.	48	68
sail	$52.48	$61.25	$32.95	$52.08	63	58	65	61
All	$37.09	$61.25	$27.63	$39.08	74	58	53	65

Notice that the values for Price and Length are printed using different formats.

4.17 Producing Simple Output with PROC REPORT

The REPORT procedure shares features with the PRINT, MEANS, TABULATE, and SORT procedures and the DATA step. With all those features rolled into one procedure, it's not surprising that PROC REPORT can be complex—in fact entire books have been written about it—but with all those features comes power.

Here is the general form of a basic REPORT procedure:

```
PROC REPORT NOWINDOWS;
COLUMN variable-list;
```

In its simplest form, the COLUMN statement is similar to a VAR statement in PROC PRINT, telling SAS which variables to include and in what order. If you leave out the COLUMN statement, SAS will, by default, include all the variables in your data set. If you leave out the NOWINDOWS option, SAS will open the interactive Report window.[1]

By default, PROC REPORT prints your data immediately beneath the column headers. To visually separate the headers and data, use the HEADLINE or HEADSKIP options like this:

```
PROC REPORT NOWINDOWS HEADLINE HEADSKIP;
```

HEADLINE draws a line under the column headers while HEADSKIP puts a blank line beneath the column headers.[2]

Numeric versus character data The type of report you get from PROC REPORT depends, in part, on the type of data you use. If you have at least one character variable in your report, then, by default, you will get a detail report with one row per observation. If, on the other hand, your report includes only numeric variables, then, by default, PROC REPORT will sum those variables. Even dates will be summed, by default, because they are numeric.[3]

Example Here are data about national parks and monuments in the USA. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds.

```
Dinosaur               NM West 2  6
Ellis Island           NM East 1  0
Everglades             NP East 5  2
Grand Canyon           NP West 5  3
Great Smoky Mountains  NP East 3 10
Hawaii Volcanoes       NP West 2  2
Lava Beds              NM West 1  1
Statue of Liberty      NM East 1  0
Theodore Roosevelt     NP .    2  2
Yellowstone            NP West 9 11
Yosemite               NP West 2 13
```

[1] The Report window is a non-programming approach to using PROC REPORT. For more information see the SAS Help and Documentation.

[2] The HEADLINE and HEADSKIP options work only for the LISTING destination. If you send your output to another destination such as HTML, SAS will ignore these options. See chapter 5 for an explanation of destinations.

[3] You can override this default by assigning one of your numeric variables a usage type of DISPLAY in a DEFINE statement. See section 4.18.

The following program reads the data in a DATA step, and then runs two reports. The first report has no COLUMN statement so SAS will use all the variables, while the second uses a COLUMN statement to select just the numeric variables.

```
DATA natparks;
   INFILE 'c:\MyRawData\Parks.dat';
   INPUT Name $ 1-21 Type $ Region $ Museums Camping;

PROC REPORT DATA = natparks NOWINDOWS HEADLINE;
   TITLE 'Report with Character and Numeric Variables';
RUN;

PROC REPORT DATA = natparks NOWINDOWS HEADLINE;
   COLUMN Museums Camping;
   TITLE 'Report with Only Numeric Variables';
RUN;
```

While the two PROC steps are only slightly different, the reports they produced differ dramatically. The first report is almost identical to the output you would get from a PROC PRINT except for the absence of the OBS column. The second report, since it contained only numeric variables, was summed.

```
          Report with Character and Numeric Variables          1

Name                   Type    Region    Museums    Camping

Dinosaur               NM      West            2          6
Ellis Island           NM      East            1          0
Everglades             NP      East            5          2
Grand Canyon           NP      West            5          3
Great Smoky Mountains  NP      East            3         10
Hawaii Volcanoes       NP      West            2          2
Lava Beds              NM      West            1          1
Statue of Liberty      NM      East            1          0
Theodore Roosevelt     NP                      2          2
Yellowstone            NP      West            9         11
Yosemite               NP      West            2         13


          Report with Only Numeric Variables                   2

                    Museums     Camping

                         33          50
```

4.18 Using DEFINE Statements in PROC REPORT

The DEFINE statement is a general purpose statement that specifies options for an individual variable. You can have a DEFINE statement for every variable, but you only need to have a DEFINE statement if you want to specify an option for that particular variable. The general form of a DEFINE statement is

```
DEFINE variable / options 'column-header';
```

In a DEFINE statement, you specify the variable name followed by a slash and any options for that particular variable.

Usage Options The most important option is a usage option that tells SAS how that variable is to be used. Possible values of usage options include:[1]

ACROSS	creates a column for each unique value of the variable.
ANALYSIS	calculates statistics for the variable. This is the default usage for numeric variables, and the default statistic is sum.
DISPLAY	creates one row for each observation in the data set. This is the default usage for character variables.
GROUP	creates a row for each unique value of the variable.
ORDER	creates one row for each observation with rows arranged according to the values of the order variable.

Changing column headers There are several ways to change column headers in PROC REPORT including using a LABEL statement as described in section 4.1, or specifying a column header in a DEFINE statement.[2] The following statement tells SAS to arrange a report by the values of the variable Age, and use the words "Age at Admission" as the column header for that variable. Using a slash in a column header tells SAS to split the header at that point.[3]

```
DEFINE Age / ORDER 'Age at/Admission';
```

Missing data By default, observations are excluded from reports if they have missing values for order, group, or across variables. If you want to keep these observations, then simply add the MISSING option to your PROC statement like this:

```
PROC REPORT NOWINDOWS MISSING;
```

Example Here again are the data about national parks and monuments. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds.

[1] Another usage type is COMPUTED. See the SAS Help and Documentation for more information.

[2] In addition to the LABEL and DEFINE statements, you can change column headers in the COLUMN statement which allows you to create spanning headers. See the SAS Help and Documentation for more information.

[3] At the time this book was written, PROC REPORT did not automatically split mixed case variable names the way most procedures do.


```
Dinosaur              NM West 2  6
Ellis Island          NM East 1  0
Everglades            NP East 5  2
Grand Canyon          NP West 5  3
Great Smoky Mountains NP East 3 10
Hawaii Volcanoes      NP West 2  2
Lava Beds             NM West 1  1
Statue of Liberty     NM East 1  0
Theodore Roosevelt    NP .    2  2
Yellowstone           NP West 9 11
Yosemite              NP West 2 13
```

The following PROC REPORT contains two DEFINE statements. The first defines Region as having a usage type of ORDER. The second specifies a column header for the variable Camping. Camping is a numeric variable and has a default usage of ANALYSIS, so the DEFINE statement does not change its usage. Since the MISSING option appears in the PROC statement, observations with missing values of Region will be included in the report.

```
DATA natparks;
   INFILE 'c:\MyRawData\Parks.dat';
   INPUT Name $ 1-21 Type $ Region $ Museums Camping;

* PROC REPORT with ORDER variable, MISSING option, and column header;
PROC REPORT DATA = natparks NOWINDOWS HEADLINE MISSING;
   COLUMN Region Name Museums Camping;
   DEFINE Region / ORDER;
   DEFINE Camping / ANALYSIS 'Camp/Grounds';
   TITLE 'National Parks and Monuments Arranged by Region';
RUN;
```

Here is the resulting output:

National Parks and Monuments Arranged by Region 1

Region	Name	Museums	Camp Grounds
	Theodore Roosevelt	2	2
East	Ellis Island	1	0
	Everglades	5	2
	Great Smoky Mountains	3	10
	Statue of Liberty	1	0
West	Dinosaur	2	6
	Grand Canyon	5	3
	Hawaii Volcanoes	2	2
	Lava Beds	1	1
	Yellowstone	9	11
	Yosemite	2	13

Notice that there are three values of Region: missing, East, and West. If you have more than one order variable, then the data will be arranged according to the values of the one that comes first in the COLUMN statement, then by the one that comes second, and so on.

4.19 Creating Summary Reports with PROC REPORT

Two different usage types cause the REPORT procedure to "roll up" data into summary groups based on the values of a variable. While the GROUP usage type produces summary rows, the ACROSS usage type produces summary columns.[1]

Group variables Defining a group variable is fairly simple. Just specify the GROUP usage option in a DEFINE statement. By default, analysis variables will be summed.[2] The following PROC REPORT tells SAS to produce a report showing the sum of Salary and of Bonus with a row for each value of Department.

Department	Salary	Bonus
A	~~~	~~
B	~~	~

```
PROC REPORT DATA = employees NOWINDOWS;
   COLUMN Department Salary Bonus;
   DEFINE Department / GROUP;
```

Across variables To define an across variable, you also use a DEFINE statement. However, by default SAS produces counts rather than sums. To obtain sums[2] for across variables, you must tell SAS which variables to summarize. You do that by putting a comma between the across variable and analysis variable (or variables if you enclose them in parentheses). The following PROC REPORT tells SAS to produce a report showing the sum of Salary and of Bonus with one column for each value of Department.

Department			
A		B	
Salary	Bonus	Salary	Bonus
~~	~~	~~	~

```
PROC REPORT DATA = employees NOWINDOWS;
   COLUMN Department , (Salary Bonus);
   DEFINE Department / ACROSS;
```

Example Here again are the data about national parks and monuments. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds.

```
Dinosaur               NM West 2  6
Ellis Island           NM East 1  0
Everglades             NP East 5  2
Grand Canyon           NP West 5  3
Great Smoky Mountains  NP East 3 10
Hawaii Volcanoes       NP West 2  2
Lava Beds              NM West 1  1
Statue of Liberty      NM East 1  0
Theodore Roosevelt     NP .    2  2
Yellowstone            NP West 9 11
Yosemite               NP West 2 13
```

[1] If you have any display or order variables in the COLUMN statement, SAS will produce a "detail" report instead of consolidating data into summary groups.

[2] To request other statistics, see section 4.21.

The following program contains two PROC REPORTs. In the first, Region and Type are both defined as group variables. In the second, Region is still a group variable, but Type is an across variable. Notice that the two COLUMN statements are the same except for punctuation added to the second procedure to cross the across variable with the analysis variables.

```
DATA natparks;
   INFILE 'c:\MyRawData\Parks.dat';
   INPUT Name $ 1-21 Type $ Region $ Museums Camping;

* Region and Type as GROUP variables;
PROC REPORT DATA = natparks NOWINDOWS HEADLINE;
   COLUMN Region Type Museums Camping;
   DEFINE Region / GROUP;
   DEFINE Type / GROUP;
   TITLE 'Summary Report with Two Group Variables';
RUN;

* Region as GROUP and Type as ACROSS with sums;
PROC REPORT DATA = natparks NOWINDOWS HEADLINE;
   COLUMN Region Type,(Museums Camping);
   DEFINE Region / GROUP;
   DEFINE Type / ACROSS;
   TITLE 'Summary Report with a Group and an Across Variable';
RUN;
```

Here is the resulting output:

```
Summary Report with Two Group Variables                    1

Region     Type        Museums     Camping

East       NM                2           0
           NP                8          12
West       NM                3           7
           NP               18          29


Summary Report with a Group and an Across Variable         2

                          Type
                    NM                  NP
Region      Museums     Camping     Museums     Camping

East              2           0           8          12
West              3           7          18          29
```

4.20 Adding Summary Breaks to PROC REPORT Output

Two kinds of statements allow you to insert breaks into a report. The BREAK statement adds a break for each unique value of the variable you specify, while the RBREAK statement does the same for the entire report (or BY-group if you are using a BY statement). The general forms of these statements are

```
BREAK  location variable / options;
RBREAK location / options;
```

where *location* has two possible values—BEFORE or AFTER—depending on whether you want the break to precede or follow that particular section of the report. The options that come after the slash tell SAS what kind of break to insert. Some of the possible options are[1]

OL	draws a line over the break
PAGE	starts a new page
SKIP	inserts a blank line
SUMMARIZE	inserts sums of numeric variables
UL	draws a line under the break

Notice that the BREAK statement requires you to specify a variable, but the RBREAK statement does not. That's because the RBREAK statement produces only one break (at the beginning or end), while the BREAK statement produces one break for every unique value of the variable you specify. That variable must be either a group or order variable and therefore must also be listed in a DEFINE statement with either the GROUP or ORDER usage option. You can use an RBREAK statement in any report, but you can use BREAK only if you have at least one group or order variable.

Example Here again are the data about national parks and monuments. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds.

```
Dinosaur              NM West 2  6
Ellis Island          NM East 1  0
Everglades            NP East 5  2
Grand Canyon          NP West 5  3
Great Smoky Mountains NP East 3 10
Hawaii Volcanoes      NP West 2  2
Lava Beds             NM West 1  1
Statue of Liberty     NM East 1  0
Theodore Roosevelt    NP .    2  2
Yellowstone           NP West 9 11
Yosemite              NP West 2 13
```

The following program defines Region as an order variable, and then uses both BREAK and RBREAK statements with the AFTER location. The SUMMARIZE option tells SAS to print totals for numeric variables, while the OL and SKIP options tell SAS to draw a line above the totals and skip a line under the totals.

[1] All these options work for the Listing destination; not all work for other destinations. At the time this book was written, PAGE and SUMMARIZE worked for HTML, RTF, and PDF; OL, UL and SKIP were ignored.


```
DATA natparks;
   INFILE 'c:\MyRawData\Parks.dat';
   INPUT Name $ 1-21 Type $ Region $ Museums Camping;

* PROC REPORT with breaks;
PROC REPORT DATA = natparks NOWINDOWS HEADLINE;
   COLUMN Name Region Museums Camping;
   DEFINE Region / ORDER;
   BREAK AFTER Region / SUMMARIZE OL SKIP;
   RBREAK AFTER / SUMMARIZE OL SKIP;
   TITLE 'National Parks';
RUN;
```

Here is the resulting output:

National Parks 1

Name	Region	Museums	Camping
Ellis Island	East	1	0
Everglades		5	2
Great Smoky Mountains		3	10
Statue of Liberty		1	0
	East	10	12
Dinosaur	West	2	6
Grand Canyon		5	3
Hawaii Volcanoes		2	2
Lava Beds		1	1
Yellowstone		9	11
Yosemite		2	13
	West	21	36
		31	48

4.21 Adding Statistics to PROC REPORT Output

There are several ways to request statistics in the REPORT procedure. An easy method is to insert statistics keywords directly into the COLUMN statement along with the variable names. This is a little like requesting statistics in a TABLE statement in PROC TABULATE, except that instead of using an asterisk to cross a statistics keyword with a variable, you use a comma. In fact, PROC REPORT can produce all the same statistics as PROC TABULATE and PROC MEANS because it uses the same internal engine to compute those statistics. These are a few of the statistics PROC REPORT can compute:

MAX	highest value
MIN	lowest value
MEAN	the arithmetic mean
MEDIAN	the median
N	number of non-missing values
NMISS	number of missing values
P90	the 90th percentile
PCTN	the percentage of observations for that group
PCTSUM	the percentage of a total sum represented by that group
STD	the standard deviation
SUM	the sum

Applying statistics to variables To request a statistic for a particular variable, insert a comma between the statistic and variable in the COLUMN statement. One statistic, N, does not require a comma because it does not apply to a particular variable. If you insert N in a COLUMN statement, then SAS will print the number of observations that contributed to that row of the report. This statement tells SAS to print two columns of data: the median of a variable named Age, and the number of observations in that row.

```
COLUMN  Age,MEDIAN  N;
```

To request multiple statistics or statistics for multiple variables, put parentheses around the statistics or variables. This statement uses parentheses to request two statistics for the variable Age, and then requests one statistic for two variables, Height and Weight.

```
COLUMN Age,(MIN MAX) (Height Weight),MEAN;
```

Example Here again are the data about national parks and monuments. The variables are name, type (NP for national park or NM for national monument), region (East or West), number of museums (including visitor centers), and number of campgrounds.

```
Dinosaur               NM West 2  6
Ellis Island           NM East 1  0
Everglades             NP East 5  2
Grand Canyon           NP West 5  3
Great Smoky Mountains  NP East 3 10
Hawaii Volcanoes       NP West 2  2
Lava Beds              NM West 1  1
Statue of Liberty      NM East 1  0
Theodore Roosevelt     NP .    2  2
Yellowstone            NP West 9 11
Yosemite               NP West 2 13
```


The following program contains two PROC REPORTs. Both procedures request the statistics N and MEAN, but the first report defines Type as a group variable, while the second defines Type as an across variable.

```
DATA natparks;
   INFILE 'c:\MyRawData\Parks.dat';
   INPUT Name $ 1-21 Type $ Region $ Museums Camping;

*Statistics in COLUMN statement with two group variables;
PROC REPORT DATA = natparks NOWINDOWS HEADLINE;
   COLUMN Region Type N (Museums Camping),MEAN;
   DEFINE Region / GROUP;
   DEFINE Type / GROUP;
   TITLE 'Statistics with Two Group Variables';
RUN;

*Statistics in COLUMN statement with group and across variables;
PROC REPORT DATA = natparks NOWINDOWS HEADLINE;
   COLUMN Region N Type,(Museums Camping),MEAN;
   DEFINE Region / GROUP;
   DEFINE Type / ACROSS;
   TITLE 'Statistics with a Group and Across Variable';
RUN;
```

Here is the resulting output:

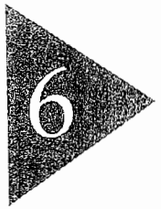
Statistics with Two Group Variables 1

Region	Type	N	Museums MEAN	Camping MEAN
East	NM	2	1	0
	NP	2	4	6
West	NM	2	1.5	3.5
	NP	4	4.5	7.25

Statistics with a Group and Across Variable 2

		Type			
		NM		NP	
Region	N	Museums MEAN	Camping MEAN	Museums MEAN	Camping MEAN
East	4	1	0	4	6
West	6	1.5	3.5	4.5	7.25

Notice that these reports are similar to the reports in section 4.19 except that these contain counts and means instead of sums.



"I usually say, 'The computer is the dumbest thing on campus. It does exactly what you tell it to; not necessarily what you want. Logic is up to you.'"

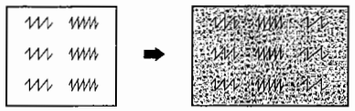
NECIA A. BLACK, R.N., PH.D.

From the SAS L Listserv, May 6, 1994. Reprinted by permission of the author.

CHAPTER 6

Modifying and Combining SAS® Data Sets

6.1 Modifying a Data Set Using the SET Statement



The SET statement in the DATA step allows you to read a SAS data set so you can add new variables, create a subset, or otherwise modify the data set. If you were short on disk space, for example, you might not want to store your computed variables in a permanent SAS data set. Instead, you might want to calculate them as needed for analysis. Likewise, to save processing time, you might want to create a subset of a SAS data set when you only want to look at a small portion of a large data set. The SET statement brings a SAS data set, one observation at a time, into the DATA step for processing.[1]

To read a SAS data set, start with the DATA statement specifying the name of the new data set. Then follow with the SET statement specifying the name of the old data set you want to read. If you don't want to create a new data set, you can specify the same name in the DATA and SET statements. Then the results of the DATA step will overwrite the old data set named in the SET statement.[2] The following shows the general form of the DATA and SET statements:

```
DATA new-data-set;
   SET data-set;
```

Any assignment, subsetting IF, or other DATA step statements usually follow the SET statement. For example, the following creates a new data set, FRIDAY, which is a replica of the SALES data set, except FRIDAY has only the observations for Fridays, and it has an additional variable, Total:

```
DATA friday;
   SET sales;
   IF Day = 'F';
   Total = Popcorn + Peanuts;
RUN;
```

Example The Fun Times Amusement Park is collecting data about their train ride. They can add more cars on the train during peak hours to shorten the wait, or take them off when they're not needed to save fuel costs. The raw data file contains data for the time of day, the number of cars on the train, and the total number of people on the train:

```
10:10  6 21
12:15 10 56
15:30 10 25
11:30  8 34
13:15  8 12
10:45  6 13
20:30  6 32
23:15  6 12
```

[1] The MODIFY statement also allows you to modify a single data set. See the SAS Help and Documentation for more information.

[2] By default, SAS will not overwrite a data set in a DATA step that has errors.

The data are read into a permanent SAS data set, TRAINS, stored in the MySASLib directory on the park's central computer by means of the following program:

```
* Create permanent SAS data set trains;
DATA 'c:\MySASLib\trains';
   INFILE 'c:\MyRawData\Train.dat';
   INPUT Time TIME5. Cars People;
RUN;
```

This example uses direct referencing to tell SAS where to store the permanent SAS data set, but you could use a LIBNAME statement instead.

Each train car holds a maximum of six people. After collecting the data, the Fun Times management decides they want to know the average number of people per car on each ride. This number was not calculated in the original DATA step which created the permanent SAS data set, but can be calculated by the following program:

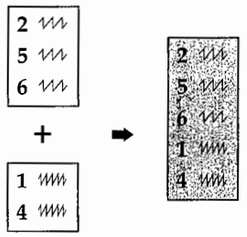
```
* Read the SAS data set trains with a SET statement;
DATA averagetrain;
   SET 'c:\MySASLib\trains';
   PeoplePerCar = People / Cars;
PROC PRINT DATA = averagetrain;
   TITLE 'Average Number of People per Train Car';
   FORMAT Time TIME5.;
RUN;
```

The DATA statement defines a new temporary SAS data set named AVERAGETRAIN. Then the SET statement reads the permanent SAS data set TRAINS, and an assignment statement creates the new variable PeoplePerCar. Here are the results of the PROC PRINT:

Average Number of People per Train Car 1

Obs	Time	Cars	People	People PerCar
1	10:10	6	21	3.50000
2	12:15	10	56	5.60000
3	15:30	10	25	2.50000
4	11:30	8	34	4.25000
5	13:15	8	12	1.50000
6	10:45	6	13	2.16667
7	20:30	6	32	5.33333
8	23:15	6	12	2.00000

6.2 Stacking Data Sets Using the SET Statement



The SET statement with one SAS data set allows you to read and modify the data. With two or more data sets, in addition to reading and modifying the data, the SET statement concatenates or stacks the data sets one on top of the other. This is useful when you want to combine data sets with all or most of the same variables but different observations. You might, for example, have data from two different locations or data taken at two separate times, but you need the data together for analysis.

In a DATA step, first specify the name of the new SAS data set in the DATA statement, then list the names of the old data sets you want to combine in the SET statement:

```
DATA new-data-set;
   SET data-set-1 data-set-n;
```

The number of observations in the new data set will equal the sum of the number of observations in the old data sets. The order of observations is determined by the order of the list of old data sets. If one of the data sets has a variable not contained in the other data sets, then the observations from the other data sets will have missing values for that variable.

Example The Fun Times Amusement Park has two entrances where they collect data about their customers. The data file for the south entrance has an S (for south) followed by the customers' Fun Times pass numbers, the sizes of their parties, and their ages. The file for the north entrance has an N (for north), the same data as the south entrance, plus one more column for the parking lot where they left their cars (the south entrance has only one lot). The following shows samples of the two data files:

Data for South Entrance

```
S 43 3 27
S 44 3 24
S 45 3  2
```

Data for North Entrance

```
N 21 5 41 1
N 87 4 33 3
N 65 2 67 1
N 66 2  7 1
```

The first two parts of the following program read the raw data for the south and north entrances into SAS data sets and print them to make sure they are correct. The third part combines the two SAS data sets using a SET statement. The same DATA step creates a new variable, AmountPaid, which tells how much each customer paid based on their age. This final data set is printed using PROC PRINT:

```
DATA southentrance;
   INFILE 'c:\MyRawData\South.dat';
   INPUT Entrance $ PassNumber PartySize Age;
PROC PRINT DATA = southentrance;
   TITLE 'South Entrance Data';

DATA northentrance;
   INFILE 'c:\MyRawData\North.dat';
   INPUT Entrance $ PassNumber PartySize Age Lot;
PROC PRINT DATA = northentrance;
   TITLE 'North Entrance Data';
```



```
* Create a data set, both, combining northentrance and southentrance;
* Create a variable, AmountPaid, based on value of variable Age;
DATA both;
   SET southentrance northentrance;
   IF Age = . THEN AmountPaid = .;
      ELSE IF Age < 3  THEN AmountPaid = 0;
      ELSE IF Age < 65 THEN AmountPaid = 35;
      ELSE AmountPaid = 27;
PROC PRINT DATA = both;
   TITLE 'Both Entrances';
RUN;
```

The following are the results of the three PRINT procedures in the program. Notice that the final data set has missing values for the variable Lot for all the observations which came from the south entrance. Because the variable Lot was not in the SOUTHENTRANCE data set, SAS assigned missing values to those observations.

South Entrance Data 1

Obs	Entrance	Pass Number	Party Size	Age
1	S	43	3	27
2	S	44	3	24
3	S	45	3	2

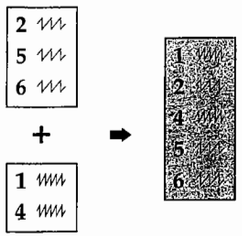
North Entrance Data 2

Obs	Entrance	Pass Number	Party Size	Age	Lot
1	N	21	5	41	1
2	N	87	4	33	3
3	N	65	2	67	1
4	N	66	2	7	1

Both Entrances 3

Obs	Entrance	Pass Number	Party Size	Age	Lot	Amount Paid
1	S	43	3	27	.	35
2	S	44	3	24	.	35
3	S	45	3	2	.	0
4	N	21	5	41	1	35
5	N	87	4	33	3	35
6	N	65	2	67	1	27
7	N	66	2	7	1	35

6.8 Interleaving Data Sets Using the SET Statement



The previous section explained how to stack data sets that have all or most of the same variables but different observations. However, if you have data sets that are already sorted by some important variable, then simply stacking the data sets may unsort the data sets. You could stack the two data sets and then re-sort them using PROC SORT. But if your data sets are already sorted, it is more efficient to preserve that order, than to stack and re-sort. All you need to do is use a BY statement with your SET statement. Here's the general form:

```
DATA new-data-set;
   SET data-set-1 data-set-n;
   BY variable-list;
```

In a DATA statement, you specify the name of the new SAS data set you want to create. In a SET statement, you list the data sets to be interleaved. Then in a BY statement, you list one or more variables that SAS should use for ordering the observations. The number of observations in the new data set will be equal to the sum of the number of observations in the old data sets. If one of the data sets has a variable not contained in the other data sets, then values of that variable will be set to missing for observations from the other data sets.

Before you can interleave observations, the data sets must be sorted by the BY variables. If one or the other of your data sets is not already sorted, then use PROC SORT to do the job.

Example To show how this is different from stacking data sets, we'll use the amusement park data again. There are two data sets, one for the south entrance and one for the north. For every customer, the park collects the following data: the entrance (S or N), the customer's Fun Times pass number, size of that customer's party, and age. For customers entering from the north, the data set also includes parking lot number. Here is a sample of the data:

Data for South Entrance

```
S 43 3 27
S 44 3 24
S 45 3  2
```

Data for North Entrance

```
N 21 5 41 1
N 87 4 33 3
N 65 2 67 1
N 66 2  7 1
```

Notice that the data for the south entrance are already sorted by pass number, but the data for the north entrance are not.

Instead of stacking the two data sets, this program interleaves the data sets by pass number. This program first reads the data for the south entrance and prints them to make sure they are correct. Then the program reads the data for the north entrance, sorts them, and prints them. Then in the final DATA step, SAS combines the two data sets, NORHTENTRANCE and SOUTHENTRANCE, creating a new data set named INTERLEAVE. The BY statement tells SAS to combine the data sets by PassNumber:

```
DATA southentrance;
   INFILE 'c:\MyRawData\South.dat';
   INPUT Entrance $ PassNumber PartySize Age;
PROC PRINT DATA = southentrance;
   TITLE 'South Entrance Data';
```



```
DATA northentrance;
   INFILE 'c:\MyRawData\North.dat';
   INPUT Entrance $ PassNumber PartySize Age Lot;
PROC SORT DATA = northentrance;
   BY PassNumber;
PROC PRINT DATA = northentrance;
   TITLE 'North Entrance Data';

* Interleave observations by PassNumber;
DATA interleave;
   SET northentrance southentrance;
   BY PassNumber;
PROC PRINT DATA = interleave;
   TITLE 'Both Entrances, By Pass Number';
RUN;
```

Here are the results of the three PRINT procedures. Notice how the observations have been interleaved so that the new data set is sorted by PassNumber:

South Entrance Data 1

Obs	Entrance	Pass Number	Party Size	Age
1	S	43	3	27
2	S	44	3	24
3	S	45	3	2

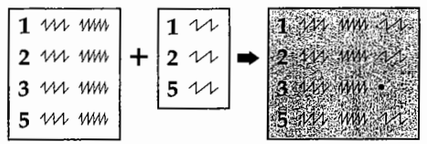
North Entrance Data 2

Obs	Entrance	Pass Number	Party Size	Age	Lot
1	N	21	5	41	1
2	N	65	2	67	1
3	N	66	2	7	1
4	N	87	4	33	3

Both Entrances, By Pass Number 3

Obs	Entrance	Pass Number	Party Size	Age	Lot
1	N	21	5	41	1
2	S	43	3	27	.
3	S	44	3	24	.
4	S	45	3	2	.
5	N	65	2	67	1
6	N	66	2	7	1
7	N	87	4	33	3

6.4 Combining Data Sets Using a One-to-One Match Merge



When you want to match observations from one data set with observations from another, use the MERGE statement in the DATA step. If you know the two data sets are in EXACTLY the same order, you don't have to have any common variables between the data sets. Typically, however, you will want to have, for matching purposes, a common variable or several variables which taken together uniquely identify each observation. This is important. Having a common variable to merge by ensures that the observations are properly matched. For example, to merge patient data with billing data, you would use the patient ID as a matching variable. Otherwise you risk getting Mary Smith's visit to the obstetrician mixed up with Matthew Smith's visit to the optometrist.

Merging SAS data sets is a simple process. First, if the data are not already sorted, use the SORT procedure to sort all data sets by the common variables. Then, in the DATA statement, name the new SAS data set to hold the results and follow with a MERGE statement listing the data sets to be combined. Use a BY statement to indicate the common variables:

```
DATA new-data-set;
   MERGE data-set-1 data-set-2;
   BY variable-list;
```

If you merge two data sets, and they have variables with the same names—besides the BY variables—then variables from the second data set will overwrite any variables having the same name in the first data set.

Example A Belgian chocolatier keeps track of the number of each type of chocolate sold each day. The code number for each chocolate and the number of pieces sold that day are kept in a file. In a separate file she keeps the names and descriptions of each chocolate as well as the code number. In order to print the day's sales along with the descriptions of the chocolates, the two files must be merged together using the code number as the common variable. Here is a sample of the data:

Sales data

```
C865 15
K086  9
A536 21
S163 34
K014  1
A206 12
B713 29
```

Descriptions

```
A206 Mokka      Coffee buttercream in dark chocolate
A536 Walnoot    Walnut halves in bed of dark chocolate
B713 Frambozen  Raspberry marzipan covered in milk chocolate
C865 Vanille    Vanilla-flavored rolled in ground hazelnuts
K014 Kroon      Milk chocolate with a mint cream center
K086 Koning     Hazelnut paste in dark chocolate
M315 Pyramide   White with dark chocolate trimming
S163 Orbais     Chocolate cream in dark chocolate
```


The first two parts of the following program read the descriptions and sales data. The descriptions data are already sorted by CodeNum, so we don't need to use PROC SORT. The sales data are not sorted, so a PROC SORT follows the DATA step. (If you attempt to merge data which are not sorted, SAS will refuse and give you this error message: ERROR: BY variables are not properly sorted.)

```
DATA descriptions;
   INFILE 'c:\MyRawData\chocolate.dat' TRUNCOVER;
   INPUT CodeNum $ 1-4 Name $ 6-14 Description $ 15-60;
DATA sales;
   INFILE 'c:\MyRawData\chocsales.dat';
   INPUT CodeNum $ 1-4 PiecesSold 6-7;
PROC SORT DATA = sales;
   BY CodeNum;

* Merge data sets by CodeNum;
DATA chocolates;
   MERGE sales descriptions;
   BY CodeNum;
PROC PRINT DATA = chocolates;
   TITLE "Today's Chocolate Sales";
RUN;
```

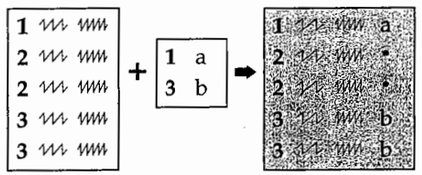
The final part of the program creates a data set named CHOCOLATES by merging the SALES data set and the DESCRIPTIONS data set. The common variable CodeNum in the BY statement is used for matching purposes. The following output shows the final data set after merging:

Today's Chocolate Sales 1

Obs	Code Num	Pieces Sold	Name	Description
1	A206	12	Mokka	Coffee buttercream in dark chocolate
2	A536	21	Walnoot	Walnut halves in bed of dark chocolate
3	B713	29	Frambozen	Raspberry marzipan covered in milk chocolate
4	C865	15	Vanille	Vanilla-flavored rolled in ground hazelnuts
5	K014	1	Kroon	Milk chocolate with a mint cream center
6	K086	9	Koning	Hazelnut paste in dark chocolate
7	M315	.	Pyramide	White with dark chocolate trimming
8	S163	34	Orbais	Chocolate cream in dark chocolate

Notice that the final data set has a missing value for PiecesSold in the seventh observation. This is because there were no sales for the Pyramide chocolate. All observations from both data sets were included in the final data set whether they had a match or not.

6.5 Combining Data Sets Using a One-to-Many Match Merge



Sometimes you need to combine two data sets by matching one observation from one data set with more than one observation in another. Suppose you had data for every state in the U.S. and wanted to combine it with data for every county. This would be a one-to-many match merge because each state observation matches with many county observations.

The statements for a one-to-many match merge are identical to the statements for a one-to-one match merge:

```
DATA new-data-set;
   MERGE data-set-1 data-set-2;
   BY variable-list;
```

The order of the data sets in the MERGE statement does not matter to SAS. In other words, a one-to-many merge is the same as a many-to-one merge.

Before you merge two data sets, they must be sorted by one or more common variables. If your data sets are not already sorted in the proper order, then use PROC SORT to do the job.

You cannot do a one-to-many merge without a BY statement. SAS uses the variables listed in the BY statement to decide which observations belong together. Without any BY variables for matching, SAS simply joins together the first observation from each data set, then the second observation from each data set, and so on. In other words, SAS performs a one-to-one unmatched merge, which is probably not what you want.

If you merge two data sets, and they have variables with the same names—besides the BY variables—then variables from the second data set will overwrite any variables having the same name in the first data set. For example, if you merge two data sets both containing a variable named Score, then the final data set will contain only one variable named Score. The values for Score will come from the second data set. You can fix this by renaming the variables (giving them names such as Score1 and Score2) so that they will not overwrite each other.[1]

Example A distributor of athletic shoes is putting all its shoes on sale at 20 to 30% off the regular price. The distributor has two data files, one with information about each type of shoe and one with the discount factors. The first file contains one record for each shoe with values for style, type of exercise (running, walking, or cross-training), and regular price. The second file contains one record for each type of exercise and its discount. Here are the two raw data files:

Shoes data

```
Max Flight      running 142.99
Zip Fit Leather walking  83.99
Zoom Airborne   running 112.99
Light Step      walking  73.99
Max Step Woven  walking  75.99
Zip Sneak       c-train  92.99
```

Discount data

```
c-train .25
running .30
walking .20
```

[1] The RENAME= data set option is discussed in section 6.9.

To find the sale price, the following program combines the two data files:

```
DATA regular;
   INFILE 'c:\MyRawData\Shoe.dat';
   INPUT Style $ 1-15 ExerciseType $ RegularPrice;
PROC SORT DATA = regular;
   BY ExerciseType;

DATA discount;
   INFILE 'c:\MyRawData\Disc.dat';
   INPUT ExerciseType $ Adjustment;

* Perform many-to-one match merge;
DATA prices;
   MERGE regular discount;
   BY ExerciseType;
   NewPrice = ROUND(RegularPrice - (RegularPrice * Adjustment), .01);
PROC PRINT DATA = prices;
   TITLE 'Price List for May';
RUN;
```

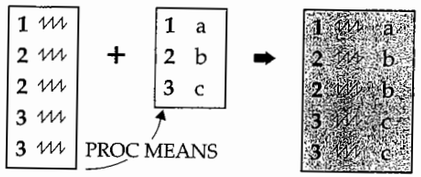
The first DATA step reads the regular prices, creating a data set named REGULAR. That data set is then sorted by ExerciseType using PROC SORT. The second DATA step reads the price adjustments, creating a data set named DISCOUNT. This data set is already arranged by ExerciseType, so it doesn't have to be sorted. The third DATA step creates a data set named PRICES, merging the first two data sets by ExerciseType, and computes a variable called NewPrice. The output looks like this:

Price List for May 1

Obs	Style	Exercise Type	Regular Price	Adjustment	New Price
1	Zip Sneak	c-train	92.99	0.25	69.74
2	Max Flight	running	142.99	0.30	100.09
3	Zoom Airborne	running	112.99	0.30	79.09
4	Zip Fit Leather	walking	83.99	0.20	67.19
5	Light Step	walking	73.99	0.20	59.19
6	Max Step Woven	walking	75.99	0.20	60.79

Notice that the values for Adjustment from the DISCOUNT data set are repeated for every observation in the REGULAR data set with the same value of ExerciseType.

6.6 Merging Summary Statistics with the Original Data



Once in a while you need to combine summary statistics with your data, such as when you want to compare each observation to the group mean, or when you want to calculate a percentage using the group total. To do this, summarize your data using PROC MEANS, and put the results in a new data set. Then merge the summarized data back with the original data using a one-to-many match merge.

Example A distributor of athletic shoes is considering doing a special promotion for the top selling styles. The vice-president of marketing has asked you to produce a report. The report should be divided by type of exercise (running, walking, or cross-training) and show the percentage of sales for each style within its type. For each shoe, the raw data file contains the style name, type of exercise, and total sales for the last quarter:

```
Max Flight       running 1930
Zip Fit Leather  walking 2250
Zoom Airborne    running 4150
Light Step       walking 1130
Max Step Woven   walking 2230
Zip Sneak        c-train 1190
```

Here is the program:

```
DATA shoes;
   INFILE 'c:\MyRawData\Shoesales.dat';
   INPUT Style $ 1-15 ExerciseType $ Sales;
PROC SORT DATA = shoes;
   BY ExerciseType;

* Summarize sales by ExerciseType and print;
PROC MEANS NOPRINT DATA = shoes;
   VAR Sales;
   BY ExerciseType;
   OUTPUT OUT = summarydata SUM(Sales) = Total;
PROC PRINT DATA = summarydata;
   TITLE 'Summary Data Set';

* Merge totals with the original data set;
DATA shoesummary;
   MERGE shoes summarydata;
   BY ExerciseType;
   Percent = Sales / Total * 100;
PROC PRINT DATA = shoesummary;
   BY ExerciseType;
   ID ExerciseType;
   VAR Style Sales Total Percent;
   TITLE 'Sales Share by Type of Exercise';
RUN;
```

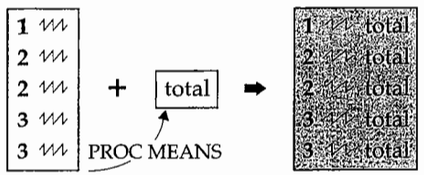

This program is long but straightforward. It starts by reading the raw data in a DATA step and sorting them with PROC SORT. Then it summarizes the data with PROC MEANS by the variable ExerciseType. The OUTPUT statement tells SAS to create a new data set named SUMMARYDATA, containing a variable named Total, which equals the sum of the variable Sales. The NOPRINT option tells SAS not to print the standard PROC MEANS report. Instead, the summary data set is printed by PROC PRINT:

```
                          Summary Data Set                          1

                Exercise
          Obs     Type       _TYPE_    _FREQ_    Total

           1    c-train         0         1       1190
           2    running         0         2       6080
           3    walking         0         3       5610
```

In the last part of the program, the original data set, SHOES, is merged with SUMMARYDATA to make a new data set, SHOESUMMARY. This DATA step computes a new variable called Percent. Then the last PROC PRINT writes the final report with percentage of sales by ExerciseType for each title. Using a BY and an ID statement together gives this report a little different look:

```
                    Sales Share by Type of Exercise                 2

  Exercise
    Type       Style              Sales    Total    Percent

  c-train      Zip Sneak           1190     1190    100.000

  running      Max Flight          1930     6080     31.743
               Zoom Airborne       4150     6080     68.257

  walking      Zip Fit Leather     2250     5610     40.107
               Light Step          1130     5610     20.143
               Max Step Woven      2230     5610     39.750
```

6.7 Combining a Grand Total with the Original Data



You can use the MEANS procedure to create a data set containing a grand total rather than BY group totals. But you cannot use a MERGE statement to combine a grand total with the original data because there is no common variable to merge by. Luckily, there is another way. You can use two SET statements like this:

```
DATA new-data-set;
   IF _N_ = 1 THEN SET summary-data-set;
   SET original-data-set;
```

In this DATA step, *original-data-set* is the data set with more than one observation (the original data) and *summary-data-set* is the data set with a single observation (the grand total). SAS reads *original-data-set* in a normal SET statement, simply reading the observations in a straightforward way. SAS also reads *summary-data-set* with a SET statement but only in the first iteration of the DATA step (when _N_ equals 1).[1] SAS then retains the values of variables from *summary-data-set* for all observations in *new-data-set*.

This works because variables read with a SET statement are automatically retained. Normally you don't notice this because the retained values are overwritten by the next observation. But in this case the variables from *summary-data-set* are read once at the first iteration of the DATA step and then retained for all other observations. The effect is similar to a RETAIN statement (discussed in section 3.9). This technique can be used any time you want to combine a single observation with many observations, without a common variable.

Example To show how this is different from merging BY group summary statistics with original data, we'll use the same data as in the previous section. A distributor of athletic shoes is considering doing a special promotion for the top-selling styles. The vice-president of marketing asks you to produce a report showing the percentage of total sales for each style. For each style of shoe the raw data file contains the style name, type of exercise, and sales for the last quarter:

```
Max Flight       running 1930
Zip Fit Leather  walking 2250
Zoom Airborne    running 4150
Light Step       walking 1130
Max Step Woven   walking 2230
Zip Sneak        c-train 1190
```

[1] See section 6.14 for an explanation of _N_.

Here is the program:

```
DATA shoes;
   INFILE 'c:\MyRawData\Shoesales.dat';
   INPUT Style $ 1-15 ExerciseType $ Sales;

* Output grand total of sales to a data set and print;
PROC MEANS NOPRINT DATA = shoes;
   VAR Sales;
   OUTPUT OUT = summarydata SUM(Sales) = GrandTotal;
PROC PRINT DATA = summarydata;
   TITLE 'Summary Data Set';

* Combine the grand total with the original data;
DATA shoesummary;
   IF _N_ = 1 THEN SET summarydata;
   SET shoes;
   Percent = Sales / GrandTotal * 100;
PROC PRINT DATA = shoesummary;
   VAR Style ExerciseType Sales GrandTotal Percent;
   TITLE 'Overall Sales Share';
RUN;
```

This program starts with a DATA step to input the raw data. Then PROC MEANS creates an output data set named SUMMARYDATA with one observation containing a variable named GrandTotal, which is equal to the sum of Sales. This will be a grand total because there is no BY or CLASS statement. The second DATA step combines the original data with the grand total using two SET statements and then computes the variable Percent using the grand total data.

The output looks like this:

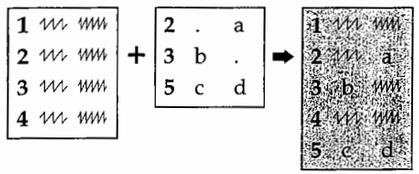
Summary Data Set 1

Obs	_TYPE_	_FREQ_	Grand Total
1	0	6	12880

Overall Sales Share 2

Obs	Style	Exercise Type	Sales	Grand Total	Percent
1	Max Flight	running	1930	12880	14.9845
2	Zip Fit Leather	walking	2250	12880	17.4689
3	Zoom Airborne	running	4150	12880	32.2205
4	Light Step	walking	1130	12880	8.7733
5	Max Step Woven	walking	2230	12880	17.3137
6	Zip Sneak	c-train	1190	12880	9.2391

6.8 Updating a Master Data Set with Transactions



The UPDATE statement is used far less than the MERGE statement, but it is just right for those times when you have a master data set that must be updated with bits of new information. A bank account is a good example of this type of transaction-oriented data, since it is regularly updated with credits and debits.

The UPDATE statement is similar to the MERGE statement, because both combine data sets by matching observations on common variables.[1] However, there are critical differences:

- First, with UPDATE the resulting master data set always has just one observation for each unique value of the common variables. That way, you don't get a new observation for your bank account every time you deposit a paycheck.
- Second, missing values in the transaction data set do not overwrite existing values in the master data set. That way, you are not obliged to enter your address and tax ID number every time you make a withdrawal.

The basic form of the UPDATE statement is

```
DATA master-data-set;
   UPDATE master-data-set transaction-data-set;
   BY variable-list;
```

Here are a few points to remember about the UPDATE statement. You can specify only two data sets: one master and one transaction. Both data sets must be sorted by their common variables. Also, the values of those BY variables must be unique in the master data set. Using the bank example, you could have many transactions for a single account, but only one observation per account in the master data set.

Example A hospital maintains a master database with information about patients. A sample appears below. Each record contains the patient's account number, last name, address, date of birth, sex, insurance code, and the date that patient's information was last updated.

```
620135 Smith    234 Aspen St.      12-21-1975 m CBC 02-16-1998
645722 Miyamoto 65 3rd Ave.        04-03-1936 f MCR 05-30-1999
645739 Jensvold 505 Glendale Ave. 06-15-1960 f HLT 09-23-1993
874329 Kazoyan  76-C La Vista      .          . MCD 01-15-2003
```

Whenever a patient is admitted to the hospital, the admissions staff check the data for that patient. They create a transaction record for every new patient and for any returning patients whose status has changed. Here are three transactions:

```
620135 .        .                  .          . HLT 06-15-2003
874329 .        .                  04-24-1954 m .   06-15-2003
235777 Harman   5656 Land Way      01-18-2000 f MCD 06-15-2003
```

[1] The MODIFY statement is another way to update a master data set. See the SAS Help and Documentation for more information.

The first transaction is for a returning patient whose insurance has changed. The second transaction fills in missing information for a returning patient. The last transaction is for a new patient who must be added to the database.

Since master data sets are updated frequently, they are usually saved as permanent SAS data sets. To make this example more realistic, this program puts the master data into a permanent data set named PATIENTMASTER in the MySASLib directory on the C drive (Windows).

```
LIBNAME perm 'c:\MySASLib';
DATA perm.patientmaster;
   INFILE 'c:\MyRawData\Admit.dat';
   INPUT Account LastName $ 8-16 Address $ 17-34
      BirthDate MMDDYY10. Sex $ InsCode $ 48-50 @52 LastUpdate MMDDYY10.;
RUN;
```

The next program reads the transaction data and sorts them with PROC SORT. Then it adds the transactions to PATIENTMASTER with an UPDATE statement. The master data set is already sorted by Account and, therefore, doesn't need to be sorted again:

```
LIBNAME perm 'c:\MySASLib';
DATA transactions;
   INFILE 'c:\MyRawData\NewAdmit.dat';
   INPUT Account LastName $ 8-16 Address $ 17-34 BirthDate MMDDYY10.
      Sex $ InsCode $ 48-50 @52 LastUpdate MMDDYY10.;
PROC SORT DATA = transactions;
   BY Account;

* Update patient data with transactions;
DATA perm.patientmaster;
   UPDATE perm.patientmaster transactions;
   BY Account;
PROC PRINT DATA = perm.patientmaster;
   FORMAT BirthDate LastUpdate MMDDYY10.;
   TITLE 'Admissions Data';
RUN;
```

The results of the PROC PRINT look like this:

```
                          Admissions Data                          1

                                                          Ins
 Obs Account LastName Address            BirthDate Sex Code LastUpdate

   1 235777 Harman   5656 Land Way       01/18/2000  f  MCD  06/15/2003
   2 620135 Smith    234 Aspen St.       12/21/1975  m  HLT  06/15/2003
   3 645722 Miyamoto 65 3rd Ave.         04/03/1936  f  MCR  05/30/1999
   4 645739 Jensvold 505 Glendale Ave.   06/15/1960  f  HLT  09/23/1993
   5 874329 Kazoyan  76-C La Vista       04/24/1954  m  MCD  06/15/2003
```

6.9 Using SAS Data Set Options

In this book, you have already seen a lot of options. It may help to keep them straight if you realize that the SAS language has three basic types of options: system options, statement options, and data set options. System options have the most global influence, followed by statement options, with data set options having the most limited effect.

System options are those that stay in effect for the duration of your job or session. These options affect how SAS operates, and are usually issued when you invoke SAS or via an OPTIONS statement. System options include the CENTER option, which tells SAS to center all output, and the LINESIZE= option setting the maximum line length for output.[1]

Statement options appear in individual statements and influence how SAS runs that particular DATA or PROC step. The NOPRINT option in PROC MEANS, for example, tells SAS not to produce a printed report. DATA= is a statement option that tells SAS which data set to use for a procedure. You can use DATA= in any procedure that reads a SAS data set. Without it, SAS defaults to the most recently created data set.

In contrast, data set options affect only how SAS reads or writes an individual data set. You can use data set options in DATA steps (in DATA, SET, MERGE, or UPDATE statements) or in PROC steps (in conjuction with a DATA= statement option). To use a data set option, you simply put it between parentheses directly following the data set name. These are the most frequently used data set options:

KEEP = *variable-list*	tells SAS which variables to keep.
DROP = *variable-list*	tells SAS which variables to drop.
RENAME = (*oldvar = newvar*)	tells SAS to rename certain variables.
FIRSTOBS = *n*	tells SAS to start reading at observation *n*.
OBS = *n*	tells SAS to stop reading at observation *n*.
IN = *new-var-name*	creates a temporary variable for tracking whether that data set contributed to the current observation.

Selecting and renaming variables Here are examples of the KEEP=, DROP=, and RENAME= data set options:

```
DATA small;
   SET animals (KEEP = Cat Mouse Rabbit);

PROC PRINT DATA = animals (DROP = Cat Mouse Rabbit);

DATA animals (RENAME = (Cat = Feline Dog = Canine));
   SET animals;

PROC PRINT DATA = animals (RENAME =(Cat = Feline Dog = Canine));
```

[1] Other system options are discussed in section 1.13.

You could probably get by without these options, but they play an important role in fine tuning SAS programs. Data sets, for example, have a way of accumulating unwanted variables. Dropping unwanted variables will make your program run faster and use less disk space. Likewise, when you read a large data set, you often need only a few variables. By using the KEEP= option, you can avoid reading a lot of variables you don't intend to use.

The DROP=, KEEP=, and RENAME= options are similar to the DROP, KEEP, and RENAME statements. However, the statements apply to all data sets named in the DATA statement while the options apply only to the particular data set whose name they follow. Also, the statements are more limited than the options since they can be used only in DATA steps, and apply only to the data set being created. In contrast, the data set options can be used in DATA or PROC steps and can apply to input or output data sets. Please note that these options do not change input data sets; they change only what is read from input data sets.

Selecting observations by observation number You can use the FIRSTOBS= and OBS= data set options together to tell SAS which observations to read from a data set. The options in the following statements tell SAS to read just 20 observations:

```
DATA animals;
   SET animals (FIRSTOBS = 101 OBS = 120);

PROC PRINT DATA = animals (FIRSTOBS = 101 OBS = 120);
```

If you use large data sets, you can save development time by testing your programs with a subset of your data with the FIRSTOBS= and OBS= options.

The FIRSTOBS= and OBS= data set options are similar to statement and system options with the same name. The statement options apply only to raw data files being read with an INFILE statement, whereas the data set options apply only to existing SAS data sets that you read in a DATA or PROC step. The system options apply to all files and data sets. If you use similar system and data set options, the data set option will override the system option for that particular data set.

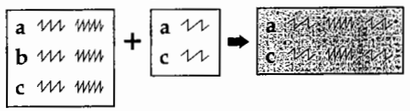
Tracking observations The IN= option is somewhat different from other options covered here. While the other options affect existing variables, IN= creates a new variable. That new variable is temporary and has the name you specify in the option. In this example, SAS would create two temporary variables, one named InAnimals and the other named InHabitat:

```
DATA animals;
   MERGE animals (IN = InAnimals) habitat (IN = InHabitat);
   BY Species;
```

These variables exist only for the duration of the current DATA step and are not added to the data set being created. SAS gives IN= variables a value of 0 if that data set did not contribute to the current observation and a value of 1 if it did. You can use the IN= variable to track, select, or eliminate observations based on the data set of origin. The next section explains the IN= option in more detail.

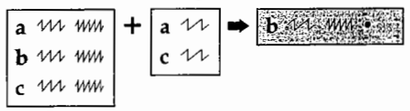
6.10 Tracking and Selecting Observations with the IN= Option

Select matching observations



OR

Select non-matching observations



When you combine two data sets, you can use IN= options to track which of the original data sets contributed to each observation in the new data set. You can think of the IN= option as a sort of tag. Instead of saying "Product of Canada," the tag says something like "Product of data set one." Once you have that information, you can use it in many ways including selecting matching or non-matching observations during a merge.

The IN= data set option can be used any time you read a SAS data set in a DATA step—with SET, MERGE, or UPDATE—but is most often used with MERGE. To use the IN= option, you simply put the option in parentheses directly following the data set you want to track, and specify a name for the IN= variable. The names of IN= variables must follow standard SAS naming conventions—start with a letter or underscore; be 32 characters or fewer in length; and contain only letters, numerals, or underscores.

The DATA step below creates a data set named BOTH by merging two data sets named STATE and COUNTY. Then the IN= options create two variables named InState and InCounty:

```
DATA both;
   MERGE state (IN = InState) county (IN = InCounty);
   BY StateName;
```

Unlike most variables, IN= variables are temporary, existing only during the current DATA step. SAS gives the IN= variables a value of 0 or 1. A value of 1 means that data set did contribute to the current observation, and a value of 0 means the data set did not contribute. Suppose the COUNTY data set above contained no data for Louisiana. (Louisiana has parishes, not counties.) In that case, the BOTH data set would contain one observation for Louisiana which would have a value of 1 for the variable InState and a value of 0 for InCounty because the STATE data set contributed to that observation, but the COUNTY data set did not.

You can use this variable like any other variable in the current DATA step, but it is most often used in subsetting IF or IF-THEN statements such as these:

Subsetting IF:
```
IF InState = 1;
IF InCounty = 0;
IF InState = 1 AND InCounty = 1;
```

IF-THEN:
```
IF InCounty = 1 THEN Origin = 1;
IF InState = 1 THEN State = 'Yes';
```

Example A sporting goods manufacturer wants to send a sales rep to contact all customers who did not place any orders during the third quarter of the year. The company has two data files, one that contains all customers and one that contains all orders placed during the third quarter. To compile a list of customers without orders, you merge the two data sets using the IN= option, and then select customers who had no observations in the orders data set. The customer data file contains the customer number, name, and address. The orders data file

contains the customer number and total price, with one observation for every order placed during the third quarter. Here are samples of the two raw data files:

Customer data

```
101 Murphy's Sports     115 Main St.              102 562.01
102 Sun N Ski           2106 Newberry Ave.        104 254.98
103 Sports Outfitters   19 Cary Way               104 1642.00
104 Cramer & Johnson    4106 Arlington Blvd.      101 3497.56
105 Sports Savers       2708 Broadway             102 385.30
```

Orders data

Here is the program that finds customers who did not place any orders:

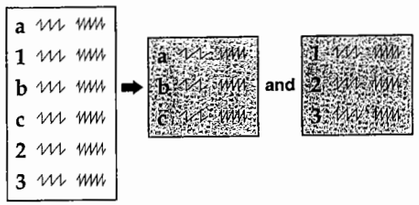
```
DATA customer;
   INFILE 'c:\MyRawData\Address.dat' TRUNCOVER;
   INPUT CustomerNumber Name $ 5-21 Address $ 23-42;
DATA orders;
   INFILE 'c:\MyRawData\OrdersQ3.dat';
   INPUT CustomerNumber Total;
PROC SORT DATA = orders;
   BY CustomerNumber;

* Combine the data sets using the IN= option;
DATA noorders;
   MERGE customer orders (IN = Recent);
   BY CustomerNumber;
   IF Recent = 0;
PROC PRINT DATA = noorders;
   TITLE 'Customers with No Orders in the Third Quarter';
RUN;
```

The customer data are already sorted by customer number and so do not need to be sorted with PROC SORT. The orders data, however, are in the order received and must be sorted by customer number before merging. In the final DATA step, the IN= option creates a variable named Recent, which equals 1 if the ORDERS data set contributed to that observation and 0 if it did not. Then a subsetting IF statement keeps only the observations where Recent is equal to 0—those observations with no orders data. Notice that there is no IN= option on the CUSTOMER data set. Only one IN= option was needed to identify customers who did not place any orders. Here is the list that can be given to sales reps:

```
          Customers with No Orders in the Third Quarter          1

        Customer
 Obs     Number          Name              Address        Total

  1        103     Sports Outfitters   19 Cary Way          .
  2        105     Sports Savers       2708 Broadway        .
```

The values for the variable Total are missing because these customers did not have observations in the ORDERS data set. The variable Recent does not appear in the output because, as a temporary variable, it was not added to the NOORDERS data set.

6.11 Writing Multiple Data Sets Using the OUTPUT Statement



Up to this point, all the DATA steps in this book have created a single data set, except for DATA _NULL_ statements which produce no data set at all. Normally you want to make only one data set in each DATA step. However, there may be times when it is more efficient or more convenient to create multiple data sets in a single DATA step. You can do this by simply putting more than one data set name in your DATA statement. The statement below tells SAS to create three data sets named LIONS, TIGERS, and BEARS:

```
DATA lions tigers bears;
```

If that is all you do, then SAS will write all the observations to all the data sets, and you will have three identical data sets. Normally, of course, you want to create different data sets. You can do that with an OUTPUT statement.

Every DATA step has an implied OUTPUT statement at the end which tells SAS to write the current observation to the output data set before returning to the beginning of the DATA step to process the next observation. You can override this implicit OUTPUT statement with your own OUTPUT statement. The basic form of the OUTPUT statement is

```
OUTPUT data-set-name;
```

If you leave out the data set name then the observation will be written to all data sets named in the DATA statement. OUTPUT statements can be used alone or in IF-THEN or DO-loop processing.

```
IF family = 'Ursidae' THEN OUTPUT bears;
```

Example A local zoo maintains a data base about the feeding of the animals. A portion of the data appears below. For each group of animals the data include the scientific class, the enclosure those animals live in, and whether they get fed in the morning, afternoon, or both:

```
bears      Mammalia E2 both
elephants  Mammalia W3 am
flamingos  Aves     W1 pm
frogs      Amphibia S2 pm
kangaroos  Mammalia N4 am
lions      Mammalia W6 pm
snakes     Reptilia S1 pm
tigers     Mammalia W9 both
zebras     Mammalia W2 am
```

To help with feeding the animals, the following program creates two lists, one for morning feedings and one for afternoon feedings.


```
DATA morning afternoon;
   INFILE 'c:\MyRawData\Zoo.dat';
   INPUT Animal $ 1-9 Class $ 11-18 Enclosure $ FeedTime $;
   IF FeedTime = 'am' THEN OUTPUT morning;
      ELSE IF FeedTime = 'pm' THEN OUTPUT afternoon;
      ELSE IF FeedTime = 'both' THEN OUTPUT;
PROC PRINT DATA = morning;
   TITLE 'Animals with Morning Feedings';
PROC PRINT DATA = afternoon;
   TITLE 'Animals with Afternoon Feedings';
RUN;
```

This DATA step creates two data sets named MORNING and AFTERNOON. Then the IF-THEN/ELSE statements tell SAS which observations to put in each data set. Because the final OUTPUT statement does not specify a data set, SAS adds those observations to both data sets. The log contains these notes saying that SAS read one input file and wrote two data sets:

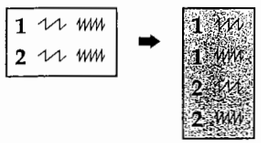
```
NOTE: 9 records were read from the infile 'c:\MyRawData\Zoo.dat'.

NOTE: The data set WORK.MORNING has 5 observations and 4 variables.

NOTE: The data set WORK.AFTERNOON has 6 observations and 4 variables.
```

Here are the two reports, one for each data set:

```
                Animals with Morning Feedings                    1

                                                        Feed
        Obs    Animal       Class       Enclosure      Time

          1    bears        Mammalia       E2          both
          2    elephants    Mammalia       W3          am
          3    kangaroos    Mammalia       N4          am
          4    tigers       Mammalia       W9          both
          5    zebras       Mammalia       W2          am

              Animals with Afternoon Feedings                    2

                                                        Feed
        Obs    Animal       Class       Enclosure      Time

          1    bears        Mammalia       E2          both
          2    flamingos    Aves           W1          pm
          3    frogs        Amphibia       S2          pm
          4    lions        Mammalia       W6          pm
          5    snakes       Reptilia       S1          pm
          6    tigers       Mammalia       W9          both
```

OUTPUT statements have other uses besides writing multiple data sets in a single DATA step and can be used any time you want to explicitly control when SAS writes observations to a data set.

6.12 Making Several Observations from One Using the OUTPUT Statement



Usually SAS writes an observation to a data set at the end of the DATA step, but you can override this default using the OUTPUT statement. If you want to write several observations for each pass through the DATA step, you can put an OUTPUT statement in a DO loop or just use several OUTPUT statements. The OUTPUT statement gives you control over when an observation is written to a SAS data set. If your DATA step doesn't have an OUTPUT statement, then it is implied at the end of the step. Once you put an OUTPUT statement in your DATA step, it is no longer implied, and SAS writes an observation only when it encounters an OUTPUT statement.

Example The following program demonstrates how you can use an OUTPUT statement in a DO loop to generate data. Here we have a mathematical equation ($y=x^2$) and we want to generate data points for later plotting:

```
* Create data for variables x and y;
DATA generate;
   DO x = 1 TO 6;
      y = x ** 2;
      OUTPUT;
   END;
PROC PRINT DATA = generate;
   TITLE 'Generated Data';
RUN;
```

This program has no INPUT or SET statement—so there is only one iteration of the entire DATA step—but it has a DO loop with six iterations. Because the OUTPUT statement is inside the DO loop, an observation is created each time through the loop. Without the OUTPUT statement, SAS would have written only one observation at the end of the DATA step when it reached the implied OUTPUT. The following are the results of the PROC PRINT:

```
                    Generated Data                    1

Obs    x     y

 1     1     1
 2     2     4
 3     3     9
 4     4    16
 5     5    25
 6     6    36
```


Example Here's how you can use OUTPUT statements to create several observations from a single pass through the DATA step. The following data are for ticket sales at three movie theaters. After the month are the theaters' names and sales for all three theaters:

```
Jan Varsity 56723 Downtown 69831 Super-6 70025
Feb Varsity 62137 Downtown 43901 Super-6 81534
Mar Varsity 49982 Downtown 55783 Super-6 69800
```

For the analysis you want to do, you need to have the theater name as one variable and the ticket sales as another variable. The month should be repeated three times, once for each theater.

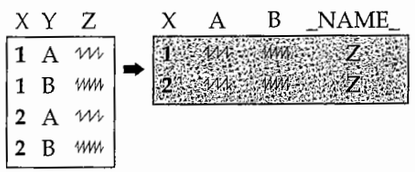
The following program has three INPUT statements all reading from the same raw data file. The first INPUT statement reads values for Month, Location, and Tickets, and then holds the data line using the trailing at sign (@). The OUTPUT statement that follows writes an observation. The next INPUT statement reads the second set of data for Location and Tickets and again holds the data line. Another OUTPUT statement writes another observation. Month still has the same value because it isn't in the second INPUT statement. The last INPUT statement reads the last values for Location and Tickets, this time releasing the data line for the next iteration through the DATA step. The final OUTPUT statement writes the third observation for that iteration of the DATA step. The program has three OUTPUT statements for the three observations created in each iteration of the DATA step:

```
* Create three observations for each data line read
*   using three OUTPUT statements;
DATA theaters;
   INFILE 'c:\MyRawData\Movies.dat';
   INPUT Month $ Location $ Tickets @;
   OUTPUT;
   INPUT Location $ Tickets @;
   OUTPUT;
   INPUT Location $ Tickets;
   OUTPUT;
PROC PRINT DATA = theaters;
   TITLE 'Ticket Sales';
RUN;
```

The following are the results of the PROC PRINT. Notice that there are three observations in the data set for each line in the raw data file and that the value for Month is repeated:

```
                        Ticket Sales                  1

Obs    Month    Location    Tickets

 1      Jan      Varsity      56723
 2      Jan      Downtown     69831
 3      Jan      Super-6      70025
 4      Feb      Varsity      62137
 5      Feb      Downtown     43901
 6      Feb      Super-6      81534
 7      Mar      Varsity      49982
 8      Mar      Downtown     55783
 9      Mar      Super-6      69800
```

6.13 Changing Observations to Variables Using PROC TRANSPOSE



We have already seen ways to combine data sets, create new variables, and sort data. Now, using PROC TRANSPOSE, we will flip data—so get your spatulas ready.

The TRANSPOSE procedure transposes SAS data sets, turning observations into variables or variables into observations. In most cases, to convert observations into variables, you can use the following statements:

```
PROC TRANSPOSE DATA = old-data-set OUT = new-data-set;
   BY variable-list;
   ID variable;
   VAR variable-list;
```

In the PROC TRANSPOSE statement, *old-data-set* refers to the SAS data set you want to transpose, and *new-data-set* is the name of the newly transposed data set.

BY statement You can use the BY statement if you have any grouping variables that you want to keep as variables. These variables are included in the transposed data set, but they are not themselves transposed. The transposed data set will have one observation for each BY level per variable transposed. For example, in the figure above, the variable X is the BY variable. The data set must be sorted by these variables before transposing.

ID statement The ID statement names the variable whose formatted values will become the new variable names. The ID values must occur only once in the data set; or if a BY statement is present, then the values must be unique within BY-groups. If the ID variable is numeric, then the new variable names have an underscore for a prefix (_1 or _2, for example). If you don't use an ID statement, then the new variables will be named COL1, COL2, and so on. In the figure above, the variable Y is the ID variable. Notice how its values are the new variable's names in the transposed data set.

VAR statement The VAR statement names the variables whose values you want to transpose. In the figure above, the variable Z is the VAR variable. SAS creates a new variable, _NAME_, which has as values the names of the variables in the VAR statement. If there is more than one VAR variable, then _NAME_ will have more than one value.

Example Suppose you have the following data about players for minor league baseball teams. You have the team name, player's number, the type of data (salary or batting average), and the entry:

```
Garlics 10 salary 43000
Peaches  8 salary 38000
Garlics 21 salary 51000
Peaches 10 salary 47500
Garlics 10 batavg .281
Peaches  8 batavg .252
Garlics 21 batavg .265
Peaches 10 batavg .301
```


You want to look at the relationship between batting average and salary. To do this, salary and batting average must be variables. The following program reads the raw data into a SAS data set and sorts the data by team and player. Then the data are transposed using PROC TRANSPOSE.

```
DATA baseball;
   INFILE 'c:\MyRawData\Transpos.dat';
   INPUT Team $ Player Type $ Entry;
PROC SORT DATA = baseball;
   BY Team Player;
PROC PRINT DATA = baseball;
   TITLE 'Baseball Data After Sorting and Before Transposing';

* Transpose data so salary and batavg are variables;
PROC TRANSPOSE DATA = baseball OUT = flipped;
   BY Team Player;
   ID Type;
   VAR Entry;
PROC PRINT DATA = flipped;
   TITLE 'Baseball Data After Transposing';
RUN;
```

In the PROC TRANSPOSE step, the BY variables are Team and Player. You want those variables to remain in the data set, and they define the new observations (you want only one observation for each team and player combination). The ID variable is Type, whose values (salary and batavg) will be the new variable names. The variable to be transposed, Entry, is specified in the VAR statement. Notice that its name, Entry, now appears as a value under the variable _NAME_. The TRANSPOSE procedure automatically generates the _NAME_ variable, but in this application it is not very meaningful and could be dropped.

Here are the results:

Baseball Data After Sorting and Before Transposing 1

Obs	Team	Player	Type	Entry
1	Garlics	10	salary	43000.00
2	Garlics	10	batavg	0.28
3	Garlics	21	salary	51000.00
4	Garlics	21	batavg	0.27
5	Peaches	8	salary	38000.00
6	Peaches	8	batavg	0.25
7	Peaches	10	salary	47500.00
8	Peaches	10	batavg	0.30

Baseball Data After Transposing 2

Obs	Team	Player	_NAME_	salary	batavg
1	Garlics	10	Entry	43000	0.281
2	Garlics	21	Entry	51000	0.265
3	Peaches	8	Entry	38000	0.252
4	Peaches	10	Entry	47500	0.301

6.14 Using SAS Automatic Variables

In addition to the variables you create in your SAS data set, SAS creates a few more called automatic variables. You don't ordinarily see these variables because they are temporary and are not saved with your data. But they are available in the DATA step, and you can use them just like you use any variable that you create yourself.

_N_ and _ERROR_ The _N_ and _ERROR_ variables are always available to you in the DATA step. _N_ indicates the number of times SAS has looped through the DATA step. This is not necessarily equal to the observation number, since a simple subsetting IF statement can change the relationship between observation number and the number of iterations of the DATA step. The _ERROR_ variable has a value of 1 if there is a data error for that observation and 0 if there isn't. Things that can cause data errors include invalid data (such as characters in a numeric field), conversion errors (like division by zero), and illegal arguments in functions (including log of zero).

FIRST.*variable* and LAST.*variable* Other automatic variables are available only in special circumstances. The FIRST.*variable* and LAST.*variable* automatic variables are available when you are using a BY statement in a DATA step. The FIRST.*variable* will have a value of 1 when SAS is processing an observation with the first occurrence of a new value for that variable and a value of 0 for the other observations. The LAST.*variable* will have a value of 1 for an observation with the last occurrence of a value for that variable and the value 0 for the other observations.

Example Your hometown is having a walk around the town square to raise money for the library. You have the following data: entry number, age group, and finishing time. (Notice that there is more than one observation per line of data.)

```
54 youth  35.5 21 adult  21.6  6 adult  25.8 13 senior 29.0
38 senior 40.3 19 youth  39.6  3 adult  19.0 25 youth  47.3
11 adult  21.9  8 senior 54.3 41 adult  43.0 32 youth  38.6
```

The first thing you want to do is create a new variable for overall finishing place and print the results. The first part of the following program reads the raw data, and sorts the data by finishing time (Time). Then another DATA step creates the new Place variable and gives it the current value of _N_. The PRINT procedure produces the list of finishers:

```
DATA walkers;
   INFILE 'c:\MyRawData\Walk.dat';
   INPUT Entry AgeGroup $ Time @@;
PROC SORT DATA = walkers;
   BY Time;
* Create a new variable, Place;
DATA ordered;
   SET walkers;
   Place = _N_;
PROC PRINT DATA = ordered;
  TITLE 'Results of Walk';
```



```
PROC SORT DATA = ordered;
   BY AgeGroup Time;
* Keep the first observation in each age group;
DATA winners;
   SET ordered;
   BY AgeGroup;
   IF FIRST.AgeGroup = 1;
PROC PRINT DATA = winners;
   TITLE 'Winners in Each Age Group';
RUN;
```

The second part of this program produces a list of the top finishers in each age category. The ORDERED data set containing the new Place variable is sorted by AgeGroup and Time. In the DATA step, the SET statement reads the ORDERED data set. The BY statement in the DATA step generates the FIRST.AgeGroup and LAST.AgeGroup temporary variables. The subsetting IF statement, IF FIRST.AgeGroup = 1, keeps only the first observation in the BY group. Since the Winners data set is sorted by AgeGroup and Time, the first observation in each BY group is the top finisher of that group.

Here are the results of the two PRINT procedures. The first page shows the data after sorting by Time and including the new variable Place. Notice that the _N_ temporary variable does not appear in the printout. The second page shows the results of the second part of the program—the winners for each age category and their overall place:

Results of Walk 1

Obs	Entry	Age Group	Time	Place
1	3	adult	19.0	1
2	21	adult	21.6	2
3	11	adult	21.9	3
4	6	adult	25.8	4
5	13	senior	29.0	5
6	54	youth	35.5	6
7	32	youth	38.6	7
8	19	youth	39.6	8
9	38	senior	40.3	9
10	41	adult	43.0	10
11	25	youth	47.3	11
12	8	senior	54.3	12

Winners in Each Age Group 2

Obs	Entry	Age Group	Time	Place
1	3	adult	19.0	1
2	13	senior	29.0	5
3	54	youth	35.5	6