Programming and Data Management for IBM SPSS Statistics 19

A Guide for IBM SPSS Statistics and SAS Users

Raynald Levesque and SPSS Inc.

IBM®

Note: Before using this information and the product it supports, read the general information under "Notices" on p. 435.

This document contains proprietary information of SPSS Inc, an IBM Company. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

When you send information to IBM or SPSS, you grant IBM and SPSS a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Preface

Experienced data analysts know that a successful analysis or meaningful report often requires more work in acquiring, merging, and transforming data than in specifying the analysis or report itself. IBM® SPSS® Statistics contains powerful tools for accomplishing and automating these tasks. While much of this capability is available through the graphical user interface, many of the most powerful features are available only through command syntax—and you can make the programming features of its command syntax significantly more powerful by adding the ability to combine it with a full-featured programming language. This book offers many examples of the kinds of things that you can accomplish using command syntax by itself and in combination with other programming language.

For SAS Users

If you have more experience with SAS for data management, see Chapter 32 for comparisons of the different approaches to handling various types of data management tasks. Quite often, there is not a simple command-for-command relationship between the two programs, although each accomplishes the desired end.

Acknowledgments

This book reflects the work of many members of the SPSS Inc. staff who have contributed examples here and in Developer Central, as well as that of Raynald Levesque, whose examples formed the backbone of earlier editions and remain important in this edition. We also wish to thank Stephanie Schaller, who provided many sample SAS jobs and helped to define what the SAS user would want to see, as well as Marsha Hollar and Brian Teasley, the authors of the original chapter "IBM® SPSS® Statistics for SAS Programmers."

A Note from Raynald Levesque

It has been a pleasure to be associated with this project from its inception. I have for many years tried to help IBM® SPSS® Statistics users understand and exploit its full potential. In this context, I am thrilled about the opportunities afforded by the Python integration and invite everyone to visit my site at *www.spsstools.net* for additional examples. And I want to express my gratitude to my spouse, Nicole Tousignant, for her continued support and understanding.

Raynald Levesque

Contents

4   File Operations   55

9   *Exporting Data and Results*   *129*

14 Working with Dictionary Information 213

15 Working with Case Data in the Active Dataset 233

32 IBM SPSS Statistics for SAS Programmers 409

Appendix

Overview

This book is divided into several sections:

- **Data management using the IBM® SPSS® Statistics command language.** Although many of these tasks can also be performed with the menus and dialog boxes, some very powerful features are available only with command syntax.

- **Programming with SPSS Statistics and Python.** The SPSS Statistics-Python Integration Plug-In provides the ability to integrate the capabilities of the Python programming language with SPSS Statistics. One of the major benefits of Python is the ability to add **jobwise** flow control to the SPSS Statistics command stream. SPSS Statistics can execute **casewise** conditional actions based on criteria that evaluate each case, but jobwise flow control—such as running different procedures for different variables based on data type or level of measurement, or determining which procedure to run next based on the results of the last procedure—is much more difficult. The Python Plug-In makes jobwise flow control much easier to accomplish. It also provides the ability to operate on output objects—for example, allowing you to customize pivot tables.

- **Programming with SPSS Statistics and R.** The SPSS Statistics-R Integration Plug-In provides the ability to integrate the capabilities of the R statistical programming language with SPSS Statistics. This allows you to take advantage of many statistical routines already available in the R language, plus the ability to write your own routines in R, all from within SPSS Statistics.

- **Extension commands.** Extension commands provide the ability to wrap programs written in Python or R in SPSS Statistics command syntax. Subcommands and keywords specified in the command syntax are first validated and then passed as argument parameters to the underlying Python or R program, which is then responsible for reading any data and generating any results. Extension commands allow users who are proficient in Python or R to share external functions with users of SPSS Statistics command syntax.

- **SPSS Statistics for SAS programmers.** For readers who may be more familiar with the commands in the SAS system, Chapter 32 provides examples that demonstrate how some common data management and programming tasks are handled in both SAS and SPSS Statistics.

Using This Book

This book is intended for use with IBM® SPSS® Statistics release 19 or later. Many examples will work with earlier versions, but some commands and features are not available in earlier releases.

Most of the examples shown in this book are designed as hands-on exercises that you can perform yourself. The command files and data files used in the examples are provided in a Zip file, available from *http://www.spss.com/spss/data_management_book.htm*. All of the sample files are contained in the *examples* folder.

- */examples/commands* contains SPSS Statistics command syntax files.

- */examples/data* contains data files in a variety of formats.

- ■ */examples/python* contains sample Python files.
- ■ */examples/extensions* contains examples of extension commands.

All of the sample command files that contain file access commands assume that you have copied the examples folder to your local hard drive. For example:

```
GET FILE='/examples/data/duplicates.sav'.
SORT CASES BY ID_house(A) ID_person(A) int_date(A) .
AGGREGATE OUTFILE = '/temp/tempdata.sav'
  /BREAK = ID_house ID_person
  /DuplicateCount = N.
```

Many examples, such as the one above, also assume that you have a */temp* folder for writing temporary files.

   Python files from */examples/python* should be copied to your Python *site-packages* directory. The location of this directory depends on your platform. Following are the locations for Python 2.6:

- ■ For Windows users, the *site-packages* directory is located in the *Lib* directory under the Python 2.6 installation directory—for example, *C:\Python26\Lib\site-packages*.
- ■ For Mac OS X 10.4 (Tiger) and 10.5 (Leopard) users, the *site-packages* directory is located at */Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages*.
- ■ For UNIX users (includes SPSS Statistics for Linux and SPSS Statistics Server for UNIX), the *site-packages* directory is located in the */lib/python2.6/* directory under the Python 2.6 installation directory—for example, */usr/local/python26/lib/python2.6/site-packages*.

Documentation Resources

The *IBM® SPSS® Statistics Core System User's Guide* documents the data management tools available through the graphical user interface. The material is similar to that available in the Help system.

   The *SPSS Statistics Command Syntax Reference*, which is installed as a PDF file with the SPSS Statistics system, is a complete guide to the specifications for each command. The guide provides many examples illustrating individual commands. It has only a few extended examples illustrating how commands can be combined to accomplish the kinds of tasks that analysts frequently encounter. Sections of the *SPSS Statistics Command Syntax Reference* that are of particular interest include:

- ■ The appendix "Defining Complex Files," which covers the commands specifically intended for reading common types of complex files.
- ■ The INPUT PROGRAM–END INPUT PROGRAM command, which provides rules for working with input programs.

All of the command syntax documentation is also available in the Help system. If you type a command name or place the cursor inside a command in a syntax window and press F1, you will be taken directly to the help for that command.

Part I:
Data Management

Best Practices and Efficiency Tips

If you haven't worked with IBM® SPSS® Statistics command syntax before, you will probably start with simple jobs that perform a few basic tasks. Since it is easier to develop good habits while working with small jobs than to try to change bad habits once you move to more complex situations, you may find the information in this chapter helpful.

Some of the practices suggested in this chapter are particularly useful for large projects involving thousands of lines of code, many data files, and production jobs run on a regular basis and/or on multiple data sources.

Working with Command Syntax

You don't need to be a programmer to write command syntax, but there are a few basic things you should know. A detailed introduction to command syntax is available in the "Universals" section in the *Command Syntax Reference*.

Creating Command Syntax Files

An command file is a simple text file. You can use any text editor to create a command syntax file, but IBM® SPSS® Statistics provides a number of tools to make your job easier. Most features available in the graphical user interface have command syntax equivalents, and there are several ways to reveal this underlying command syntax:

- **Use the Paste button.** Make selections from the menus and dialog boxes, and then click the Paste button instead of the OK button. This will paste the underlying commands into a command syntax window.

- **Record commands in the log.** Select Display commands in the log on the Viewer tab in the Options dialog box (Edit menu > Options), or run the command SET PRINTBACK ON. As you run analyses, the commands for your dialog box selections will be recorded and displayed in the log in the Viewer window. You can then copy and paste the commands from the Viewer into a syntax window or text editor. This setting persists across sessions, so you have to specify it only once.

- **Retrieve commands from the journal file.** Most actions that you perform in the graphical user interface (and all commands that you run from a command syntax window) are automatically recorded in the journal file in the form of command syntax. The default name of the journal file is *statistics.jnl*. The default location varies, depending on your operating system. Both the name and location of the journal file are displayed on the General tab in the Options dialog box (Edit > Options).

- **Use auto-complete in the Syntax Editor to build command syntax interactively.** Starting with version 17.0, the built-in Syntax Editor contains many tools to help you build and debug command syntax.

Using the Syntax Editor to Build Commands

The Syntax Editor provides assistance in the form of auto-completion of commands, subcommands, keywords, and keyword values. By default, you are prompted with a context-sensitive list of available terms. You can display the list on demand by pressing CTRL+SPACEBAR and you can close the list by pressing the ESC key.

The Auto Complete menu item on the Tools menu toggles the automatic display of the auto-complete list on or off. You can also enable or disable automatic display of the list from the Syntax Editor tab in the Options dialog box. Toggling the Auto Complete menu item overrides the setting on the Options dialog but does not persist across sessions.

Figure 2-1
Auto-complete in Syntax Editor
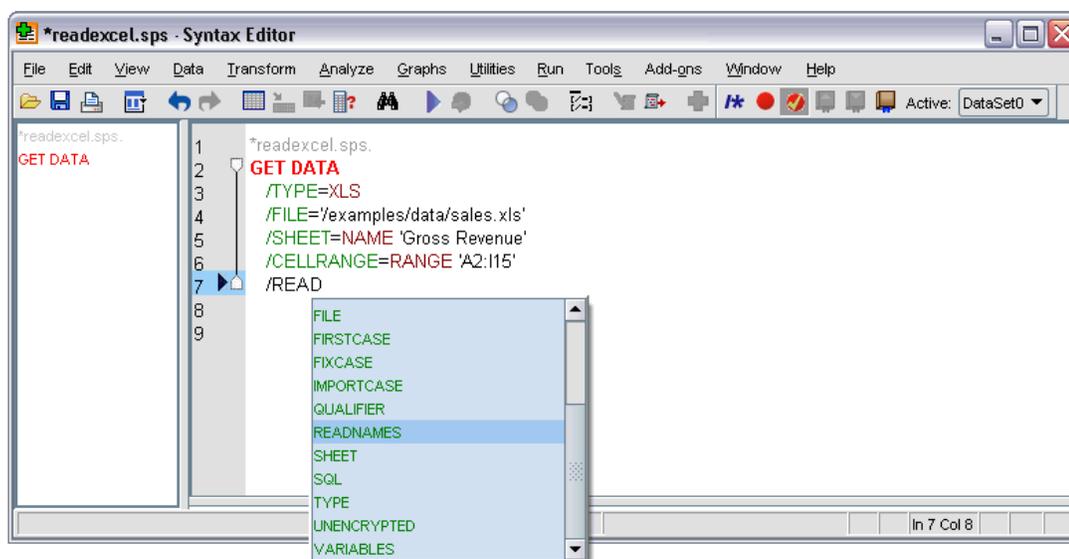
Running Commands

Once you have a set of commands, you can run the commands in a number of ways:

■ Highlight the commands that you want to run in a command syntax window and click the Run button.

■ Invoke one command file from another with the INCLUDE or INSERT command. For more information, see the topic "Using INSERT with a Master Command Syntax File" on p. 15.

■ Use the Production Facility to create production jobs that can run unattended and even start unattended (and automatically) using common scheduling software. See the Help system for more information about the Production Facility.

■ Use IBM® SPSS® Statistics Batch Facility (available only with the server version) to run command files from a command line and automatically route results to different output destinations in different formats. See the SPSS Statistics Batch Facility documentation supplied with the SPSS Statistics server software for more information.

Syntax Rules

- Commands run from a command syntax window during a typical IBM® SPSS® Statistics session must follow the **interactive** command syntax rules.

- Commands files run via SPSS Statistics Batch Facility or invoked via the INCLUDE command must follow the **batch** command syntax rules.

Interactive Rules

The following rules apply to command specifications in interactive mode:

- Each command must start on a new line. Commands can begin in any column of a command line and continue for as many lines as needed. The exception is the END DATA command, which must begin in the first column of the first line after the end of data.

- Each command should end with a period as a command terminator. It is best to omit the terminator on BEGIN DATA, however, so that inline data are treated as one continuous specification.

- The command terminator must be the last nonblank character in a command.

- In the absence of a period as the command terminator, a blank line is interpreted as a command terminator.

Note: For compatibility with other modes of command execution (including command files run with INSERT or INCLUDE commands in an interactive session), each line of command syntax should not exceed 256 bytes.

Batch Rules

The following rules apply to command specifications in batch mode:

- All commands in the command file must begin in column 1. You can use plus (+) or minus (–) signs in the first column if you want to indent the command specification to make the command file more readable.

- If multiple lines are used for a command, column 1 of each continuation line must be blank.

- Command terminators are optional.

- A line cannot exceed 256 bytes; any additional characters are truncated.

Protecting the Original Data

The original data file should be protected from modifications that may alter or delete original variables and/or cases. If the original data are in an external file format (for example, text, Excel, or database), there is little risk of accidentally overwriting the original data while working in IBM® SPSS® Statistics. However, if the original data are in SPSS Statistics data files (*.sav*), there are many transformation commands that can modify or destroy the data, and it is not difficult to inadvertently overwrite the contents of a data file in SPSS Statistics format. Overwriting the original data file may result in a loss of data that cannot be retrieved.

There are several ways in which you can protect the original data, including:

- Storing a copy in a separate location, such as on a CD, that can't be overwritten.

- Using the operating system facilities to change the read-write property of the file to read-only. If you aren't familiar with how to do this in the operating system, you can choose Mark File Read Only from the File menu or use the PERMISSIONS subcommand on the SAVE command.

The ideal situation is then to load the original (protected) data file into SPSS Statistics and do *all* data transformations, recoding, and calculations using SPSS Statistics. The objective is to end up with one or more command syntax files that start from the original data and produce the required results without any manual intervention.

Do Not Overwrite Original Variables

It is often necessary to recode or modify original variables, and it is good practice to assign the modified values to new variables and keep the original variables unchanged. For one thing, this allows comparison of the initial and modified values to verify that the intended modifications were carried out correctly. The original values can subsequently be discarded if required.

Example

```
*These commands overwrite existing variables.
COMPUTE var1=var1*2.
RECODE var2 (1 thru 5 = 1) (6 thru 10 = 2).
*These commands create new variables.
COMPUTE var1_new=var1*2.
RECODE var2 (1 thru 5 = 1) (6 thru 10 = 2)(ELSE=COPY)
  /INTO var2_new.
```

- The difference between the two COMPUTE commands is simply the substitution of a new variable name on the left side of the equals sign.

- The second RECODE command includes the INTO subcommand, which specifies a new variable to receive the recoded values of the original variable. ELSE=COPY makes sure that any values not covered by the specified ranges are preserved.

Using Temporary Transformations

You can use the TEMPORARY command to temporarily transform existing variables for analysis. The temporary transformations remain in effect through the first command that reads the data (for example, a statistical procedure), after which the variables revert to their original values.

Example

```
*temporary.sps.
DATA LIST FREE /var1 var2.
BEGIN DATA
1 2
3 4
5 6
7 8
9 10
END DATA.
TEMPORARY.
```

```
COMPUTE var1=var1+ 5.
RECODE var2 (1 thru 5=1) (6 thru 10=2).
FREQUENCIES
  /VARIABLES=var1 var2
  /STATISTICS=MEAN STDDEV MIN MAX.
DESCRIPTIVES
  /VARIABLES=var1 var2
  /STATISTICS=MEAN STDDEV MIN MAX.
```

■ The transformed values from the two transformation commands that follow the TEMPORARY command will be used in the FREQUENCIES procedure.

■ The original data values will be used in the subsequent DESCRIPTIVES procedure, yielding different results for the same summary statistics.

Under some circumstances, using TEMPORARY will improve the efficiency of a job when short-lived transformations are appropriate. Ordinarily, the results of transformations are written to the virtual active file for later use and eventually are merged into the saved IBM® SPSS® Statistics data file. However, temporary transformations will not be written to disk, assuming that the command that concludes the temporary state is not otherwise doing this, saving both time and disk space. (TEMPORARY followed by SAVE, for example, would write the transformations.)

If many temporary variables are created, not writing them to disk could be a noticeable saving with a large data file. However, some commands require two or more passes of the data. In this situation, the temporary transformations are recalculated for the second or later passes. If the transformations are lengthy and complex, the time required for repeated calculation might be greater than the time saved by not writing the results to disk. Experimentation may be required to determine which approach is more efficient.

Using Temporary Variables

For transformations that require intermediate variables, use scratch (temporary) variables for the intermediate values. Any variable name that begins with a pound sign (#) is treated as a scratch variable that is discarded at the end of the series of transformation commands when IBM® SPSS® Statistics encounters an EXECUTE command or other command that reads the data (such as a statistical procedure).

Example

```
*scratchvar.sps.
DATA LIST FREE / var1.
BEGIN DATA
1 2 3 4 5
END DATA.
COMPUTE factor=1.
LOOP #tempvar=1 TO var1.
- COMPUTE factor=factor * #tempvar.
END LOOP.
EXECUTE.
```

Figure 2-2
Result of loop with scratch variable

- The loop structure computes the factorial for each value of *var1* and puts the factorial value in the variable *factor*.

- The scratch variable *#tempvar* is used as an index variable for the loop structure.

- For each case, the COMPUTE command is run iteratively up to the value of *var1*.

- For each iteration, the current value of the variable *factor* is multiplied by the current loop iteration number stored in *#tempvar*.

- The EXECUTE command runs the transformation commands, after which the scratch variable is discarded.

The use of scratch variables doesn't technically "protect" the original data in any way, but it does prevent the data file from getting cluttered with extraneous variables. If you need to remove temporary variables that still exist after reading the data, you can use the DELETE VARIABLES command to eliminate them.

Use EXECUTE Sparingly

IBM® SPSS® Statistics is designed to work with large data files. Since going through every case of a large data file takes time, the software is also designed to minimize the number of times it has to read the data. Statistical and charting procedures always read the data, but most transformation commands (for example, COMPUTE, RECODE, COUNT, SELECT IF) do not require a separate data pass.

The default behavior of the graphical user interface, however, is to read the data for each separate transformation so that you can see the results in the Data Editor immediately. Consequently, every transformation command generated from the dialog boxes is followed by an EXECUTE command. So if you create command syntax by pasting from dialog boxes or copying from the log or journal, your command syntax may contain a large number of superfluous EXECUTE commands that can significantly increase the processing time for very large data files.

In most cases, you can remove virtually all of the auto-generated EXECUTE commands, which will speed up processing, particularly for large data files and jobs that contain many transformation commands.

To turn off the automatic, immediate execution of transformations and the associated pasting of EXECUTE commands:

▶ From the menus, choose:
Edit > Options...

▶ Click the Data tab.

▶ Select Calculate values before used.

Lag Functions

One notable exception to the above rule is transformation commands that contain lag functions. In a series of transformation commands without any intervening EXECUTE commands or other commands that read the data, lag functions are calculated after all other transformations, regardless of command order. While this might not be a consideration most of the time, it requires special consideration in the following cases:

■ The lag variable is also used in any of the other transformation commands.

■ One of the transformations selects a subset of cases and deletes the unselected cases, such as SELECT IF or SAMPLE.

Example

```
*lagfunctions.sps.
*create some data.
DATA LIST FREE /var1.
BEGIN DATA
1 2 3 4 5
END DATA.
COMPUTE var2=var1.
*******************************.
*Lag without intervening EXECUTE.
COMPUTE lagvar1=LAG(var1).
COMPUTE var1=var1*2.
EXECUTE.
*******************************.
*Lag with intervening EXECUTE.
COMPUTE lagvar2=LAG(var2).
EXECUTE.
COMPUTE var2=var2*2.
EXECUTE.
```

Figure 2-3
Results of lag functions displayed in Data Editor

- Although *var1* and *var2* contain the same data values, *lagvar1* and *lagvar2* are very different from each other.

- Without an intervening EXECUTE command, *lagvar1* is based on the transformed values of *var1*.

- With the EXECUTE command between the two transformation commands, the value of *lagvar2* is based on the original value of *var2*.

- Any command that reads the data will have the same effect as the EXECUTE command. For example, you could substitute the FREQUENCIES command and achieve the same result.

In a similar fashion, if the set of transformations includes a command that selects a subset of cases and deletes unselected cases (for example, SELECT IF), lags will be computed after the case selection. You will probably want to avoid case selection criteria based on lag values—unless you EXECUTE the lags first.

Starting with version 17.0, you can use the SHIFT VALUES command to calculate both lags and leads. SHIFT VALUES is a procedure that reads the data, resulting in execution of any pending transformations. This eliminates the potential pitfalls you might encounter with the LAG function.

Using $CASENUM to Select Cases

The value of the system variable *$CASENUM* is dynamic. If you change the sort order of cases, the value of *$CASENUM* for each case changes. If you delete the first case, the case that formerly had a value of 2 for this system variable now has the value 1. Using the value of *$CASENUM* with the SELECT IF command can be a little tricky because SELECT IF deletes each unselected case, changing the value of *$CASENUM* for all remaining cases.

For example, a SELECT IF command of the general form:

```
SELECT IF ($CASENUM > [positive value]).
```

will delete all cases because regardless of the value specified, the value of *$CASENUM* for the current case will never be greater than 1. When the first case is evaluated, it has a value of 1 for *$CASENUM* and is therefore deleted because it doesn't have a value greater than the specified positive value. The erstwhile second case then becomes the first case, with a value of 1, and is consequently also deleted, and so on.

The simple solution to this problem is to create a new variable equal to the original value of *$CASENUM*. However, command syntax of the form:

```
COMPUTE CaseNumber=$CASENUM.
SELECT IF (CaseNumber > [positive value]).
```

will still delete all cases because each case is deleted before the value of the new variable is computed. The correct solution is to insert an EXECUTE command between COMPUTE and SELECT IF, as in:

```
COMPUTE CaseNumber=$CASENUM.
EXECUTE.
SELECT IF (CaseNumber > [positive value]).
```

MISSING VALUES Command

If you have a series of transformation commands (for example, COMPUTE, IF, RECODE) followed by a MISSING VALUES command that involves the same variables, you may want to place an EXECUTE statement before the MISSING VALUES command. This is because the MISSING VALUES command changes the dictionary before the transformations take place.

Example

```
IF (x = 0) y = z*2.
MISSING VALUES x (0).
```

The cases where x = 0 would be considered user-missing on *x*, and the transformation of *y* would not occur. Placing an EXECUTE before MISSING VALUES allows the transformation to occur before 0 is assigned missing status.

WRITE and XSAVE Commands

In some circumstances, it may be necessary to have an EXECUTE command after a WRITE or an XSAVE command. For more information, see the topic "Using XSAVE in a Loop to Build a Data File" in Chapter 8 on p. 126.

Using Comments

It is always a good practice to include explanatory comments in your code. You can do this in several ways:

```
COMMENT Get summary stats for scale variables.
* An asterisk in the first column also identifies comments.
FREQUENCIES
  VARIABLES=income ed reside
  /FORMAT=LIMIT(10) /*avoid long frequency tables
  /STATISTICS=MEAN /*arithmetic average*/ MEDIAN.
* A macro name like !mymacro in this comment may invoke the macro.
/* A macro name like !mymacro in this comment will not invoke the macro*/.
```

- The first line of a comment can begin with the keyword COMMENT or with an asterisk (*).
- Comment text can extend for multiple lines and can contain any characters. The rules for continuation lines are the same as for other commands. Be sure to terminate a comment with a period.
- Use /* and */ to set off a comment within a command.
- The closing */ is optional when the comment is at the end of the line. The command can continue onto the next line just as if the inserted comment were a blank.
- To ensure that comments that refer to macros by name don't accidently invoke those macros, use the /* [comment text] */ format.

Using SET SEED to Reproduce Random Samples or Values

When doing research involving random numbers—for example, when randomly assigning cases to experimental treatment groups—you should explicitly set the random number seed value if you want to be able to reproduce the same results.

The random number generator is used by the SAMPLE command to generate random samples and is used by many distribution functions (for example, NORMAL, UNIFORM) to generate distributions of random numbers. The generator begins with a **seed**, a large integer. Starting with the same seed, the system will repeatedly produce the same sequence of numbers and will select the same sample from a given data file. At the start of each session, the seed is set to a value that may vary or may be fixed, depending on your current settings. The seed value changes each time a series of transformations contains one or more commands that use the random number generator.

Example

To repeat the same random distribution within a session or in subsequent sessions, use SET SEED before each series of transformations that use the random number generator to explicitly set the seed value to a constant value.

```
*set_seed.sps.
GET FILE = '/examples/data/onevar.sav'.
SET SEED = 123456789.
SAMPLE .1.
LIST.
GET FILE = '/examples/data/onevar.sav'.
SET SEED = 123456789.
SAMPLE .1.
LIST.
```

- Before the first sample is taken the first time, the seed value is explicitly set with SET SEED.
- The LIST command causes the data to be read and the random number generator to be invoked once for each original case. The result is an updated seed value.
- The second time the data file is opened, SET SEED sets the seed to the same value as before, resulting in the same sample of cases.
- Both SET SEED commands are required because you aren't likely to know what the initial seed value is unless you set it yourself.

Note: This example opens the data file before each SAMPLE command because successive SAMPLE commands are *cumulative* within the active dataset.

SET SEED versus SET MTINDEX

There are two random number generators, and SET SEED sets the starting value for only the default random number generator (SET RNG=MC). If you are using the newer Mersenne Twister random number generator (SET RNG=MT), the starting value is set with SET MTINDEX.

Divide and Conquer

A time-proven method of winning the battle against programming bugs is to split the tasks into separate, manageable pieces. It is also easier to navigate around a syntax file of 200–300 lines than one of 2,000–3,000 lines.

Therefore, it is good practice to break down a program into separate stand-alone files, each performing a specific task or set of tasks. For example, you could create separate command syntax files to:

- Prepare and standardize data.
- Merge data files.
- Perform tests on data.
- Report results for different groups (for example, gender, age group, income category).

Using the INSERT command and a master command syntax file that specifies all of the other command files, you can partition all of these tasks into separate command files.

Using INSERT with a Master Command Syntax File

The INSERT command provides a method for linking multiple syntax files together, making it possible to reuse blocks of command syntax in different projects by using a "master" command syntax file that consists primarily of INSERT commands that refer to other command syntax files.

Example

```
INSERT FILE = "/examples/data/prepare data.sps" CD=YES.
INSERT FILE = "combine data.sps".
INSERT FILE = "do tests.sps".
INSERT FILE = "report groups.sps".
```

- Each INSERT command specifies a file that contains command syntax.
- By default, inserted files are read using **interactive** syntax rules, and each command should end with a period.
- The first INSERT command includes the additional specification CD=YES. This changes the working directory to the directory included in the file specification, making it possible to use relative (or no) paths on the subsequent INSERT commands.

INSERT versus INCLUDE

INSERT is a newer, more powerful and flexible alternative to INCLUDE. Files included with INCLUDE must always adhere to batch syntax rules, and command processing stops when the first error in an included file is encountered. You can effectively duplicate the INCLUDE behavior with SYNTAX=BATCH and ERROR=STOP on the INSERT command.

Defining Global Settings

In addition to using INSERT to create modular master command syntax files, you can define global settings that will enable you to use those same command files for different reports and analyses.

Example

You can create a separate command syntax file that contains a set of FILE HANDLE commands that define file locations and a set of macros that define global variables for client name, output language, and so on. When you need to change any settings, you change them once in the global definition file, leaving the bulk of the command syntax files unchanged.

```
*define_globals.sps.
FILE HANDLE data /NAME='/examples/data'.
FILE HANDLE commands /NAME='/examples/commands'.
FILE HANDLE spssdir /NAME='/program files/spssinc/statistics'.
FILE HANDLE tempdir /NAME='d:/temp'.

DEFINE !enddate()DATE.DMY(1,1,2004)!ENDDEFINE.
DEFINE !olang()English!ENDDEFINE.
DEFINE !client()"ABC Inc"!ENDDEFINE.
DEFINE !title()TITLE !client.!ENDDEFINE.
```

- The first two FILE HANDLE commands define the paths for the data and command syntax files. You can then use these file handles instead of the full paths in any file specifications.

- The third FILE HANDLE command contains the path to the IBM® SPSS® Statistics folder. This path can be useful if you use any of the command syntax or script files that are installed with SPSS Statistics.

- The last FILE HANDLE command contains the path of a temporary folder. It is very useful to define a temporary folder path and use it to save any intermediary files created by the various command syntax files making up the project. The main purpose of this is to avoid crowding the data folders with useless files, some of which might be very large. Note that here the temporary folder resides on the *D* drive. When possible, it is more efficient to keep the temporary and main folders on different hard drives.

- The DEFINE-!ENDDEFINE structures define a series of macros. This example uses simple string substitution macros, where the defined strings will be substituted wherever the macro names appear in subsequent commands during the session.

- !enddate contains the end date of the period covered by the data file. This can be useful to calculate ages or other duration variables as well as to add footnotes to tables or graphs.

- !olang specifies the output language.

- !client contains the client's name. This can be used in titles of tables or graphs.

- !title specifies a TITLE command, using the value of the macro *!client* as the title text.

The master command syntax file might then look something like this:

```
INSERT FILE = "/examples/commands/define_globals.sps".
!title.
INSERT FILE = "/data/prepare data.sps".
INSERT FILE = "/commands/combine data.sps".
INSERT FILE = "/commands/do tests.sps".
INCLUDE FILE = "/commands/report groups.sps".
```

- The first INSERT runs the command syntax file that defines all of the global settings. This needs to be run before any commands that invoke the macros defined in that file.

- `!title` will print the client's name at the top of each page of output.
- `"data"` and `"commands"` in the remaining `INSERT` commands will be expanded to `"/examples/data"` and `"/examples/commands"`, respectively.

Note: Using absolute paths or file handles that represent those paths is the most reliable way to make sure that SPSS Statistics finds the necessary files. Relative paths may not work as you might expect, since they refer to the current working directory, which can change frequently. You can also use the `CD` command or the `CD` keyword on the `INSERT` command to change the working directory.

Getting Data into IBM SPSS Statistics

Before you can work with data in IBM® SPSS® Statistics, you need some data to work with. There are several ways to get data into the application:

- Open a data file that has already been saved in SPSS Statistics format.
- Enter data manually in the Data Editor.
- Read a data file from another source, such as a database, text data file, spreadsheet, SAS, or Stata.

Opening SPSS Statistics data files is simple, and manually entering data in the Data Editor is not likely to be your first choice, particularly if you have a large amount of data. This chapter focuses on how to read data files created and saved in other applications and formats.

Getting Data from Databases

IBM® SPSS® Statistics relies primarily on ODBC (open database connectivity) to read data from databases. ODBC is an open standard with versions available on many platforms, including Windows, UNIX, Linux, and Macintosh.

Installing Database Drivers

You can read data from any database format for which you have a database driver. In local analysis mode, the necessary drivers must be installed on your local computer. In distributed analysis mode (available with the Server version), the drivers must be installed on the remote server.

ODBC database drivers are available for a wide variety of database formats, including:

- Access
- Btrieve
- DB2
- dBASE
- Excel
- FoxPro
- Informix
- Oracle
- Paradox
- Progress
- SQL Base
- SQL Server
- Sybase

For Windows and Linux operating systems, many of these drivers can be installed by installing the Data Access Pack. You can install the Data Access Pack from the AutoPlay menu on the installation DVD.

Before you can use the installed database drivers, you may also need to configure the drivers. For the Data Access Pack, installation instructions and information on configuring data sources are located in the *Installation Instructions* folder on the installation DVD.

OLE DB

Starting with release 14.0, some support for OLE DB data sources is provided.

To access OLE DB data sources (available only on Microsoft Windows operating systems), you must have the following items installed:

- .NET framework. To obtain the most recent version of the .NET framework, go to *http://www.microsoft.com/net*.
- IBM® SPSS® Data Collection Survey Reporter Developer Kit. For information on obtaining a compatible version of SPSS Survey Reporter Developer Kit, go to support.spss.com (*http://support.spss.com*).

The following limitations apply to OLE DB data sources:

- Table joins are not available for OLE DB data sources. You can read only one table at a time.
- You can add OLE DB data sources only in local analysis mode. To add OLE DB data sources in distributed analysis mode on a Windows server, consult your system administrator.
- In distributed analysis mode (available with IBM® SPSS® Statistics Server), OLE DB data sources are available only on Windows servers, and both .NET and SPSS Survey Reporter Developer Kit must be installed on the server.

Database Wizard

It's probably a good idea to use the Database Wizard (File > Open Database) the first time you retrieve data from a database source. At the last step of the wizard, you can paste the equivalent commands into a command syntax window. Although the SQL generated by the wizard tends to be overly verbose, it also generates the CONNECT string, which you might never figure out without the wizard.

Reading a Single Database Table

IBM® SPSS® Statistics reads data from databases by reading database tables. You can read information from a single table or merge data from multiple tables in the same database. A single database table has basically the same two-dimensional structure as a data file in SPSS Statistics format: records are cases and fields are variables. So, reading a single table can be very simple.

Example

This example reads a single table from an Access database. It reads all records and fields in the table.

```
*access1.sps.
GET DATA /TYPE=ODBC /CONNECT=
 'DSN=Microsoft Access;DBQ=c:\examples\data\dm_demo.mdb;'+
 ' DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
 /SQL = 'SELECT  * FROM  CombinedTable'.
EXECUTE.
```

- The GET DATA command is used to read the database.

- TYPE=ODBC indicates that an ODBC driver will be used to read the data. This is required for reading data from any database, and it can also be used for other data sources with ODBC drivers, such as Excel workbooks. For more information, see the topic "Reading Multiple Worksheets" on p. 30.

- CONNECT identifies the data source. For this example, the CONNECT string was copied from the command syntax generated by the Database Wizard. The entire string must be enclosed in single or double quotes. In this example, we have split the long string onto two lines using a plus sign (+) to combine the two strings.

- The SQL subcommand can contain any SQL statements supported by the database format. Each line must be enclosed in single or double quotes.

- SELECT * FROM CombinedTable reads all of the fields (columns) and all records (rows) from the table named *CombinedTable* in the database.

- Any field names that are not valid variable names are automatically converted to valid variable names, and the original field names are used as variable labels. In this database table, many of the field names contain spaces, which are removed in the variable names.

Figure 3-1
Database field names converted to valid variable names

Example

Now we'll read the same database table—except this time, we'll read only a subset of fields and records.

```
*access2.sps.
```

```
GET DATA /TYPE=ODBC /CONNECT=
 'DSN=MS Access Database;DBQ=C:\examples\data\dm_demo.mdb;'+
 'DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
 /SQL =
 'SELECT  Age,  Education,  [Income Category]'
 ' FROM CombinedTable'
 ' WHERE ([Marital Status] <> 1 AND Internet = 1 )'.
EXECUTE.
```

■ The SELECT clause explicitly specifies only three fields from the file; so, the active dataset will contain only three variables.

■ The WHERE clause will select only records where the value of the *Marital Status* field is not 1 and the value of the *Internet* field is 1. In this example, that means only unmarried people who have Internet service will be included.

Two additional details in this example are worth noting:

■ The field names *Income Category* and *Marital Status* are enclosed in brackets. Since these field names contain spaces, they must be enclosed in brackets or quotes. Since single quotes are already being used to enclose each line of the SQL statement, the alternative to brackets here would be double quotes.

■ We've put the FROM and WHERE clauses on separate lines to make the code easier to read; however, in order for this command to be read properly, each of those lines also has a blank space between the starting single quote and the first word on the line. When the command is processed, all of the lines of the SQL statement are merged together in a very literal fashion. Without the space before WHERE, the program would attempt to read a table named *CombinedTableWhere*, and an error would result. As a general rule, you should probably insert a blank space between the quotation mark and the first word of each continuation line.

Reading Multiple Tables

You can combine data from two or more database tables by "joining" the tables. The active dataset can be constructed from more than two tables, but each "join" defines a relationship between only two of those tables:

■ **Inner join.** Records in the two tables with matching values for one or more specified fields are included. For example, a unique ID value may be used in each table, and records with matching ID values are combined. Any records without matching identifier values in the other table are omitted.

■ **Left outer join.** All records from the first table are included regardless of the criteria used to match records.

■ **Right outer join.** Essentially the opposite of a left outer join. So, the appropriate one to use is basically a matter of the order in which the tables are specified in the SQL SELECT clause.

Example

In the previous two examples, all of the data resided in a single database table. But what if the data were divided between two tables? This example merges data from two different tables: one containing demographic information for survey respondents and one containing survey responses.

```
*access_multtables1.sps.
GET DATA /TYPE=ODBC /CONNECT=
 'DSN=MS Access Database;DBQ=C:\examples\data\dm_demo.mdb;'+
  'DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
 /SQL =
 'SELECT  * FROM DemographicInformation, SurveyResponses'
 ' WHERE DemographicInformation.ID=SurveyResponses.ID'.
EXECUTE.
```

- The SELECT clause specifies all fields from both tables.

- The WHERE clause matches records from the two tables based on the value of the *ID* field in both tables. Any records in either table without matching *ID* values in the other table are excluded.

- The result is an inner join in which only records with matching *ID* values in both tables are included in the active dataset.

Example

In addition to one-to-one matching, as in the previous inner join example, you can also merge tables with a one-to-many matching scheme. For example, you could match a table in which there are only a few records representing data values and associated descriptive labels with values in a table containing hundreds or thousands of records representing survey respondents.

In this example, we read data from an SQL Server database, using an outer join to avoid omitting records in the larger table that don't have matching identifier values in the smaller table.

```
*sqlserver_outer_join.sps.
GET DATA /TYPE=ODBC
 /CONNECT= 'DSN=SQLServer;UID=;APP=SPSS Statistics;'
  'WSID=USERLAP;Network=DBMSSOCN;Trusted_Connection=Yes'
 /SQL =
 'SELECT  SurveyResponses.ID,  SurveyResponses.Internet,'
 ' [Value Labels].[Internet Label]'
 ' FROM SurveyResponses LEFT OUTER JOIN [Value Labels]'
 ' ON SurveyResponses.Internet'
 ' = [Value Labels].[Internet Value]'.
```

Figure 3-2
SQL Server tables to be merged with outer join

Figure 3-3
Active dataset in IBM SPSS Statistics

■ `FROM SurveyResponses LEFT OUTER JOIN [Value Labels]` will include all records from the table *SurveyResponses* even if there are no records in the *Value Labels* table that meet the matching criteria.

■ `ON SurveyResponses.Internet = [Value Labels].[Internet Value]` matches records based on the value of the field *Internet* in the table *SurveyResponses* and the value of the field *Internet Value* in the table *Value Labels*.

■ The resulting active dataset has an *Internet Label* value of *No* for all cases with a value of 0 for *Internet* and *Yes* for all cases with a value of 1 for *Internet*.

■ Since the left outer join includes all records from *SurveyResponses*, there are cases in the active dataset with values of 8 or 9 for *Internet* and no value (a blank string) for *Internet Label*, since the values of 8 and 9 do not occur in the *Internet Value* field in the table *Value Labels*.

Reading IBM SPSS Statistics Data Files with SQL Statements

You can select subsets of variables when you read IBM® SPSS® Statistics data files with the GET command, and you can select subsets of cases with SELECT IF. You can also use standard SQL statements to read subsets of variables and cases using the SPSS Statistics Data File Driver.

The SPSS Statistics data file driver allows you to read SPSS Statistics (*.sav*) data files in applications that support Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC). SPSS Statistics itself supports ODBC in the Database Wizard, providing you with the ability to leverage the Structured Query Language (SQL) when reading *.sav* data files in SPSS Statistics.

There are three flavors of the SPSS Statistics data file driver, all of which are available for Windows, UNIX, and Linux:

- **Standalone driver**. The standalone driver provides ODBC support without requiring installation of additional components. After the standalone driver is installed, you can immediately set up an ODBC data source and use it to read *.sav* files.

- **Service driver**. The service driver provides both ODBC and JDBC support. The service driver handles data requests from the service client driver, which may be installed on the same computer or on one or more remote computers. Thus you can configure one service driver that may be used by many clients. If you put your data files on the same computer on which the service driver is installed, the service driver can reduce network traffic because all the queries occur on the server. Only the resulting cases are sent to the service client. If the server has a faster processor or more RAM compared to service client machines, there may also be performance improvements.

- **Service client driver**. The service client driver provides an interface between the client application that needs to read the *.sav* data file and the service driver that handles the request for the data. Unlike the standalone driver, it supports both ODBC and JDBC. The operating system of the service client driver does not need to match the operating system of the service driver. For example, you can install the service driver on a UNIX machine and the service client driver on a Windows machine.

Using the standalone and service client drivers is similar to connecting to a database with any other ODBC or JDBC driver. After configuring the driver, creating data sources, and connecting to the SPSS Statistics data file, you will see that the data file is represented as a collection of tables. In other words, the data file looks like a database source.

Installing the IBM SPSS Statistics Data File Driver

You can download and install the IBM® SPSS® Statistics data file driver from *http://www.spss.com/drivers/*. The *SPSS Statistics Data File Driver Guide*, available from the same location, contains information on installing and configuring the driver.

Using the Standalone Driver

This example uses the ODBC standalone driver to select a subset of variables and cases when reading a data file in IBM® SPSS® Statistics format into SPSS Statistics. For an example of how to use the driver to read SPSS Statistics data files into other applications, see "Example: Using the Standalone Driver with Excel" in Chapter 9.

```
*sav_odbc.sps.
GET DATA
  /TYPE=ODBC
  /CONNECT=
  "DRIVER=IBM SPSS Statistics 19 Data File Driver - Standalone;"
  "SDSN=SAVDB;"
  "HST=C:\Program Files\IBM\SPSS\StatisticsDataFileDriver\19"
      "\Standalone\cfg\oadm.ini;"
  "PRT=StatisticsSAVDriverStandalone;"
  "CP_CONNECT_STRING=C:\examples\data\demo.sav;"
  "CP_UserMissingIsNull=0"
   /SQL="SELECT age, marital, inccat, gender  FROM demo.Cases "
   "WHERE (age > 40 AND gender = 'm')".
CACHE.
EXECUTE.
APPLY DICTIONARY FROM '/examples/data/demo.sav'.
```

- **DRIVER.** Instead of specifying a DSN (data source name), the CONNECT statement specifies the driver name. You could define DSNs for each SPSS Statistics data file that you want to access with the ODBC driver (using the ODBC Data Source Administrator on Windows), but specifying the driver and all other parameters on the CONNECT statement makes it easier to reuse and modify the same basic syntax for different data files. The driver name is always SPSS Statistics <version> Data File Driver - Standalone, where <*version*> is the product version number.

- **SDSN.** This is always set to SAVDB.

- **HST.** This specifies the location of the *oadm.ini* file. It is located in the *cfg* sub-directory of the driver installation directory.

- **PRT.** This is always set to StatisticsSAVDriverStandalone.

- **CP_CONNECT_STRING.** The full path and name of the SPSS Statistics data file. This path cannot contain an equals sign (=) or semicolon (;).

- **CP_UserMissingIsNull.** This specifies the treatment of user-defined missing values. If it is set to 0, user-defined missing values are read as valid values. If it is set to 1, user-defined missing values are set to system-missing for numeric variables and blank for string variables. In this example, the user-defined missing values will be read as valid values and then the original user-missing definitions will be reapplied with APPLY DICTIONARY.

- **SQL.** The SQL subcommand uses standard SQL syntax to specify the variables (fields) to include, the name of the database table, and the case (record) selection rules.

- **SELECT** specifies the subset of variables (fields) to read. In this example, the variables *age*, *marital*, *inccat*, and *gender*.

- **FROM** specifies the database table to read. The prefix is the name of the SPSS Statistics data file. The *Cases* table contains the case data values.

- **WHERE** specifies the criteria for selecting cases (records). In this example, males over 40 years of age.
- **APPLY DICTIONARY** applies the dictionary information (variable labels, value labels, missing value definitions, and so forth) from the original SPSS Statistics data file. When you use `GET DATA /TYPE=ODBC` to read SPSS Statistics data files, the dictionary information is not included, but this is easily restored with `APPLY DICTIONARY`.

Reading Excel Files

You can read individual Excel worksheets and multiple worksheets in the same Excel workbook. The basic mechanics of reading Excel files are relatively straightforward—rows are read as cases and columns are read as variables. However, reading a typical Excel spreadsheet—where the data may not start in row 1, column 1—requires a little extra work, and reading multiple worksheets requires treating the Excel workbook as a database. In both instances, we can use the `GET DATA` command to read the data.

Reading a "Typical" Worksheet

When reading an individual worksheet, IBM® SPSS® Statistics reads a rectangular area of the worksheet, and everything in that area must be data related. The first row of the area may or may not contain variable names (depending on your specifications); the remainder of the area must contain the data to be read. A typical worksheet, however, may also contain titles and other information that may not be appropriate for data in SPSS Statistics and may even cause the data to be read incorrectly if you don't explicitly specify the range of cells to read.

Example

Figure 3-4
Typical Excel worksheet

To read this spreadsheet without the title row or total row and column:

```
*readexcel.sps.
GET DATA
   /TYPE=XLS
   /FILE='/examples/data/sales.xls'
   /SHEET=NAME 'Gross Revenue'
   /CELLRANGE=RANGE 'A2:I15'
   /READNAMES=on .
```

- The `TYPE` subcommand identifies the file type as Excel 95 or later. For earlier versions, use `GET TRANSLATE`. For Excel 2007 or later, user `GET DATA /TYPE=XLSX` (or `XLSM`).

- The `SHEET` subcommand identifies which worksheet of the workbook to read. Instead of the `NAME` keyword, you could use the `INDEX` keyword and an integer value indicating the sheet location in the workbook. Without this subcommand, the first worksheet is read.

- The `CELLRANGE` subcommand indicates that the data should be read starting at column *A*, row 2, and read through column *I*, row 15.

- The `READNAMES` subcommand indicates that the first row of the specified range contains column labels to be used as variable names.

Figure 3-5
Excel worksheet read into IBM SPSS Statistics

- The Excel column label *Store Number* is automatically converted to the variable name *StoreNumber*, since variable names cannot contain spaces. The original column label is retained as the variable label.

- The original data type from Excel is preserved whenever possible, but since data type is determined at the individual cell level in Excel and at the column (variable) level in SPSS Statistics, this isn't always possible.

- When mixed data types exist in the same column, the variable is assigned the string data type; so, the variable *Toys* in this example is assigned the string data type.

READNAMES Subcommand

The READNAMES subcommand treats the first row of the spreadsheet or specified range as either variable names (ON) or data (OFF). This subcommand will always affect the way the Excel spreadsheet is read, even when it isn't specified, since the default setting is ON.

- With READNAMES=ON (or in the absence of this subcommand), if the first row contains data instead of column headings, IBM® SPSS® Statistics will attempt to read the cells in that row as variable names instead of as data—alphanumeric values will be used to create variable names, numeric values will be ignored, and default variable names will be assigned.

- With READNAMES=OFF, if the first row does, in fact, contain column headings or other alphanumeric text, then those column headings will be read as data values, and all of the variables will be assigned the string data type.

Reading Multiple Worksheets

An Excel file (workbook) can contain multiple worksheets, and you can read multiple worksheets from the same workbook by treating the Excel file as a database. This requires an ODBC driver for Excel.

Figure 3-6
Multiple worksheets in same workbook

When reading multiple worksheets, you lose some of the flexibility available for reading individual worksheets:

- You cannot specify cell ranges.

- The first non-empty row of each worksheet should contain column labels that will be used as variable names.

- Only basic data types—string and numeric—are preserved, and string variables may be set to an arbitrarily long width.

Example

In this example, the first worksheet contains information about store location, and the second and third contain information for different departments. All three contain a column, *Store Number*, that uniquely identifies each store, so, the information in the three sheets can be merged correctly regardless of the order in which the stores are listed on each worksheet.

```
*readexcel2.sps.
GET DATA
  /TYPE=ODBC
  /CONNECT=
    'DSN=Excel Files;DBQ=c:\examples\data\sales.xls;' +
    'DriverId=790;MaxBufferSize=2048;PageTimeout=5;'
    /SQL =
```

```
'SELECT  Location$.[Store Number], State, Region, City,'
' Power, Hand, Accessories,'
' Tires, Batteries, Gizmos, Dohickeys'
' FROM [Location$], [Tools$], [Auto$]'
' WHERE [Tools$].[Store Number]=[Location$].[Store Number]'
' AND [Auto$].[Store Number]=[Location$].[Store Number]'.
```

■ If these commands look like random characters scattered on the page to you, try using the Database Wizard (File > Open Database) and, in the last step, paste the commands into a syntax window.

■ Even if you are familiar with SQL statements, you may want to use the Database Wizard the first time to generate the proper CONNECT string.

■ The SELECT statement specifies the columns to read from each worksheet, as identified by the column headings. Since all three worksheets have a column labeled *Store Number*, the specific worksheet from which to read this column is also included.

■ If the column headings can't be used as variable names, you can either let IBM® SPSS® Statistics automatically create valid variable names or use the AS keyword followed by a valid variable name. In this example, *Store Number* is not a valid variable name; so, a variable name of *StoreNumber* is automatically created, and the original column heading is used as the variable label.

■ The FROM clause identifies the worksheets to read.

■ The WHERE clause indicates that the data should be merged by matching the values of the column *Store Number* in the three worksheets.

Figure 3-7
Merged worksheets in IBM SPSS Statistics

Reading Text Data Files

A text data file is simply a text file that contains data. Text data files fall into two broad categories:

- **Simple** text data files, in which all variables are recorded in the same order for all cases, and all cases contain the same variables. This is basically how all data files appear once they are read into IBM® SPSS® Statistics.

- **Complex** text data files, including files in which the order of variables may vary between cases and hierarchical or nested data files in which some records contain variables with values that apply to one or more cases contained on subsequent records that contain a different set of variables (for example, city, state, and street address on one record and name, age, and gender of each household member on subsequent records).

Text data files can be further subdivided into two more categories:

- **Delimited.** Spaces, commas, tabs, or other characters are used to separate variables. The variables are recorded in the same order for each case but not necessarily in the same column locations. This is also referred to as **freefield format**. Some applications export text data in comma-separated values (CSV) format; this is a delimited format.

- **Fixed width.** Each variable is recorded in the same column location on the same line (record) for each case in the data file. No delimiter is required between values. In fact, in many text data files generated by computer programs, data values may appear to run together without even spaces separating them. The column location determines which variable is being read.

Complex data files are typically also fixed-width format data files.

Simple Text Data Files

In most cases, the Text Wizard (File > Read Text Data) provides all of the functionality that you need to read simple text data files. You can preview the original text data file and resulting IBM® SPSS® Statistics data file as you make your choices in the wizard, and you can paste the command syntax equivalent of your choices into a command syntax window at the last step.

Two commands are available for reading text data files: GET DATA and DATA LIST. In many cases, they provide the same functionality, and the choice of one versus the other is a matter of personal preference. In some instances, however, you may need to take advantage of features in one command that aren't available in the other.

GET DATA

Use GET DATA instead of DATA LIST if:

- The file is in CSV format.
- The text data file is very large.

DATA LIST

Use DATA LIST instead of GET DATA if:

- The text data is "inline" data contained in a command syntax file using BEGIN DATA–END DATA.

- The file has a complex structure, such as a mixed or hierarchical structure. For more information, see the topic "Reading Complex Text Data Files" on p. 41.

- You want to use the TO keyword to define a large number of sequential variable names (for example, var1 TO var1000).

- You need to specify the encoding of the text file. For more information, see the topic "Code Page and Unicode Data Sources" on p. 52.

Many examples in other chapters use DATA LIST to define sample data simply because it supports the use of inline data contained in the command syntax file rather than in an external data file, making the examples self-contained and requiring no additional files to work.

Delimited Text Data

In a simple delimited (or "freefield") text data file, the absolute position of each variable isn't important; only the relative position matters. Variables should be recorded in the same order for each case, but the actual column locations aren't relevant. More than one case can appear on the same record, and some records can span multiple records, while others do not.

Example

One of the advantages of delimited text data files is that they don't require a great deal of structure. The sample data file, *simple_delimited.txt*, looks like this:

```
1 m 28 1 2 2 1 2 2 f 29 2 1 2 1 2
003 f 45 3 2 1 4 5 128 m 17 1 1
1 9 4
```

The DATA LIST command to read the data file is:

```
*simple_delimited.sps.
DATA LIST FREE
   FILE = '/examples/data/simple_delimited.txt'
   /id (F3) sex (A1) age (F2) opinion1 TO opinion5 (5F).
EXECUTE.
```

- FREE indicates that the text data file is a delimited file, in which only the order of variables matters. By default, commas and spaces are read as delimiters between data values. In this example, all of the data values are separated by spaces.

- Eight variables are defined, so after reading eight values, the next value is read as the first variable for the next case, even if it's on the same line. If the end of a record is reached before eight values have been read for the current case, the first value on the next line is read as the next value for the current case. In this example, four cases are contained on three records.

- If all of the variables were simple numeric variables, you wouldn't need to specify the format for any of them, but if there are any variables for which you need to specify the format, any preceding variables also need format specifications. Since you need to specify a string format for *sex*, you also need to specify a format for *id*.

- In this example, you don't need to specify formats for any of the numeric variables that appear after the string variable, but the default numeric format is F8.2, which means that values are displayed with two decimals even if the actual values are integers. (F2) specifies an integer with a maximum of two digits, and (5F) specifies five integers, each containing a single digit.

The "defined format for all preceding variables" rule can be quite cumbersome, particularly if you have a large number of simple numeric variables interspersed with a few string variables or other variables that require format specifications. You can use a shortcut to get around this rule:

```
DATA LIST FREE
  FILE = '/examples/data/simple_delimited.txt'
  /id * sex (A1) age opinion1 TO opinion5.
```

The asterisk indicates that all preceding variables should be read in the default numeric format (F8.2). In this example, it doesn't save much over simply defining a format for the first variable, but if *sex* were the last variable instead of the second, it could be useful.

Example

One of the drawbacks of DATA LIST FREE is that if a single value for a single case is accidently missed in data entry, all subsequent cases will be read incorrectly, since values are read sequentially from the beginning of the file to the end regardless of what line each value is recorded on. For delimited files in which each case is recorded on a separate line, you can use DATA LIST LIST, which will limit problems caused by this type of data entry error to the current case.

The data file, *delimited_list.txt*, contains one case that has only seven values recorded, whereas all of the others have eight:

```
001 m 28 1 2 2 1 2
002 f 29 2 1 2 1 2
003 f 45 3 2 4 5
128 m 17 1 1 1 9 4
```

The DATA LIST command to read the file is:

```
*delimited_list.sps.
DATA LIST LIST
  FILE='/examples/data/delimited_list.txt'
  /id(F3) sex (A1) age opinion1 TO opinion5 (6F1).
EXECUTE.
```

Figure 3-8
Text data file read with DATA LIST LIST

- Eight variables are defined, so eight values are expected on each line.
- The third case, however, has only seven values recorded. The first seven values are read as the values for the first seven defined variables. The eighth variable is assigned the system-missing value.

You don't know which variable for the third case is actually missing. In this example, it could be any variable after the second variable (since that's the only string variable, and an appropriate string value was read), making all of the remaining values for that case suspect; so, a warning message is issued whenever a case doesn't contain enough data values:

```
>Warning # 1116
>Under LIST input, insufficient data were contained on one record to
>fulfill the variable list.
>Remaining numeric variables have been set to the system-missing
>value and string variables have been set to blanks.
>Command line: 6  Current case: 3  Current splitfile group: 1
```

CSV Delimited Text Files

A CSV file uses commas to separate data values and encloses values that include commas in quotation marks. Many applications export text data in this format. To read CSV files correctly, you need to use the GET DATA command.

Example

The file *CSV_file.csv* was exported from Microsoft Excel:

```
ID,Name,Gender,Date Hired,Department
1,"Foster, Chantal",f,10/29/1998,1
2,"Healy, Jonathan",m,3/1/1992,3
3,"Walter, Wendy",f,1/23/1995,2
4,"Oliver, Kendall",f,10/28/2003,2
```

This data file contains variable descriptions on the first line and a combination of string and numeric data values for each case on subsequent lines, including string values that contain commas. The GET DATA command syntax to read this file is:

```
*delimited_csv.sps.
GET DATA  /TYPE = TXT
```

```
        /FILE = '/examples/data/CSV_file.csv'
        /DELIMITERS = ","
        /QUALIFIER = '"'
        /ARRANGEMENT = DELIMITED
        /FIRSTCASE = 2
        /VARIABLES = ID F3 Name A15 Gender A1
         Date_Hired ADATE10 Department F1.
```

- ■ `DELIMITERS = ","` specifies the comma as the delimiter between values.

- ■ `QUALIFIER = '"'` specifies that values that contain commas are enclosed in double quotes so that the embedded commas won't be interpreted as delimiters.

- ■ `FIRSTCASE = 2` skips the top line that contains the variable descriptions; otherwise, this line would be read as the first case.

- ■ `ADATE10` specifies that the variable *Date_Hired* is a date variable of the general format mm/dd/yyyy. For more information, see the topic "Reading Different Types of Text Data" on p. 40.

Note: The command syntax in this example was adapted from the command syntax generated by the Text Wizard (File > Read Text Data), which automatically generated valid variable names from the information on the first line of the data file.

Fixed-Width Text Data

In a fixed-width data file, variables start and end in the same column locations for each case. No delimiters are required between values, and there is often no space between the end of one value and the start of the next. For fixed-width data files, the command that reads the data file (GET DATA or DATA LIST) contains information on the column location and/or width of each variable.

Example

In the simplest type of fixed-width text data file, each case is contained on a single line (record) in the file. In this example, the text data file *simple_fixed.txt* looks like this:

```
001 m 28 12212
002 f 29 21212
003 f 45 32145
128 m 17 11194
```

Using DATA LIST, the command syntax to read the file is:

```
*simple_fixed.sps.
DATA LIST FIXED
  FILE='/examples/data/simple_fixed.txt'
  /id 1-3 sex 5 (A) age 7-8 opinion1 TO opinion5 10-14.
EXECUTE.
```

- ■ The keyword FIXED is included in this example, but since it is the default format, it can be omitted.

- ■ The forward slash before the variable *id* separates the variable definitions from the rest of the command specifications (unlike other commands where subcommands are separated by forward slashes). The forward slash actually denotes the start of each record that will be read, but in this case there is only one record per case.

- The variable *id* is located in columns 1 through 3. Since no format is specified, the standard numeric format is assumed.

- The variable *sex* is found in column 5. The format (A) indicates that this is a string variable, with values that contain something other than numbers.

- The numeric variable *age* is in columns 7 and 8.

- `opinion1 TO opinion5 10-14` defines five numeric variables, with each variable occupying a single column: *opinion1* in column 10, *opinion2* in column 11, and so on.

You could define the same data file using variable width instead of column locations:

```
*simple_fixed_alt.sps.
DATA LIST FIXED
  FILE='/examples/data/simple_fixed.txt'
  /id (F3, 1X) sex (A1, 1X) age (F2, 1X)
   opinion1 TO opinion5 (5F1).
EXECUTE.
```

- `id (F3, 1X)` indicates that the variable *id* is in the first three column positions, and the next column position (column 4) should be skipped.

- Each variable is assumed to start in the next sequential column position; so, *sex* is read from column 5.

Figure 3-9
Fixed-width text data file displayed in Data Editor

Example

Reading the same file with GET DATA, the command syntax would be:

```
*simple_fixed_getdata.sps.
GET DATA  /TYPE = TXT
 /FILE = '/examples/data/simple_fixed.txt'
 /ARRANGEMENT = FIXED
 /VARIABLES =/1  id 0-2 F3 sex 4-4 A1 age 6-7 F2
  opinion1 9-9 F opinion2 10-10 F opinion3 11-11 F
  opinion4 12-12 F opinion5 13-13 F.
```

- The first column is column 0 (in contrast to DATA LIST, in which the first column is column 1).

- There is no default data type. You must explicitly specify the data type for all variables.

- You must specify both a start and an end column position for each variable, even if the variable occupies only a single column (for example, sex 4-4).

- All variables must be explicitly specified; you cannot use the keyword TO to define a range of variables.

Reading Selected Portions of a Fixed-Width File

With fixed-format text data files, you can read all or part of each record and/or skip entire records.

Example

In this example, each case takes two lines (records), and the first line of the file should be skipped because it doesn't contain data. The data file, *skip_first_fixed.txt*, looks like this:

```
Employee age, department, and salary information
John Smith
26 2 40000
Joan Allen
32 3 48000
Bill Murray
45 3 50000
```

The DATA LIST command syntax to read the file is:

```
*skip_first_fixed.sps.
DATA LIST FIXED
 FILE = '/examples/data/skip_first_fixed.txt'
  RECORDS=2
  SKIP=1
  /name 1-20 (A)
  /age 1-2 dept 4 salary 6-10.
EXECUTE.
```

- The RECORDS subcommand indicates that there are two lines per case.

- The SKIP subcommand indicates that the first line of the file should not be included.

- The first forward slash indicates the start of the list of variables contained on the first record for each case. The only variable on the first record is the string variable *name*.

- The second forward slash indicates the start of the variables contained on the second record for each case.

Figure 3-10
Fixed-width, multiple-record text data file displayed in Data Editor
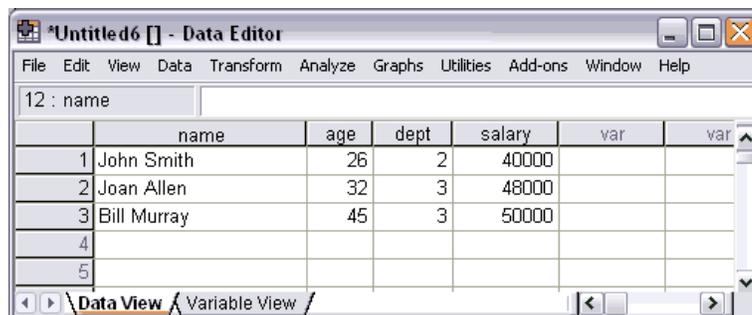
Example

With fixed-width text data files, you can easily read selected portions of the data. For example, using the *skip_first_fixed.txt* data file from the above example, you could read just the age and salary information.

```
*selected_vars_fixed.sps.
DATA LIST FIXED
  FILE = '/examples/data/skip_first_fixed.txt'
  RECORDS=2
  SKIP=1
  /2 age 1-2 salary 6-10.
EXECUTE.
```

- As in the previous example, the command specifies that there are two records per case and that the first line in the file should not be read.
- `/2` indicates that variables should be read from the second record for each case. Since this is the only list of variables defined, the information on the first record for each case is ignored, and the employee's name is not included in the data to be read.
- The variables *age* and *salary* are read exactly as before, but no information is read from columns 3–5 between those two variables because the command does not define a variable in that space—so the department information is not included in the data to be read.

DATA LIST FIXED and Implied Decimals

If you specify a number of decimals for a numeric format with `DATA LIST FIXED` and some data values for that variable do not contain decimal indicators, those values are assumed to contain **implied decimals**.

Example

```
*implied_decimals.sps.
DATA LIST FIXED /var1 (F5.2).
BEGIN DATA
123
123.0
1234
123.4
end data.
```

- The values of 123 and 1234 will be read as containing two implied decimals positions, resulting in values of 1.23 and 12.34.
- The values of 123.0 and 123.4, however, contain **explicit** decimal indicators, resulting in values of 123.0 and 123.4.

`DATA LIST FREE` (and `LIST`) and `GET DATA /TYPE=TEXT` do *not* read implied decimals; so a value of 123 with a format of F5.2 will be read as 123.

Text Data Files with Very Wide Records

Some machine-generated text data files with a large number of variables may have a single, very wide record for each case. If the record width exceeds 8,192 columns/characters, you need to specify the record length with the FILE HANDLE command before reading the data file.

```
*wide_file.sps.
*Read text data file with record length of 10,000.
*This command will stop at column 8,192.
DATA LIST FIXED
  FILE='/examples/data/wide_file.txt'
  /var1 TO var1000 (1000F10).
EXECUTE.
*Define record length first.
FILE HANDLE  wide_file NAME = '/examples/data/wide_file.txt'
  /MODE = CHARACTER /LRECL = 10000.
DATA LIST FIXED
  FILE = wide_file
  /var1 TO var1000 (1000F10).
EXECUTE.
```

- Each record in the data file contains 1,000 10-digit values, for a total record length of 10,000 characters.

- The first DATA LIST command will read only the first 819 values (8,190 characters), and the remaining variables will be set to the system-missing value. A warning message is issued for each variable that is set to system-missing, which in this example means 181 warning messages.

- FILE HANDLE assigns a "handle" of *wide_file* to the data file *wide_file.txt*.

- The LRECL subcommand specifies that each record is 10,000 characters wide.

- The FILE subcommand on the second DATA LIST command refers to the file handle *wide_file* instead of the actual filename, and all 1,000 variables are read correctly.

Reading Different Types of Text Data

You can read text data recorded in a wide variety of formats. Some of the more common formats are listed in the following table:

Type	Example	Format specification
Numeric	123	F3
	123.45	F6.2
Period as decimal indicator, comma as thousands separator	12,345	COMMA6
	1,234.5	COMMA7.1
Comma as decimal indicator, period as thousands separator	123,4	DOT6
	1.234,5	DOT7.1
Dollar	$12,345	DOLLAR7
	$12,234.50	DOLLAR9.2
String (alphanumeric)	Female	A6
International date	28-OCT-1986	DATE11
American date	10/28/1986	ADATE10
Date and time	28 October, 1986 23:56	DATETIME22

For more information on date and time formats, see "Date and Time" in the "Universals" section of the *Command Syntax Reference*. For a complete list of supported data formats, see "Variables" in the "Universals" section of the *Command Syntax Reference*.

Example

```
*delimited_formats.sps.
DATA LIST LIST (" ")
  /numericVar (F4) dotVar(DOT7.1) stringVar(a4) dateVar(DATE11).
BEGIN DATA
1 2 abc 28/10/03
111 2.222,2 abcd 28-OCT-2003
111.11 222.222,222 abcdefg 28-October-2003
END DATA.
```

Figure 3-11
Different data types displayed in Data Editor

- All of the numeric and date values are read correctly even if the actual values exceed the maximum width (number of digits and characters) defined for the variables.

- Although the third case appears to have a truncated value for *numericVar*, the entire value of 111.11 is stored internally. Since the defined format is also used as the display format, and (F4) defines a format with no decimals, 111 is displayed instead of the full value. Values are not actually truncated for display; they are rounded. A value of 111.99 would display as 112.

- The *dateVar* value of 28-October-2003 is displayed as 28-OCT-2003 to fit the defined width of 11 digits/characters.

- For string variables, the defined width is more critical than with numeric variables. Any string value that exceeds the defined width is truncated, so only the first four characters for *stringVar* in the third case are read. Warning messages are displayed in the log for any strings that exceed the defined width.

Reading Complex Text Data Files

"Complex" text data files come in a variety of flavors, including:

- Mixed files in which the order of variables isn't necessarily the same for all records and/or some record types should be skipped entirely.

- Grouped files in which there are multiple records for each case that need to be grouped together.

- Nested files in which record types are related to each other hierarchically.

Mixed Files

A mixed file is one in which the order of variables may differ for some records and/or some records may contain entirely different variables or information that shouldn't be read.

Example

In this example, there are two record types that should be read: one in which *state* appears before *city* and one in which *city* appears before *state*. There is also an additional record type that shouldn't be read.

```
*mixed_file.sps.
FILE TYPE MIXED RECORD = 1-2.
- RECORD TYPE 1.
-  DATA LIST FIXED
     /state 4-5 (A) city 7-17 (A) population 19-26 (F).
- RECORD TYPE 2.
-  DATA LIST FIXED
     /city 4-14 (A) state 16-17 (A) population 19-26 (F).
END FILE TYPE.
BEGIN DATA
01 TX Dallas      3280310
01 IL Chicago     8008507
02 Ancorage    AK 257808
99 What am I doing here?
02 Casper      WY 63157
01 WI Madison     428563
END DATA.
```

- The commands that define how to read the data are all contained within the FILE TYPE–END FILE TYPE structure.

- MIXED identifies the type of data file.

- RECORD = 1-2 indicates that the record type identifier appears in the first two columns of each record.

- Each DATA LIST command reads only records with the identifier value specified on the preceding RECORD TYPE command. So if the value in the first two columns of the record is 1 (or 01), *state* comes before *city*, and if the value is 2, *city* comes before *state*.

- The record with the value 99 in the first two columns is not read, since there are no corresponding RECORD TYPE and DATA LIST commands.

You can also include a variable that contains the record identifier value by including a variable name on the RECORD subcommand of the FILE TYPE command, as in:

```
FILE TYPE MIXED /RECORD = recID 1-2.
```

You can also specify the format for the identifier value, using the same type of format specifications as the DATA LIST command. For example, if the value is a string instead of a simple numeric value:

```
FILE TYPE MIXED /RECORD = recID 1-2 (A).
```

Grouped Files

In a grouped file, there are multiple records for each case that should be grouped together based on a unique case identifier. Each case usually has one record of each type. All records for a single case must be together in the file.

Example

In this example, there are three records for each case. Each record contains a value that identifies the case, a value that identifies the record type, and a grade or score for a different course.

```
* grouped_file.sps.
* A case is made up of all record types.
FILE TYPE GROUPED RECORD=6 CASE=student 1-4.
RECORD TYPE 1.
- DATA LIST /english 8-9 (A).
RECORD TYPE 2.
- DATA LIST /reading 8-10.
RECORD TYPE 3.
- DATA LIST /math 8-10.
END FILE TYPE.

BEGIN DATA
0001 1 B+
0001 2 74
0001 3 83
0002 1 A
0002 3 71
0002 2 100
0003 1 B-
0003 2 88
0003 3 81
0004 1 C
0004 2 94
0004 3 91
END DATA.
```

- The commands that define how to read the data are all contained within the FILE TYPE–END FILE TYPE structure.

- GROUPED identifies the type of data file.

- RECORD=6 indicates that the record type identifier appears in column 6 of each record.

- CASE=student 1-4 indicates that the unique case identifier appears in the first four columns and assigns that value to the variable *student* in the active dataset.

- The three RECORD TYPE and subsequent DATA LIST commands determine how each record is read, based on the value in column 6 of each record.

Figure 3-12
Grouped data displayed in Data Editor

Example

In order to read a grouped data file correctly, all records for the same case must be contiguous in the source text data file. If they are not, you need to sort the data file before reading it as a grouped data file. You can do this by reading the file as a simple text data file, sorting it and saving it, and then reading it again as a grouped file.

```
*grouped_file2.sps.
* Data file is sorted by record type instead of by
  identification number.
DATA LIST FIXED
  /alldata 1-80 (A) caseid 1-4.
BEGIN DATA
0001 1 B+
0002 1 A
0003 1 B-
0004 1 C
0001 2 74
0002 2 100
0003 2 88
0004 2 94
0001 3 83
0002 3 71
0003 3 81
0004 3 91
END DATA.
SORT CASES BY caseid.
WRITE OUTFILE='/temp/tempdata.txt'
 /alldata.
EXECUTE.
* read the sorted file.
FILE TYPE GROUPED FILE='/temp/tempdata.txt'
 RECORD=6 CASE=student 1-4.
- RECORD TYPE 1.
-   DATA LIST /english 8-9 (A).
- RECORD TYPE 2.
-   DATA LIST /reading 8-10.
- RECORD TYPE 3.
-   DATA LIST /math 8-10.
END FILE TYPE.
EXECUTE.
```

■ The first DATA LIST command reads all of the data on each record as a single string variable.

- In addition to being part of the string variable spanning the entire record, the first four columns are read as the variable *caseid*.

- The data file is then sorted by *caseid*, and the string variable *alldata*, containing all of the data on each record, is written to the text file *tempdata.txt*.

- The sorted file, *tempdata.txt*, is then read as a grouped data file, just like the inline data in the previous example.

Prior to release 13.0, the maximum width of a string variable was 255 bytes. So in earlier releases, for a file with records wider than 255 bytes, you would need to modify the job slightly to read and write multiple string variables. For example, if the record width is 1,200:

```
DATA LIST FIXED
  /string1 to string6 1-1200 (A) caseid 1-4.
```

This would read the file as six 200-byte string variables.

IBM® SPSS® Statistics can now handle much longer strings in a single variable: 32,767 bytes. So this workaround is unnecessary for release 13.0 or later. (If the record length exceeds 8,192 bytes, you need to use the FILE HANDLE command to specify the record length. See the *Command Syntax Reference* for more information.)

Nested (Hierarchical) Files

In a nested file, the record types are related to each other hierarchically. The record types are grouped together by a case identification number that identifies the highest level—the first record type—of the hierarchy. Usually, the last record type specified—the lowest level of the hierarchy—defines a case. For example, in a file containing information on a company's sales representatives, the records could be grouped by sales region. Information from higher record types can be spread to each case. For example, the sales region information can be spread to the records for each sales representative in the region.

Example

In this example, sales data for each sales representative are nested within sales regions (cities), and those regions are nested within years.

```
*nested_file1.sps.
FILE TYPE NESTED RECORD=1(A).
- RECORD TYPE 'Y'.
- DATA LIST / Year 3-6.
- RECORD TYPE 'R'.
- DATA LIST / Region 3-13 (A).
- RECORD TYPE 'P'.
- DATA LIST / SalesRep 3-13 (A) Sales 20-23.
END FILE TYPE.
BEGIN DATA
Y 2002
R Chicago
P Jones             900
P Gregory           400
R Baton Rouge
P Rodriguez         300
P Smith             333
P Grau              100
```

```
END DATA.
```

Figure 3-13
Nested data displayed in Data Editor

- The commands that define how to read the data are all contained within the FILE TYPE–END FILE TYPE structure.

- NESTED identifies the type of data file.

- The value that identifies each record type is a string value in column 1 of each record.

- The order of the RECORD TYPE and associated DATA LIST commands defines the nesting hierarchy, with the highest level of the hierarchy specified first. So, 'Y' (year) is the highest level, followed by 'R' (region), and finally 'P' (person).

- Eight records are read, but one of those contains year information and two identify regions; so, the active dataset contains five cases, all with a value of 2002 for *Year*, two in the Chicago *Region* and three in Baton Rouge.

Using INPUT PROGRAM to Read Nested Files

The previous example imposes some strict requirements on the structure of the data. For example, the value that identifies the record type must be in the same location on all records, and it must also be the same type of data value (in this example, a one-character string).

Instead of using a FILE TYPE structure, we can read the same data with an INPUT PROGRAM, which can provide more control and flexibility.

Example

This first input program reads the same data file as the FILE TYPE NESTED example and obtains the same results in a different manner.

```
* nested_input1.sps.
INPUT PROGRAM.
- DATA LIST FIXED END=#eof /#type 1 (A).
- DO IF #eof.
-   END FILE.
- END IF.
- DO IF #type='Y'.
-   REREAD.
-   DATA LIST /Year 3-6.
-   LEAVE Year.
- ELSE IF #type='R'.
-   REREAD.
```

```
-   DATA LIST / Region 3-13 (A).
-   LEAVE Region.
- ELSE IF #type='P'.
-   REREAD.
-   DATA LIST / SalesRep 3-13 (A) Sales 20-23.
-   END CASE.
- END IF.
END INPUT PROGRAM.
BEGIN DATA
Y 2002
R Chicago
P Jones            900
P Gregory          400
R Baton Rouge
P Rodriguez        300
P Smith            333
P Grau             100
END DATA.
```

- The commands that define how to read the data are all contained within the INPUT PROGRAM structure.

- The first DATA LIST command reads the temporary variable *#type* from the first column of each record.

- END=#eof creates a temporary variable named *#eof* that has a value of 0 until the end of the data file is reached, at which point the value is set to 1.

- DO IF #eof evaluates as *true* when the value of *#eof* is set to 1 at the end of the file, and an END FILE command is issued, which tells the INPUT PROGRAM to stop reading data. In this example, this isn't really necessary, since we're reading the entire file; however, it will be used later when we want to define an end point prior to the end of the data file.

- The second DO IF-ELSE IF-END IF structure determines what to do for each value of *type*.

- REREAD reads the same record again, this time reading either *Year*, *Region*, or *SalesRep* and *Sales*, depending on the value of *#type*.

- LEAVE retains the value(s) of the specified variable(s) when reading the next record. So the value of *Year* from the first record is retained when reading *Region* from the next record, and both of those values are retained when reading *SalesRep* and *Sales* from the subsequent records in the hierarchy. Thus, the appropriate values of *Year* and *Region* are spread to all of the cases at the lowest level of the hierarchy.

- END CASE marks the end of each case. So, after reading a record with a *#type* value of 'P', the process starts again to create the next case.

Example

In this example, the data file reflects the nested structure by indenting each nested level; so the values that identify record type do not appear in the same place on each record. Furthermore, at the lowest level of the hierarchy, the record type identifier is the last value instead of the first. Here, an INPUT PROGRAM provides the ability to read a file that cannot be read correctly by FILE TYPE NESTED.

```
*nested_input2.sps.
INPUT PROGRAM.
- DATA LIST FIXED END=#eof
    /#yr 1 (A) #reg 3(A) #person 25 (A).
```

```
-  DO IF #eof.
-   END FILE.
-  END IF.
-  DO IF #yr='Y'.
-   REREAD.
-   DATA LIST /Year 3-6.
-   LEAVE Year.
-  ELSE IF #reg='R'.
-   REREAD.
-   DATA LIST / Region 5-15 (A).
-   LEAVE Region.
-  ELSE IF #person='P'.
-   REREAD.
-   DATA LIST / SalesRep 7-17 (A) Sales 20-23.
-   END CASE.
-  END IF.
END INPUT PROGRAM.
BEGIN DATA
Y 2002
  R Chicago
      Jones          900  P
      Gregory        400  P
  R Baton Rouge
      Rodriguez      300  P
      Smith          333  P
      Grau           100  P
END DATA.
```

■ This time, the first `DATA LIST` command reads three temporary variables at different locations, one for each record type.

■ The `DO IF–ELSE IF–END IF` structure then determines how to read each record based on the values of *#yr*, *#reg*, or *#person*.

■ The remainder of the job is essentially the same as the previous example.

Example

Using the input program, we can also select a random sample of cases from each region and/or stop reading cases at a specified maximum.

```
*nested_input3.sps.
INPUT PROGRAM.
NUMERIC #count (F8).
- DATA LIST FIXED END=#eof
   /#yr 1 (A) #reg 3(A) #person 25 (A).
- DO IF #eof OR #count = 1000.
-   END FILE.
- END IF.
- DO IF #yr='Y'.
-   REREAD.
-   DATA LIST /Year 3-6.
-   LEAVE Year.
- ELSE IF #reg='R'.
-   REREAD.
-   DATA LIST / Region 5-15 (A).
-   LEAVE Region.
- ELSE IF #person='P' AND UNIFORM(1000) < 500.
-   REREAD.
-   DATA LIST / SalesRep 7-17 (A) Sales 20-23.
-   END CASE.
-   COMPUTE #count=#count+1.
- END IF.
END INPUT PROGRAM.
```

```
BEGIN DATA
Y 2002
  R Chicago
      Jones         900  P
      Gregory       400  P
  R Baton Rouge
      Rodriguez     300  P
      Smith         333  P
      Grau          100  P
END DATA.
```

- `NUMERIC #count (F8)` uses a scratch (temporary) variable as a case-counter variable. Scratch variables are initialized to 0 and retain their values for subsequent cases.

- `ELSE IF #person='P' AND UNIFORM(1000) < 500` will read a random sample of approximately 50% from each region, since `UNIFORM(1000)` will generate a value less than 500 approximately 50% of the time.

- `COMPUTE #count=#count+1` increments the case counter by 1 for each case that is included.

- `DO IF #eof OR #count = 1000` will issue an `END FILE` command if the case counter reaches 1,000, limiting the total number of cases in the active dataset to no more than 1,000.

Since the source file must be sorted by year and region, limiting the total number of cases to 1,000 (or any value) may omit some years or regions within the last year entirely.

Repeating Data

In a repeating data file structure, multiple cases are constructed from a single record. Information common to each case on the record may be entered once and then spread to all of the cases constructed from the record. In this respect, a file with a repeating data structure is like a hierarchical file, with two levels of information recorded on a single record rather than on separate record types.

Example

In this example, we read essentially the same information as in the examples of nested file structures, except now all of the information for each region is stored on a single record.

```
*repeating_data.sps.
INPUT PROGRAM.
DATA LIST FIXED
  /Year 1-4 Region 6-16 (A) #numrep 19.
REPEATING DATA STARTS=22 /OCCURS=#numrep
  /DATA=SalesRep 1-10 (A) Sales 12-14.
END INPUT PROGRAM.
BEGIN DATA
2002 Chicago     2   Jones       900Gregory     400
2002 Baton Rouge 3   Rodriguez   300Smith       333Grau        100
END DATA.
```

- The commands that define how to read the data are all contained within the `INPUT PROGRAM` structure.

- The `DATA LIST` command defines two variables, *Year* and *Region*, that will be spread across all of the cases read from each record. It also defines a temporary variable, *#numrep*.

- On the REPEATING DATA command, STARTS=22 indicates that the case starts in column 22.

- OCCURS=#numrep uses the value of the temporary variable, *#numrep* (defined on the previous DATA LIST command), to determine how many cases to read from each record. So, two cases will be read from the first record, and three will be read from the second.

- The DATA subcommand defines two variables for each case. The column locations for those variables are relative locations. For the first case, column 22 (specified on the STARTS subcommand) is read as column 1. For the next case, column 1 is the first column after the end of the defined column span for the last variable in the previous case, which would be column 36 (22+14=36).

The end result is an active dataset that looks remarkably similar to the data file created from the hierarchical source data file.

Figure 3-14
Repeating data displayed in Data Editor

Reading SAS Data Files

IBM® SPSS® Statistics can read the following types of SAS files:

- SAS long filename, versions 7 through 9

- SAS short filenames, versions 7 through 9

- SAS version 6 for Windows

- SAS version 6 for UNIX

- SAS Transport

The basic structure of a SAS data file is very similar to a data file in SPSS Statistics format—rows are cases (observations), and columns are variables—and reading SAS data files requires only a single, simple command: GET SAS.

Example

In its simplest form, the GET SAS command has a single subcommand that specifies the SAS filename.

```
*get_sas.sps.
GET SAS DATA='/examples/data/demo.sd7'.
```

■ SAS variable names that do not conform to SPSS Statistics variable-naming rules are converted to valid variable names.

■ SAS variable labels specified on the `LABEL` statement in the `DATA` step are used as variable labels in SPSS Statistics.

Example

SAS value formats are similar to SPSS Statistics value labels, but SAS value formats are saved in a separate file; so if you want to use value formats as value labels, you need to use the `FORMATS` subcommand to specify the formats file.

```
GET SAS DATA='datafilename.sd7'
  /FORMATS='formatsfilename.sd7'.
```

■ Labels assigned to single values are retained.

■ Labels assigned to a range of values are ignored.

■ Labels assigned to the SAS keywords `LOW`, `HIGH`, and `OTHER` are ignored.

The file specified on the `FORMATS` subcommand must be a SAS-format catalog file created with the `proc format` command. For example:

```
libname mylib 'c:\mydir\' ;

proc format library = mylib ;
value YesNo
  0='No'
  1='Yes' ;
value HighLow
   1 = 'Low'
   2 = 'Medium'
   3 = 'High' ;

options  fmtsearch=(mylib);

proc datasets library = mylib ;
modify mydata;
    format    var1 var2 var3 YesNo.;
    format    var4 var5 var6 HighLow.;

quit;
```

■ `libname` defines a "library," which is a directory path.

■ `proc format` defines two formats: *YesNo* and *HighLow*. Each format defines a set of value labels associated with data values.

■ `proc datasets` identifies the data file—*mydata*—and the variables to which each of the defined formats should be applied. So the *YesNo* format is applied to variables *var1*, *var2*, and *var3*, and the *HighLow* format is applied to the variables *var4*, *var5*, and *var6*.

■ This creates the SAS catalog file *c:\mydir\formats.sas7bcat*.

Reading Stata Data Files

`GET STATA` reads Stata-format data files created by Stata versions 4 through 8. The only specification is the `FILE` keyword, which specifies the Stata data file to be read.

- **Variable names.** Stata variable names are converted to IBM® SPSS® Statistics variable names in case-sensitive form. Stata variable names that are identical except for case are converted to valid variable names by appending an underscore and a sequential letter (*_A*, *_B*, *_C*, ..., *_Z*, *_AA*, *_AB*, ..., and so forth).

- **Variable labels.** Stata variable labels are converted to SPSS Statistics variable labels.

- **Value labels.** Stata value labels are converted to SPSS Statistics value labels, except for Stata value labels assigned to "extended" missing values.

- **Missing values.** Stata "extended" missing values are converted to system-missing values.

- **Date conversion.** Stata date format values are converted to SPSS Statistics DATE format (d-m-y) values. Stata "time-series" date format values (weeks, months, quarters, and so on) are converted to simple numeric (F) format, preserving the original, internal integer value, which is the number of weeks, months, quarters, and so on, since the start of 1960.

Example

```
GET STATA FILE='/examples/data/statafile.dta'.
```

Code Page and Unicode Data Sources

Starting with release 16.0, you can read and write Unicode data files.

SET UNICODE NO|YES controls the default behavior for determining the encoding for reading and writing data files and syntax files.

NO. *Use the current locale setting to determine the encoding for reading and writing data and command syntax files.* This is referred to as **code page mode**. This is the default. The alias is OFF.

YES. *Use Unicode encoding (UTF-8) for reading and writing data and command syntax files.* This is referred to as **Unicode mode**. The alias is ON.

- You can change the UNICODE setting only when there are no open data sources.
- The UNICODE setting persists across sessions and remains in effect until it is explicitly changed.

There are a number of important implications regarding Unicode mode and Unicode files:

- Data and syntax files saved in Unicode encoding should not be used in releases prior to 16.0. For syntax files, you can specify local encoding when you save the file. For data files, you should open the data file in code page mode and then resave it if you want to read the file with earlier versions.

- When code page data files are read in Unicode mode, the defined width of all string variables is tripled.

- The GET command determines the file encoding for IBM® SPSS® Statistics data files from the file itself, regardless of the current mode setting (and defined string variable widths in code page files are tripled in Unicode mode).

- For text data files read with DATA LIST and related commands (for example, REPEATING DATA and FILE TYPE) or written with PRINT or WRITE, you can override the default encoding with the ENCODING subcommand.

■ GET DATA uses the default encoding for reading text data files (TYPE=TXT), which is UTF-8 in Unicode mode or the code page determined by the current locale in code page mode.

■ OMS uses default encoding for writing text files (FORMAT=TEXT and FORMAT=TABTEXT) and for writing SPSS Statistics data files (FORMAT=SAV).

■ GET TRANSLATE reads data in the current locale code page, regardless of mode.

■ By default, GET STATA, GET SAS, and SAVE TRANSLATE read and write data in the current locale code page, regardless of mode. You can use the ENCODING subcommand to specify a different encoding.

■ By default, SAVE TRANSLATE saves SAS 9 files in UTF-8 format in Unicode mode and in the current locale code page in code page mode. You can use the ENCODING subcommand to specify a different encoding.

■ For syntax files run via INCLUDE or INSERT, you can override the default encoding with the ENCODING subcommand.

■ For syntax files, the encoding is set to Unicode *after* execution of the block of commands that includes SET UNICODE=YES. You must run SET UNICODE=YES separately from subsequent commands that contain Unicode characters not recognized by the local encoding in effect prior to switching to Unicode.

Example: Reading Code Page Text Data in Unicode Mode

```
*read_codepage.sps.
CD '/examples/data'.
DATASET CLOSE ALL.
NEW FILE.
SET UNICODE YES.
DATA LIST LIST FILE='codepage.txt'
   /NumVar (F3) StringVar (A8).
EXECUTE.
DATA LIST LIST FILE='codepage.txt' ENCODING='Locale'
   /NumVar (F3) StringVar (A8).
COMPUTE ByteLength=LENGTH(RTRIM(StringVar)).
COMPUTE CharLength=CHAR.LENGTH(StringVar).
SUMMARIZE
  /TABLES=StringVar ByteLength CharLength
  /FORMAT=VALIDLIST /CELLS=COUNT
  /TITLE='Unicode Byte and Character Counts'.
DISPLAY DICTIONARY VARIABLES=StringVar.
DATASET CLOSE ALL.
NEW FILE.
SET UNICODE NO.
```

■ SET UNICODE YES switches from the default code page mode to Unicode mode. Since you can change modes only when there are no open data sources, this is preceded by DATASET CLOSE ALL to close all named datasets and NEW FILE to replace the active dataset with a new, empty dataset.

■ The text data file *codepage.txt* is a code page file, not a Unicode file; so any string values that contain anything other than 7-bit ASCII characters will be read incorrectly when attempting to read the file as if it were Unicode. In this example, the string value *résumé* contains two accented characters that are not 7-bit ASCII.

■ The first `DATA LIST` command attempts to read the text data file in the default encoding. In Unicode mode, the default encoding is Unicode (UTF-8), and the string value *résumé* cannot be read correctly, which generates a warning:

```
>Warning # 1158
>An invalid character was encountered in a field read under an A format.  In
>double-byte data such as Japanese, Chinese, or Korean text, this could be
>caused by a single character being split between two fields.  The character
>will be treated as an exclamation point.
```

■ `ENCODING='Locale'` on the second `DATA LIST` command identifies the encoding for the text data file as the code page for the current locale, and the string value *résumé* is read correctly. (If your current locale is not English, use `ENCODING='1252'`.)

■ `LENGTH(RTRIM(StringVar))` returns the number of bytes in each value of *StringVar*. Note that *résumé* is eight bytes in Unicode mode because each accented *e* takes two bytes.

■ `CHAR.LENGTH(StringVar)` returns the number characters in each value of *StringVar*. While an accented *e* is two bytes in Unicode mode, it is only one character; so both *résumé* and *resume* contain six characters.

■ The output from the `DISPLAY DICTIONARY` command shows that the defined width of *StringVar* has been tripled from the input width of A8 to A24. To minimize the expansion of string widths when reading code page data in Unicode mode, you can use the `ALTER TYPE` command to automatically set the width of each string variable to the maximum observed string value for that variable. For more information, see the topic "Changing Data Types and String Widths" in Chapter 6 on p. 95.

Figure 3-15
String width in Unicode mode

Unicode Byte and Character Counts

		Case Number	StringVar	ByteLength	CharLength
1		1	résumé	8.00	6.00
2		2	resume	6.00	6.00
Total	N		2	2	2

Variable Information

Variable	Position	Label	Measurement Level	Column Width	Alignment	Print Format	Write Format
StringVar	2	\<none\>	Nominal	26	Left	A24	A24

File Operations

You can combine and manipulate data sources in a number of ways, including:

- Using multiple data sources
- Merging data files
- Aggregating data
- Weighting data
- Changing file structure
- Using output as input (For more information, see "Using Output as Input with OMS" in Chapter 9 on p. 143.)

Using Multiple Data Sources

Starting with release 14.0, you can have multiple data sources open at the same time.

- When you use the dialog boxes and wizards in the graphical user interface to open data sources, the default behavior is to open each data source in a new Data Editor window, and any previously open data sources remain open and available for further use. You can change the active dataset simply by clicking anywhere in the Data Editor window of the data source that you want to use or by selecting the Data Editor window for that data source from the Window menu.

- In command syntax, the default behavior remains the same as in previous releases: reading a new data source automatically replaces the active dataset. If you want to work with multiple datasets using command syntax, you need to use the DATASET commands.

The DATASET commands (DATASET NAME, DATASET ACTIVATE, DATASET DECLARE, DATASET COPY, DATASET CLOSE) provide the ability to have multiple data sources open at the same time and control which open data source is active at any point in the session. Using defined dataset names, you can then:

- Merge data (for example, MATCH FILES, ADD FILES, UPDATE) from multiple different source types (for example, text data, database, spreadsheet) without saving each one as an external IBM® SPSS® Statistics data file first.

- Create new datasets that are subsets of open data sources (for example, males in one subset, females in another, people under a certain age in another, or original data in one set and transformed/computed values in another subset).

- Copy and paste variables, cases, and/or variable properties between two or more open data sources in the Data Editor.

Operations

- Commands operate on the active dataset. The **active** dataset is the data source most recently opened (for example, by commands such as GET DATA, GET SAS, GET STATA, GET TRANSLATE) or most recently activated by a DATASET ACTIVATE command.

  Note: The active dataset can also be changed by clicking anywhere in the Data Editor window of an open data source or selecting a dataset from the list of available datasets in a syntax window toolbar.

- Variables from one dataset are not available when another dataset is the active dataset.

- Transformations to the active dataset—before or after defining a dataset name—are preserved with the named dataset during the session, and any pending transformations to the active dataset are automatically executed whenever a different data source becomes the active dataset.

- Dataset names can be used in most commands that can contain references to SPSS Statistics data files.

- Wherever a dataset name, file handle (defined by the FILE HANDLE command), or filename can be used to refer to SPSS Statistics data files, defined dataset names take precedence over file handles, which take precedence over filenames. For example, if *file1* exists as both a dataset name and a file handle, FILE=file1 in the MATCH FILES command will be interpreted as referring to the dataset named *file1*, not the file handle.

Example

```
*multiple_datasets.sps.
DATA LIST FREE /file1Var.
BEGIN DATA
11 12 13
END DATA.
DATASET NAME file1.
COMPUTE file1Var=MOD(file1Var,10).
DATA LIST FREE /file2Var.
BEGIN DATA
21 22 23
END DATA.
DATASET NAME file2.
*file2 is now the active dataset; so the following
 command will generate an error.
FREQUENCIES VARIABLES=file1Var.
*now activate dataset file1 and rerun Frequencies.
DATASET ACTIVATE file1.
FREQUENCIES VARIABLES=file1Var.
```

- The first DATASET NAME command assigns a name to the active dataset (the data defined by the first DATA LIST command). This keeps the dataset open for subsequent use in the session after other data sources have been opened. Without this command, the dataset would automatically close when the next command that reads/opens a data source is run.

- The COMPUTE command applies a transformation to a variable in the active dataset. This transformation will be preserved with the dataset named *file1*. The order of the DATASET NAME and COMPUTE commands is not important. Any transformations to the active dataset, before or after assigning a dataset name, are preserved with that dataset during the session.

■ The second `DATA LIST` command creates a new dataset, which automatically becomes the active dataset. The subsequent `FREQUENCIES` command that specifies a variable in the first dataset will generate an error, because *file1* is no longer the active dataset, and there is no variable named *file1Var* in the active dataset.

■ `DATASET ACTIVATE` makes *file1* the active dataset again, and now the `FREQUENCIES` command will work.

Example

```
*dataset_subsets.sps.
DATASET CLOSE ALL.
DATA LIST FREE /gender.
BEGIN DATA
0 0 1 1 0 1 1 1 0 0
END DATA.
DATASET NAME original.
DATASET COPY males.
DATASET ACTIVATE males.
SELECT IF gender=0.
DATASET ACTIVATE original.
DATASET COPY females.
DATASET ACTIVATE females.
SELECT IF gender=1.
EXECUTE.
```

■ The first `DATASET COPY` command creates a new dataset, *males*, that represents the state of the active dataset at the time it was copied.

■ The *males* dataset is activated and a subset of males is created.

■ The original dataset is activated, restoring the cases deleted from the *males* subset.

■ The second `DATASET COPY` command creates a second copy of the original dataset with the name *females*, which is then activated and a subset of females is created.

■ Three different versions of the initial data file are now available in the session: the original version, a version containing only data for males, and a version containing only data for females.

Figure 4-1
Multiple subsets available in the same session
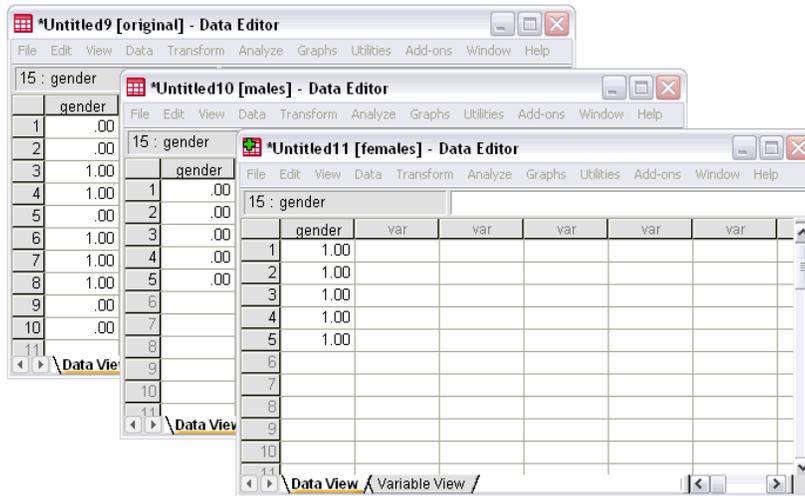
Merging Data Files

You can merge two or more datasets in several ways:

- Merge datasets with the same cases but different variables
- Merge datasets with the same variables but different cases
- Update values in a master data file with values from a transaction file

Merging Files with the Same Cases but Different Variables

The MATCH FILES command merges two or more data files that contain the same cases but different variables. For example, demographic data for survey respondents might be contained in one data file, and survey responses for surveys taken at different times might be contained in multiple additional data files. The cases are the same (respondents), but the variables are different (demographic information and survey responses).

This type of data file merge is similar to joining multiple database tables except that you are merging multiple IBM® SPSS® Statistics data files rather than database tables. For information on reading multiple database tables with joins, see "Reading Multiple Tables" in Chapter 3 on p. 22.

One-to-One Matches

The simplest type of match assumes that there is basically a one-to-one relationship between cases in the files being merged—for each case in one file, there is a corresponding case in the other file.

Example

This example merges a data file containing demographic data with another file containing survey responses for the same cases.

```
*match_files1.sps.
*first make sure files are sorted correctly.
GET FILE='/examples/data/match_response1.sav'.
SORT CASES BY id.
DATASET NAME responses.
GET FILE='/examples/data/match_demographics.sav'.
SORT CASES BY id.
*now merge the survey responses with the demographic info.
MATCH FILES /FILE=*
  /FILE=responses
  /BY id.
EXECUTE.
```

- `DATASET NAME` is used to name the first dataset, so it will remain available after the second dataset is opened.

- `SORT CASES BY id` is used to sort both datasets in the same case order. Cases are merged sequentially, so both datasets must be sorted in the same order to make sure that cases are merged correctly.

- `MATCH FILES` merges the two datasets. `FILE=*` indicates the active dataset (the demographic dataset).

- The `BY` subcommand matches cases by the value of the ID variable in both datasets. In this example, this is not technically necessary, since there is a one-to-one correspondence between cases in the two datasets and the datasets are sorted in the same case order. However, if the datasets are *not* sorted in the same order and no key variable is specified on the `BY` subcommand, the datasets will be merged incorrectly with no warnings or error messages; whereas, if a key variable is specified on the `BY` subcommand and the datasets are not sorted in the same order of the key variable, the merge will fail and an appropriate error message will be displayed. If the datasets contain a common case identifier variable, it is a good practice to use the `BY` subcommand.

- Any variables with the same name are assumed to contain the same information, and only the variable from the first dataset specified on the `MATCH FILES` command is included in the merged dataset. In this example, the ID variable (*id*) is present in both datasets, and the merged dataset contains the values of the variable from the demographic dataset—which is the first dataset specified on the `MATCH FILES` command. (In this case, the values are identical anyway.)

- For string variables, variables with the same name must have the same defined width in both files. If they have different defined widths, an error results and the command does not run. This includes string variables used as `BY` variables.

Example

Expanding the previous example, we will merge the same two data files plus a third data file that contains survey responses from a later date. Three aspects of this third file warrant special attention:

- The variable names for the survey questions are the same as the variable names in the survey response data file from the earlier date.

- One of the cases that is present in both the demographic data file and the first survey response file is missing from the new survey response data file.

- The source file is not in IBM® SPSS® Statistics format; it's an Excel worksheet.

```
*match_files2.sps.
GET FILE='/examples/data/match_response1.sav'.
SORT CASES BY id.
DATASET NAME response1.
GET DATA /TYPE=XLS
   /FILE='/examples/data/match_response2.xls'.
SORT CASES BY id.
DATASET NAME response2.
GET FILE='/examples/data/match_demographics.sav'.
SORT CASES BY id.
MATCH FILES /FILE=*
  /FILE=response1
  /FILE=response2
  /RENAME  opinion1=opinion1_2 opinion2=opinion2_2
   opinion3=opinion3_2 opinion4=opinion4_2
  /BY id.
EXECUTE.
```

- As before, all of the datasets are sorted by the values of the ID variable.

- MATCH FILES specifies three datasets this time: the active dataset that contains the demographic information and the two datasets containing survey responses from two different dates.

- The RENAME command after the FILE subcommand for the second survey response dataset provides new names for the survey response variables in that dataset. This is necessary to include these variables in the merged dataset. Otherwise, they would be excluded because the original variable names are the same as the variable names in the first survey response dataset.

- The BY subcommand is necessary in this example because one case (*id* = 184) is missing from the second survey response dataset, and without using the BY variable to match cases, the datasets would be merged incorrectly.

- All cases are included in the merged dataset. The case missing from the second survey response dataset is assigned the system-missing value for the variables from that dataset (*opinion1_2–opinion4_2*).

Figure 4-2
Merged files displayed in Data Editor

Table Lookup (One-to-Many) Matches

A **table lookup file** is a file in which data for each case can be applied to multiple cases in the other data file(s). For example, if one file contains information on individual family members (such as gender, age, education) and the other file contains overall family information (such as total income, family size, location), you can use the file of family data as a table lookup file and apply the common family data to each individual family member in the merged data file.

Specifying a file with the TABLE subcommand instead of the FILE subcommand indicates that the file is a table lookup file. The following example merges two text files, but they could be any combination of data source formats. For information on reading different types of data, see Chapter 3 on p. 19.

```
*match_table_lookup.sps.
DATA LIST LIST
  FILE='/examples/data/family_data.txt'
   /household_id total_income family_size region.
SORT CASES BY household_id.
DATASET NAME household.
DATA LIST LIST
  FILE='/examples/data/individual_data.txt'
  /household_id indv_id age gender education.
SORT CASE BY household_id.
DATASET NAME individual.
MATCH FILES TABLE='household'
  /FILE='individual'
  /BY household_id.
EXECUTE.
```

Merging Files with the Same Variables but Different Cases

The ADD FILES command merges two or more data files that contain the same variables but different cases. For example, regional revenue for two different company divisions might be stored in two separate data files. Both files have the same variables (region indicator and revenue) but different cases (each region for each division is a case).

Example

ADD FILES relies on variable names to determine which variables represent the "same" variables in the data files being merged. In the simplest example, all of the files contain the same set of variables, using the exact same variable names, and all you need to do is specify the files to be merged. In this example, the two files both contain the same two variables, with the same two variable names: *Region* and *Revenue*.

```
*add_files1.sps.
ADD FILES
  /FILE = '/examples/data/catalog.sav'
  /FILE ='/examples/data/retail.sav'
    /IN = Division.
EXECUTE.
VALUE LABELS Division 0 'Catalog' 1 'Retail Store'.
```

Figure 4-3
Cases from one file added to another file

- Cases are added to the active dataset in the order in which the source data files are specified on the ADD FILES command; all of the cases from *catalog.sav* appear first, followed by all of the cases from *retail.sav*.

- The IN subcommand after the FILE subcommand for *retail.sav* creates a new variable named *Division* in the merged dataset, with a value of 1 for cases that come from *retail.sav* and a value of 0 for cases that come from *catalog.sav*. (If the IN subcommand was placed immediately after the FILE subcommand for *catalog.sav*, the values would be reversed.)

- The VALUE LABELS command provides descriptive labels for the *Division* values of 0 and 1, identifying the division for each case in the merged dataset.

Example

Now that we've had a good laugh over the likelihood that all of the files have the exact same structure with the exact same variable names, let's look at a more realistic example. What if the revenue variable had a different name in one of the files and one of the files contained additional variables not present in the other files being merged?

```
*add_files2.sps.
***first throw some curves into the data***.
GET FILE = '/examples/data/catalog.sav'.
RENAME VARIABLES (Revenue=Sales).
DATASET NAME catalog.
GET FILE = '/examples/data/retail.sav'.
COMPUTE ExtraVar = 9.
EXECUTE.
DATASET NAME retail.
***show default behavior***.
ADD FILES
  /FILE = 'catalog'
  /FILE = 'retail'
    /IN = Division.
EXECUTE.
***now treat Sales and Revenue as same variable***.
***and drop ExtraVar from the merged file***.
ADD FILES
  /FILE = 'catalog'
    /RENAME (Sales = Revenue)
  /FILE = 'retail'
    /IN = Division
```

```
    /DROP ExtraVar
 /BY Region.
EXECUTE.
```

- All of the commands prior to the first ADD FILES command simply modify the original data files to contain minor variations—*Revenue* is changed to *Sales* in one data file, and an extra variable, *ExtraVar*, is added to the other data file.

- The first ADD FILES command is similar to the one in the previous example and shows the default behavior if nonmatching variable names and extraneous variables are not accounted for—the merged dataset has five variables instead of three, and it also has a lot of missing data. *Sales* and *Revenue* are treated as different variables, resulting in half of the cases having values for *Sales* and half of the cases having values for *Revenue*—and cases from the second data file have values for *ExtraVar*, but cases from the first data file do not, since this variable does not exist in that file.

Figure 4-4
Probably not what you want when you add cases from another file

- In the second ADD FILES command, the RENAME subcommand after the FILE subcommand for *catalog* will treat the variable *Sales* as if its name were *Revenue*, so the variable name will match the corresponding variable in *retail*.

- The DROP subcommand following the FILE subcommand for *temp2.sav* (and the associated IN subcommand) will exclude *ExtraVar* from the merged dataset. (The DROP subcommand must come after the FILE subcommand for the file that contains the variables to be excluded.)

- The BY subcommand adds cases to the merged data file in ascending order of values of the variable *Region* instead of adding cases in file order—but this requires that both files already be sorted in the same order of the BY variable.

Figure 4-5

Cases added in order of Region variable instead of file order

Updating Data Files by Merging New Values from Transaction Files

You can use the UPDATE command to replace values in a master file with updated values recorded in one or more files called transaction files.

```
*update.sps.
GET FILE = '/examples/data/update_transaction.sav'.
SORT CASE BY id.
DATASET NAME transaction.
GET FILE = '/examples/data/update_master.sav'.
SORT CASES BY id.
UPDATE /FILE = *
  /FILE = transaction
    /IN = updated
  /BY id.
EXECUTE.
```

- SORT CASES BY id is used to sort both files in the same case order. Cases are updated sequentially, so both files must be sorted in the same order.

- The first FILE subcommand on the UPDATE command specifies the master data file. In this example, FILE = * specifies the active dataset.

- The second FILE subcommand specifies the dataset name assigned to the transaction file.

- The IN subcommand immediately following the second FILE subcommand creates a new variable called *updated* in the master data file; this variable will have a value of 1 for any cases with updated values and a value of 0 for cases that have not changed.

- The BY subcommand matches cases by *id*. This subcommand is required. Transaction files often contain only a subset of cases, and a key variable is necessary to match cases in the two files.

Figure 4-6

Original file, transaction file, and update file

- The *salary* values for the cases with the *id* values of 103 and 201 are both updated.

- The *department* value for case 201 is updated, but the *department* value for case 103 is *not* updated. System-missing values in the transaction files do not overwrite existing values in the master file, so the transaction files can contain partial information for each case.

Aggregating Data

The AGGREGATE command creates a new dataset where each case represents one or more cases from the original dataset. You can save the aggregated data to a new dataset or replace the active dataset with aggregated data. You can also append the aggregated results as new variables to the current active dataset.

Example

In this example, information was collected for every person living in a selected sample of households. In addition to information for each individual, each case contains a variable that identifies the household. You can change the unit of analysis from individuals to households by aggregating the data based on the value of the household ID variable.

```
*aggregate1.sps.
***create some sample data***.
DATA LIST FREE (" ")
  /ID_household (F3) ID_person (F2) Income (F8).
BEGIN DATA
```

```
101 1 12345 101 2 47321 101 3 500 101 4 0
102 1 77233 102 2 0
103 1 19010 103 2 98277 103 3 0
104 1 101244
END DATA.
***now aggregate based on household id***.
AGGREGATE
  /OUTFILE = * MODE = REPLACE
  /BREAK = ID_household
  /Household_Income = SUM(Income)
  /Household_Size = N.
```

- `OUTFILE = * MODE = REPLACE` replaces the active dataset with the aggregated data.

- `BREAK = ID_household` combines cases based on the value of the household ID variable.

- `Household_Income = SUM(Income)` creates a new variable in the aggregated dataset that is the total income for each household.

- `Household_Size = N` creates a new variable in the aggregated dataset that is the number of original cases in each aggregated case.

Figure 4-7
Original and aggregated data

Example

You can also use `MODE = ADDVARIABLES` to add group summary information to the original data file. For example, you could create two new variables in the original data file that contain the number of people in the household and the per capita income for the household (total income divided by number of people in the household).

```
*aggregate2.sps.
DATA LIST FREE (" ")
  /ID_household (F3) ID_person (F2) Income (F8).
BEGIN DATA
101 1 12345 101 2 47321 101 3 500 101 4 0
102 1 77233 102 2 0
103 1 19010 103 2 98277 103 3 0
104 1 101244
END DATA.
AGGREGATE
```

```
/OUTFILE = * MODE = ADDVARIABLES
/BREAK = ID_household
/per_capita_Income = MEAN(Income)
/Household_Size = N.
```

■ As with the previous example, OUTFILE = * specifies the active dataset as the target for the aggregated results.

■ Instead of replacing the original data with aggregated data, MODE = ADDVARIABLES will add aggregated results as new variables to the active dataset.

■ As with the previous example, cases will be aggregated based on the household ID value.

■ The MEAN function will calculate the per capita household incomes.

Figure 4-8
Aggregate summary data added to original data

Aggregate Summary Functions

The new variables created when you aggregate a data file can be based on a wide variety of numeric and statistical functions applied to each group of cases defined by the BREAK variables, including:

■ Number of cases in each group

■ Sum, mean, median, and standard deviation

■ Minimum, maximum, and range

■ Percentage of cases between, above, and/or below specified values

■ First and last nonmissing value in each group

■ Number of missing values in each group

For a complete list of aggregate functions, see the AGGREGATE command in the *Command Syntax Reference*.

Weighting Data

The WEIGHT command simulates case replication by treating each case as if it were actually the number of cases indicated by the value of the weight variable. You can use a weight variable to adjust the distribution of cases to more accurately reflect the larger population or to simulate raw data from aggregated data.

Example

A sample data file contains 52% males and 48% females, but you know that in the larger population the real distribution is 49% males and 51% females. You can compute and apply a weight variable to simulate this distribution.

```
*weight_sample.sps.
***create sample data of 52 males, 48 females***.
NEW FILE.
INPUT PROGRAM.
- STRING gender (A6).
- LOOP #I =1 TO 100.
-   DO IF #I <= 52.
-     COMPUTE gender='Male'.
-   ELSE.
-     COMPUTE Gender='Female'.
-   END IF.
-   COMPUTE AgeCategory = trunc(uniform(3)+1).
-   END CASE.
- END LOOP.
- END FILE.
END INPUT PROGRAM.
FREQUENCIES VARIABLES=gender AgeCategory.
***create and apply weightvar***.
***to simulate 49 males, 51 females***.
DO IF gender = 'Male'.
- COMPUTE weightvar=49/52.
ELSE IF gender = 'Female'.
- COMPUTE weightvar=51/48.
END IF.
WEIGHT BY weightvar.
FREQUENCIES VARIABLES=gender AgeCategory.
```

- Everything prior to the first FREQUENCIES command simply generates a sample dataset with 52 males and 48 females.

- The DO IF structure sets one value of *weightvar* for males and a different value for females. The formula used here is: *desired proportion/observed proportion*. For males, it is 49/52 (0.94), and for females, it is 51/48 (1.06).

- The WEIGHT command weights cases by the value of *weightvar*, and the second FREQUENCIES command displays the weighted distribution.

Note: In this example, the weight values have been calculated in a manner that does not alter the total number of cases. If the weighted number of cases exceeds the original number of cases, tests of significance are inflated; if it is smaller, they are deflated. More flexible and reliable weighting techniques are available in the Complex Samples add-on module.

Example

You want to calculate measures of association and/or significance tests for a crosstabulation, but all you have to work with is the summary table, not the raw data used to construct the table. The table looks like this:

	Male	Female	Total
Under $50K	25	35	60
$50K+	30	10	40
Total	55	45	100

You then read the data using rows, columns, and cell counts as variables; then, use the cell count variable as a weight variable.

```
*weight.sps.
DATA LIST LIST /Income Gender count.
BEGIN DATA
1, 1, 25
1, 2, 35
2, 1, 30
2, 2, 10
END DATA.
VALUE LABELS
  Income 1 'Under $50K' 2 '$50K+'
  /Gender 1 'Male' 2 'Female'.
WEIGHT BY count.
CROSSTABS TABLES=Income by Gender
  /STATISTICS=CC PHI.
```

- The values for *Income* and *Gender* represent the row and column positions from the original table, and *count* is the value that appears in the corresponding cell in the table. For example, `1, 2, 35` indicates that the value in the first row, second column is 35. (The *Total* row and column are not included.)

- The `VALUE LABELS` command assigns descriptive labels to the numeric codes for *Income* and *Gender*. In this example, the value labels are the row and column labels from the original table.

- The `WEIGHT` command weights cases by the value of *count*, which is the number of cases in each cell of the original table.

- The `CROSSTABS` command produces a table very similar to the original and provides statistical tests of association and significance.

Figure 4-9
Crosstabulation and significance tests for reconstructed table

Income * Gender Crosstabulation

		Gender		Total
		Male	Female	
Income	Under $50K	25	35	60
	$50K+	30	10	40
Total		55	45	100

Symmetric Measures

		Value	Approx. Sig.
Nominal by Nominal	Phi	-.328	.001
	Cramer's V	.328	.001
	Contingency Coefficient	.312	.001
N of Valid Cases		100	

Changing File Structure

IBM® SPSS® Statistics expects data to be organized in a certain way, and different types of analysis may require different data structures. Since your original data can come from many different sources, the data may require some reorganization before you can create the reports or analyses that you want.

Transposing Cases and Variables

You can use the FLIP command to create a new data file in which the rows and columns in the original data file are transposed so that cases (rows) become variables and variables (columns) become cases.

Example

Although IBM® SPSS® Statistics expects cases in the rows and variables in the columns, applications such as Excel don't have that kind of data structure limitation. So what do you do with an Excel file in which cases are recorded in the columns and variables are recorded in the rows?

Figure 4-10
Excel file with cases in columns, variables in rows

Here are the commands to read the Excel spreadsheet and transpose the rows and columns:

```
*flip_excel.sps.
GET DATA /TYPE=XLS
  /FILE='/examples/data/flip_excel.xls'
  /READNAMES=ON .
FLIP VARIABLES=Newton Boris Kendall Dakota Jasper Maggie
  /NEWNAME=V1.
RENAME VARIABLES (CASE_LBL = Name).
```

- `READNAMES=ON` in the `GET DATA` command reads the first row of the Excel spreadsheet as variable names. Since the first cell in the first row is blank, it is assigned a default variable name of *V1*.

- The `FLIP` command creates a new active dataset in which all of the variables specified will become cases and all cases in the file will become variables.

- The original variable names are automatically stored as values in a new variable called *CASE_LBL*. The subsequent `RENAME VARIABLES` command changes the name of this variable to *Name*.

- `NEWNAME=V1` uses the values of variable *V1* as variable names in the transposed data file.

Figure 4-11
Original and transposed data in Data Editor
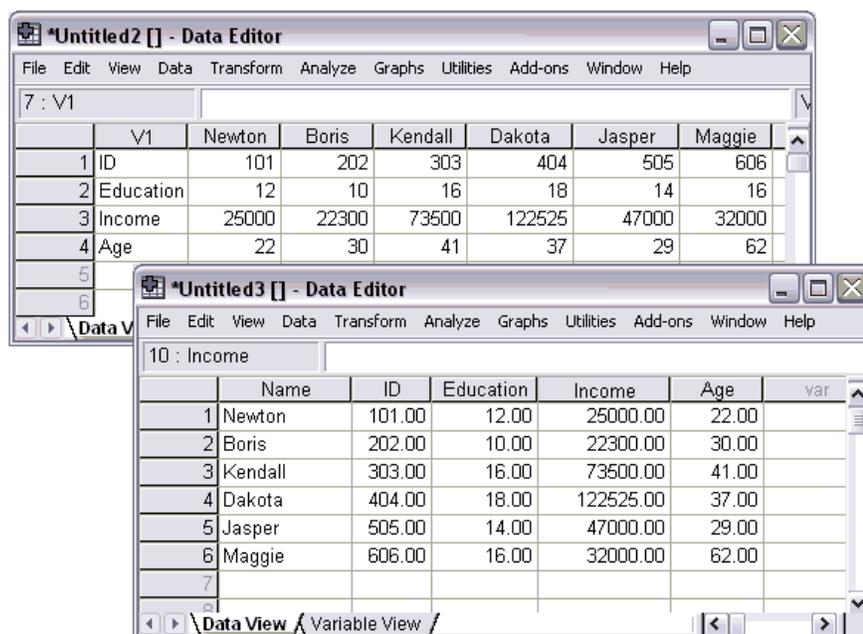
Cases to Variables

Sometimes you may need to restructure your data in a slightly more complex manner than simply flipping rows and columns.

Many statistical techniques in IBM® SPSS® Statistics are based on the assumption that cases (rows) represent independent observations and/or that related observations are recorded in separate variables rather than separate cases. If a data file contains groups of related cases, you may not be

able to use the appropriate statistical techniques (for example, the paired samples *t* test or repeated measures GLM) because the data are not organized in the required fashion for those techniques.

In this example, we use a data file that is very similar to the data used in the AGGREGATE example. For more information, see the topic "Aggregating Data" on p. 65. Information was collected for every person living in a selected sample of households. In addition to information for each individual, each case contains a variable that identifies the household. Cases in the same household represent related observations, not independent observations, and we want to restructure the data file so that each group of related cases is one case in the restructured file and new variables are created to contain the related observations.

Figure 4-12
Data file before restructuring cases to variables

The CASESTOVARS command combines the related cases and produces the new variables.

```
*casestovars.sps.
GET FILE = '/examples/data/casestovars.sav'.
SORT CASES BY ID_household.
CASESTOVARS
 /ID = ID_household
 /INDEX = ID_person
 /SEPARATOR = "_"
 /COUNT = famsize.
VARIABLE LABELS
  Income_1 "Husband/Father Income"
  Income_2 "Wife/Mother Income"
  Income_3 "Other Income".
```

■ SORT CASES sorts the data file by the variable that will be used to group cases in the CASESTOVARS command. The data file must be sorted by the variable(s) specified on the ID subcommand of the CASESTOVARS command.

■ The ID subcommand of the CASESTOVARS command indicates the variable(s) that will be used to group cases together. In this example, all cases with the same value for *ID_household* will become a single case in the restructured file.

■ The optional INDEX subcommand identifies the original variables that will be used to create new variables in the restructured file. Without the INDEX subcommand, all unique values of all non-ID variables will generate variables in the restructured file. In this example, only values of *ID_person* will be used to generate new variables. Index variables can be either

string or numeric. Numeric index values must be nonmissing, positive integers; string index values cannot be blank.

- The SEPARATOR subcommand specifies the character(s) that will be used to separate original variable names and the values appended to those names for the new variable names in the restructured file. By default, a period is used. You can use any characters that are allowed in a valid variable name (which means the character cannot be a space). If you do not want any separator, specify a null string (SEPARATOR = " ").

- The COUNT subcommand will create a new variable that indicates the number of original cases represented by each combined case in the restructured file.

- The VARIABLE LABELS command provides descriptive labels for the new variables in the restructured file.

Figure 4-13
Data file after restructuring cases to variables

Variables to Cases

The previous example turned related cases into related variables for use with statistical techniques that compare and contrast related samples. But sometimes you may need to do the exact opposite—convert variables that represent unrelated observations to variables.

Example

A simple Excel file contains two columns of information: income for males and income for females. There is no known or assumed relationship between male and female values that are recorded in the same row; the two columns represent independent (unrelated) observations, and we want to create cases (rows) from the columns (variables) and create a new variable that indicates the gender for each case.

Figure 4-14
Data file before restructuring variables to cases

The VARSTOCASES command creates cases from the two columns of data.
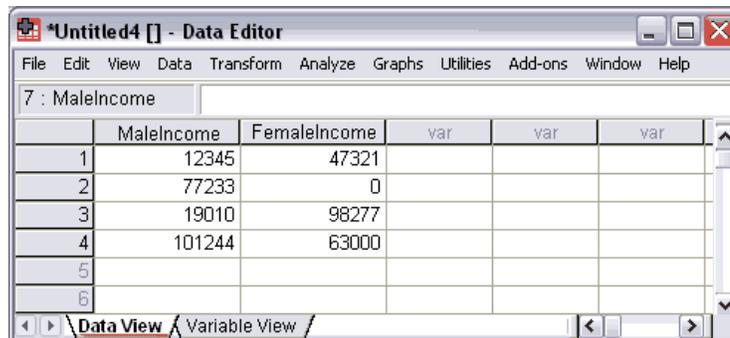
```
*varstocases1.sps.
GET DATA /TYPE=XLS
 /FILE = '/examples/data/varstocases.xls'
 /READNAMES = ON.
VARSTOCASES
 /MAKE Income FROM MaleIncome FemaleIncome
 /INDEX = Gender.
VALUE LABELS Gender 1 'Male' 2 'Female'.
```

■ The MAKE subcommand creates a single income variable from the two original income variables.

■ The INDEX subcommand creates a new variable named *Gender* with integer values that represent the sequential order in which the original variables are specified on the MAKE subcommand. A value of 1 indicates that the new case came from the original male income column, and a value of 2 indicates that the new case came from the original female income column.

■ The VALUE LABELS command provides descriptive labels for the two values of the new *Gender* variable.

Figure 4-15
Data file after restructuring variables to cases
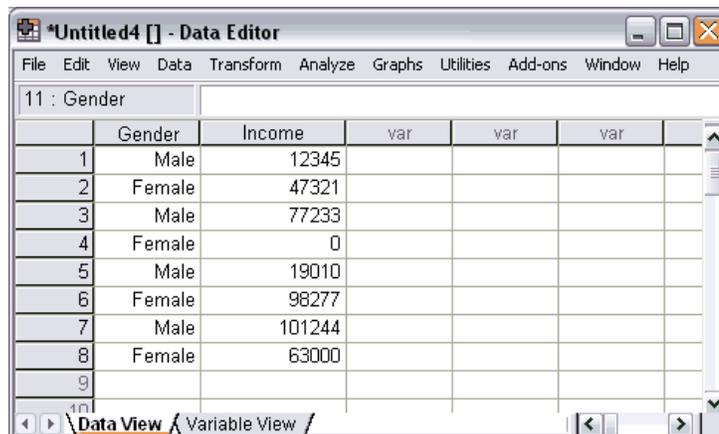
Example

In this example, the original data contain separate variables for two measures taken at three separate times for each case. This is the correct data structure for many procedures that compare related observations. There are, however, some procedures (for example, MIXED and GENLINMIXED in the Advanced Statistics option) that require a data structure in which related observations are recorded as separate cases.

Figure 4-16
Related observations recorded as separate variables


```
*varstocases2.sps.
GET FILE = '/examples/data/varstocases.sav'.
VARSTOCASES  /MAKE V1 FROM V1_Time1 V1_Time2 V1_Time3
 /MAKE V2 FROM V2_Time1 V2_Time2 V2_Time3
 /INDEX = Time
 /KEEP = ID Age.
```

■ The two MAKE subcommands create two variables, one for each group of three related variables.

■ The INDEX subcommand creates a variable named *Time* that indicates the sequential order of the original variables used to create the cases, as specified on the MAKE subcommand.

■ The KEEP subcommand retains the original variables *ID* and *Age*.

Figure 4-17
Related variables restructured into cases

Variable and File Properties

In addition to the basic data type (numeric, string, date, and so forth), you can assign other properties that describe the variables and their associated values. You can also define properties that apply to the entire data file. In a sense, these properties can be considered **metadata**—data that describe the data. These properties are automatically saved with the data when you save the data in IBM® SPSS® Statistics data files.

Variable Properties

You can use variable attributes to provide descriptive information about data and control how data are treated in analyses, charts, and reports.

- Variable labels and value labels provide descriptive information that make it easier to understand your data and results.

- Missing value definitions and measurement level affect how variables and specific data values are treated by statistical and charting procedures.

Example

```
*define_variables.sps.
DATA LIST LIST
  /id (F3) Interview_date (ADATE10) Age (F3) Gender (A1)
   Income_category (F1) Religion (F1) opinion1 to opinion4 (4F1).
BEGIN DATA
150 11/1/2002 55 m 3 4 5 1 3 1
272 10/24/02 25 f 3 9 2 3 4 3
299 10-24-02 900 f 8 4 2 9 3 4
227 10/29/2002 62 m 9 4 2 3 5 3
216 10/26/2002 39 F 7 3 9 3 2 1
228 10/30/2002 24 f 4 2 3 5 1 5
333 10/29/2002 30 m 2 3 5 1 2 3
385 10/24/2002 23 m 4 4 3 3 9 2
170 10/21/2002 29 f 4 2 2 2 2 5
391 10/21/2002 58 m 1 3 5 1 5 3
END DATA.
FREQUENCIES VARIABLES=opinion3 Income_Category.
VARIABLE LABELS
  Interview_date "Interview date"
  Income_category "Income category"
  opinion1 "Would buy this product"
  opinion2 "Would recommend this product to others"
  opinion3 "Price is reasonable"
  opinion4 "Better than a poke in the eye with a sharp stick".
VALUE LABELS
  Gender "m" "Male" "f" "Female"
  /Income_category 1 "Under 25K" 2 "25K to 49K" 3 "50K to 74K" 4 "75K+"
  7 "Refused to answer" 8 "Don't know" 9 "No answer"
  /Religion 1 "Catholic" 2 "Protestant" 3 "Jewish" 4 "Other" 9 "No answer"
  /opinion1 TO opinion4 1 "Strongly Disagree" 2 "Disagree" 3 "Ambivalent"
  4 "Agree" 5 "Strongly Agree" 9 "No answer".
MISSING VALUES
  Income_category (7, 8, 9)
  Religion opinion1 TO opinion4 (9).
VARIABLE LEVEL
  Income_category, opinion1 to opinion4 (ORDINAL).
FREQUENCIES VARIABLES=opinion3 Income_Category.
```

Figure 5-1
Frequency tables before assigning variable properties

opinion3

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1	1	10.0	10.0	10.0
	2	3	30.0	30.0	40.0
	3	2	20.0	20.0	60.0
	4	1	10.0	10.0	70.0
	5	2	20.0	20.0	90.0
	9	1	10.0	10.0	100.0
	Total	10	100.0	100.0	

Income_category

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1	1	10.0	10.0	10.0
	2	1	10.0	10.0	20.0
	3	2	20.0	20.0	40.0
	4	3	30.0	30.0	70.0
	7	1	10.0	10.0	80.0
	8	1	10.0	10.0	90.0
	9	1	10.0	10.0	100.0
	Total	10	100.0	100.0	

■ The first FREQUENCIES command, run before any variable properties are assigned, produces the preceding frequency tables.

■ For both variables in the two tables, the actual numeric values do not mean a great deal by themselves, since the numbers are really just codes that represent categorical information.

■ For *opinion3*, the variable name itself does not convey any particularly useful information either.

■ The fact that the reported values for *opinion3* go from 1 to 5 and then jump to 9 may mean something, but you really cannot tell what.

Figure 5-2
Frequency tables after assigning variable properties

Price is reasonable

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Strongly Disagree	1	10.0	11.1	11.1
	Disagree	3	30.0	33.3	44.4
	Ambivalent	2	20.0	22.2	66.7
	Agree	1	10.0	11.1	77.8
	Strongly Agree	2	20.0	22.2	100.0
	Total	9	90.0	100.0	
Missing	No answer	1	10.0		
Total		10	100.0		

Income category

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Under 25K	1	10.0	14.3	14.3
	25K to 49K	1	10.0	14.3	28.6
	50K to 74K	2	20.0	28.6	57.1
	75K+	3	30.0	42.9	100.0
	Total	7	70.0	100.0	
Missing	Refused to answer	1	10.0		
	Don't know	1	10.0		
	No answer	1	10.0		
	Total	3	30.0		
Total		10	100.0		

- The second FREQUENCIES command is exactly the same as the first, except this time it is run after a number of properties have been assigned to the variables.

- By default, any defined variable labels and value labels are displayed in output instead of variable names and data values. You can also choose to display variable names and/or data values or to display both names/values and variable and value labels. (See the SET command and the TVARS and TNUMBERS subcommands in the *Command Syntax Reference*.)

- User-defined missing values are flagged for special handling. Many procedures and computations automatically exclude user-defined missing values. In this example, missing values are displayed separately and are not included in the computation of *Valid Percent* or *Cumulative Percent*.

- If you save the data as IBM® SPSS® Statistics data files, variable labels, value labels, missing values, and other variable properties are automatically saved with the data file. You do not need to reassign variable properties every time you open the data file.

Variable Labels

The VARIABLE LABELS command provides descriptive labels up to 255 bytes. Variable names can be up to 64 bytes, but variable names cannot contain spaces and cannot contain certain characters. For more information, see "Variables" in the "Universals" section of the *Command Syntax Reference*.

```
VARIABLE LABELS
  Interview_date "Interview date"
  Income_category "Income category"
  opinion1 "Would buy this product"
  opinion2 "Would recommend this product to others"
```

```
opinion3 "Price is reasonable"
opinion4 "Better than a poke in the eye with a sharp stick".
```

■ The variable labels *Interview date* and *Income category* do not provide any additional information, but their appearance in the output is better than the variable names with underscores where spaces would normally be.

■ For the four opinion variables, the descriptive variable labels are more informative than the generic variable names.

Value Labels

You can use the VALUE LABELS command to assign descriptive labels for each value of a variable. This is particularly useful if your data file uses numeric codes to represent non-numeric categories. For example, *income_category* uses the codes 1 through 4 to represent different income ranges, and the four opinion variables use the codes 1 through 5 to represent level of agreement/disagreement.

```
VALUE LABELS
  Gender "m" "Male" "f" "Female"
  /Income_category 1 "Under 25K" 2 "25K to 49K" 3 "50K to 74K" 4 "75K+"
  7 "Refused to answer" 8 "Don't know" 9 "No answer"
  /Religion 1 "Catholic" 2 "Protestant" 3 "Jewish" 4 "Other" 9 "No answer"
  /opinion1 TO opinion4 1 "Strongly Disagree" 2 "Disagree" 3 "Ambivalent"
  4 "Agree" 5 "Strongly Agree" 9 "No answer".
```

■ Value labels can be up to 120 bytes.

■ For string variables, both the values and the labels need to be enclosed in quotes. Also, remember that string values are case sensitive; "f" "Female" is *not* the same as "F" "Female".

■ You cannot assign value labels to long string variables (string variables longer than eight characters).

■ Use ADD VALUE LABELS to define additional value labels without deleting existing value labels.

Missing Values

The MISSING VALUES command identifies specified data values as **user-missing**. It is often useful to know why information is missing. For example, you might want to distinguish between data that is missing because a respondent refused to answer and data that is missing because the question did not apply to that respondent. Data values specified as user-missing are flagged for special treatment and are excluded from most calculations.

```
MISSING VALUES
  Income_category (7, 8, 9)
  Religion opinion1 TO opinion4 (9).
```

■ You can assign up to three discrete (individual) missing values, a range of missing values, or a range plus one discrete value.

■ Ranges can be specified only for numeric variables.

■ You cannot assign missing values to long string variables (string variables longer than eight characters).

Measurement Level

You can assign measurement levels (nominal, ordinal, scale) to variables with the VARIABLE LEVEL command.

```
VARIABLE LEVEL
  Income_category, opinion1 to opinion4 (ORDINAL).
```

■ By default, all new string variables are automatically assigned a nominal measurement level, and numeric variables are automatically assigned either the nominal or scale measurement level, based on the evaluation of numerous criteria, including number of unique values, presence of non-integer or negative values, and data format.

■ In this example, the default rules work well for many variables, but there is no rule for assigning the ordinal measurement level, and there are five ordinal variables in the dataset.

■ The opinion variables and *Income_category* are all categorical variables with a meaningful order to the categories; so we use the VARIABLE LEVEL command to assign the ordinal measurement level to those variables.

Custom Variable Properties

You can use the VARIABLE ATTRIBUTE command to create and assign custom variable attributes.

Example

```
*variable_attributes.sps.
DATA LIST LIST /ID Age Region Income1 Income2 Income3.
BEGIN DATA
1 27 1 35500 42700 40250
2 34 2 72300 75420 81000
3 50 1 85400 82900 84350
END DATA.
COMPUTE AvgIncome=MEAN(Income1, Income2, Income3).
COMPUTE MaxIncome=MAX(Income1, Income2, Income3).
VARIABLE ATTRIBUTE
  VARIABLES=AvgIncome
  ATTRIBUTE=Formula('mean(Income1, Income2, Income3)')
 /VARIABLES=MaxIncome
  ATTRIBUTE=Formula('max(Income1, Income2, Income3)')
 /VARIABLES=AvgIncome MaxIncome
  ATTRIBUTE=DerivedFrom[1]('Income1')
  DerivedFrom[2]('Income2')
  DerivedFrom[3]('Income3')
 /VARIABLES=ALL ATTRIBUTE=Notes('').
```

- The attributes *Formula* and *DerivedFrom* are assigned to the two computed variables. Each variable has a different value for *Formula*, which describes the code used to compute the value. For *DerivedFrom*, which lists the variables used to compute the values, both variables have the same attribute values.

- The attribute *DerivedFrom* is an **attribute array**. The value in square brackets defines the position within the array. The highest value specified defines the total number of array elements. For example,

  ```
  ATTRIBUTE=MyAtt[20] ('')
  ```

  would create an array of 20 attributes (*MyAtt[1]*, *MyAtt[2]*, *MyAtt[3]*, ... *MyAtt[20]*).

- The attribute *Notes* is assigned to all variables and is assigned a null value.

Use DISPLAY ATTRIBUTES to display a table of all defined attributes and their values. You can also display and modify attribute values in Variable View of the Data Editor (View menu, Display Custom Attributes).

Figure 5-3
Custom Variable Attributes in Variable View

- Custom variable attribute names are enclosed in square brackets.

- Attribute names that begin with a dollar sign are reserved and cannot be modified.

- A blank cell indicates that the attribute does not exist for that variable; the text *Empty* displayed in a cell indicates that the attribute exists for that variable but no value has been assigned to the attribute for that variable. Once you enter text in the cell, the attribute exists for that variable with the value you enter.

- The text *Array...* displayed in a cell indicates that this is an attribute array—an attribute that contains multiple values. Click the button in the cell to display the list of values.

Using Variable Properties as Templates

You can reuse the assigned variable properties in a data file as templates for new data files or other variables in the same data file, selectively applying different properties to different variables.

Example

The data and the assigned variable properties at the beginning of this chapter are saved in the IBM® SPSS® Statistics data file *variable_properties.sav*. In this example, we apply some of those variable properties to a new data file with similar variables.

```
*apply_properties.sps.
DATA LIST LIST
  /id (F3) Interview_date (ADATE10) Age (F3) Gender (A1) Income_category (F1)
   attitude1 to attitude4(4F1).
BEGIN DATA
456 11/1/2002 55 m 3 5 1 3 1
789 10/24/02 25 f 3 2 3 4 3
131 10-24-02 900 f 8 2 9 3 4
659 10/29/2002 62 m 9 2 3 5 3
217 10/26/2002 39 f 7 9 3 2 1
399 10/30/2002 24 f 4 3 5 1 5
end data.
APPLY DICTIONARY
  /FROM '/examples/data/variable_properties.sav'
  /SOURCE VARIABLES = Interview_date Age Gender Income_category
  /VARINFO ALL.
APPLY DICTIONARY
  /FROM '/examples/data/variable_properties.sav'
  /SOURCE VARIABLES =  opinion1
  /TARGET VARIABLES =  attitude1 attitude2 attitude3 attitude4
  /VARINFO LEVEL MISSING VALLABELS.
```

■ The first APPLY DICTIONARY command applies all variable properties from the specified SOURCE VARIABLES in *variable_properties.sav* to variables in the new data file with matching names and data types. For example, *Income_category* in the new data file now has the same variable label, value labels, missing values, and measurement level (and a few other properties) as the variable of the same name in the source data file.

■ The second APPLY DICTIONARY command applies selected properties from the variable *opinion1* in the source data file to the four attitude variables in the new data file. The selected properties are measurement level, missing values, and value labels.

■ Since it is unlikely that the variable label for *opinion1* would be appropriate for all four attitude variables, the variable label is not included in the list of properties to apply to the variables in the new data file.

File Properties

File properties, such as a descriptive file label or comments that describe the change history of the data, are useful for data that you plan to save and store in IBM® SPSS® Statistics format.

Example

```
*file_properties.sps.
DATA LIST FREE /var1.
BEGIN DATA
1 2 3
END DATA.
FILE LABEL
  Fake data generated with Data List and inline data.
ADD DOCUMENT
  'Original version of file prior to transformations.'.
DATAFILE ATTRIBUTE ATTRIBUTE=VersionNumber ('1').
SAVE OUTFILE='/temp/temp.sav'.
NEW FILE.
```

```
GET FILE '/temp/temp.sav'.
DISPLAY DOCUMENTS.
DISPLAY ATTRIBUTES.
```

Figure 5-4
File properties displayed in output

Notes

Output Created		03-JAN-2006 14:35:54
Comments		
Input	Data	c:\temp\temp.sav
	File Label	Fake data generated with Data List and inline data
	Filter	<none>
	Weight	<none>
	Split File	<none>
Syntax		DISPLAY DOCUMENTS.
Resources	Elapsed Time	0:00:00.00

Document

1ᵃ	Original version of file prior to transformations.

a. Entered 03-Jan-2006

Datafile Attributes

Attribute	Value
VersionNumber	1

- ■ FILE LABEL creates a descriptive label of up to 64 bytes. The label is displayed in the Notes table.

- ■ ADD DOCUMENT saves a block of text of any length, along with the date the text was added to the data file. The text from each ADD DOCUMENT command is appended to the end of the list of documentation. Use DROP DOCUMENTS to delete all document text. Use DISPLAY DOCUMENTS to display document text.

- ■ DATAFILE ATTRIBUTE creates custom file attributes. You can create data file attribute arrays using the same conventions used for defining variable attribute arrays. For more information, see the topic "Custom Variable Properties" on p. 81. Use DISPLAY ATTRIBUTES to display custom attribute values.

Data Transformations

In an ideal situation, your raw data are perfectly suitable for the reports and analyses that you need. Unfortunately, this is rarely the case. Preliminary analysis may reveal inconvenient coding schemes or coding errors, and data transformations may be required in order to coax out the true relationship between variables.

You can perform data transformations ranging from simple tasks, such as collapsing categories for reports, to more advanced tasks, such as creating new variables based on complex equations and conditional statements.

Recoding Categorical Variables

You can use the RECODE command to change, rearrange, and/or consolidate values of a variable. For example, questionnaires often use a combination of high-low and low-high rankings. For reporting and analysis purposes, you probably want these all coded in a consistent manner.

```
*recode.sps.
DATA LIST FREE /opinion1 opinion2.
BEGIN DATA
1 5
2 4
3 3
4 2
5 1
END DATA.
RECODE opinion2
 (1 = 5) (2 = 4) (4 = 2) (5 = 1)
 (ELSE = COPY)
 INTO opinion2_new.
EXECUTE.
VALUE LABELS opinion1 opinion2_new
 1 'Really bad' 2 'Bad' 3 'Blah'
 4 'Good' 5 'Terrific!'.
```

■ The RECODE command essentially reverses the values of *opinion2*.

■ ELSE = COPY retains the value of 3 (which is the middle value in either direction) and any other unspecified values, such as user-missing values, which would otherwise be set to system-missing for the new variable.

■ INTO creates a new variable for the recoded values, leaving the original variable unchanged.

Binning Scale Variables

Creating a small number of discrete categories from a continuous scale variable is sometimes referred to as **binning**. For example, you can recode salary data into a few salary range categories. Although it is not difficult to write command syntax to bin a scale variable into range categories, we recommend that you try the Visual Binning dialog box, available on the Transform menu, because it can help you make the best recoding choices by showing the actual distribution of values and where your selected category boundaries occur in the distribution. It also provides

Chapter 6

a number of different binning methods and can automatically generate descriptive labels for the binned categories.

Figure 6-1
Visual Binning dialog box
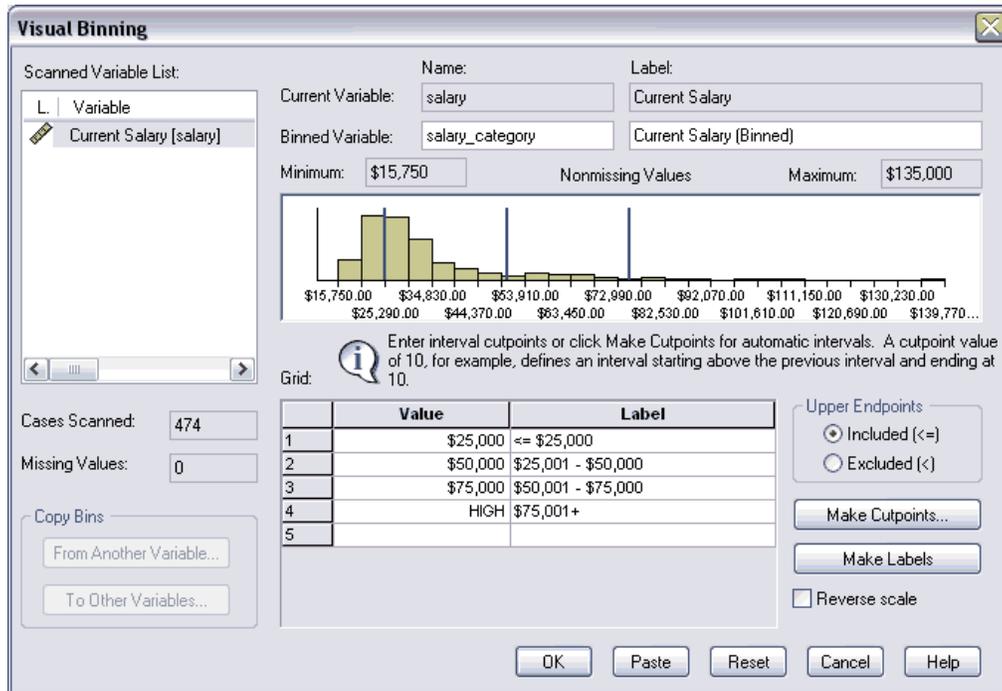
■ The histogram shows the distribution of values for the selected variable. The vertical lines indicate the binned category divisions for the specified range groupings.

■ In this example, the range groupings were automatically generated using the Make Cutpoints dialog box, and the descriptive category labels were automatically generated with the Make Labels button.

■ You can use the Make Cutpoints dialog box to create binned categories based on equal width intervals, equal percentiles (equal number of cases in each category), or standard deviations.

Figure 6-2
Make Cutpoints dialog box
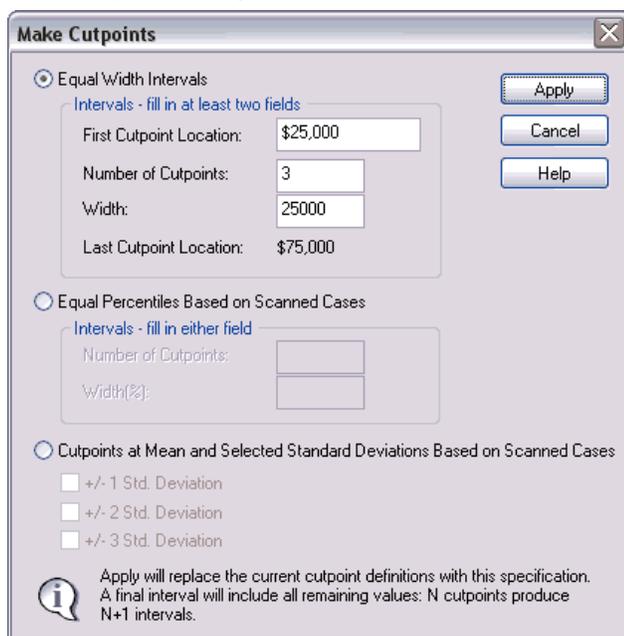
You can use the Paste button in the Visual Binning dialog box to paste the command syntax for your selections into a command syntax window. The RECODE command syntax generated by the Binning dialog box provides a good model for a proper recoding method.

```
*visual_binning.sps.
****commands generated by visual binning dialog***.
RECODE salary
      ( MISSING = COPY )
      ( LO THRU 25000 =1 )
      ( LO THRU 50000 =2 )
      ( LO THRU 75000 =3 )
      ( LO THRU HI = 4 )
      ( ELSE = SYSMIS ) INTO salary_category.
VARIABLE LABELS salary_category 'Current Salary (Binned)'.
FORMAT salary_category (F5.0).
VALUE LABELS salary_category
        1   '<= $25,000'
        2   '$25,001 - $50,000'
        3   '$50,001 - $75,000'
        4   '$75,001+'
        0   'missing'.
MISSING VALUES salary_category ( 0 ).
VARIABLE LEVEL salary_category ( ORDINAL ).
EXECUTE.
```

■ The RECODE command encompasses all possible values of the original variable.

■ MISSING = COPY preserves any user-missing values from the original variable. Without this, user-missing values could be inadvertently combined into a nonmissing category for the new variable.

■ The general recoding scheme of LO THRU *value* ensures that no values fall through the cracks. For example, 25001 THRU 50000 would not include a value of 25000.50.

- Since the RECODE expression is evaluated from left to right and each original value is recoded only once, each subsequent range specification can start with LO because this means the lowest remaining value that has not already been recoded.

- LO THRU HI includes all remaining values (other than system-missing) not included in any of the other categories, which in this example should be any salary value above $75,000.

- INTO creates a new variable for the recoded values, leaving the original variable unchanged. Since binning or combining/collapsing categories can result in loss of information, it is a good idea to create a new variable for the recoded values rather than overwriting the original variable.

- The VALUE LABELS and MISSING VALUES commands generated by the Binning dialog box preserve the user-missing category and its label from the original variable.

Simple Numeric Transformations

You can perform simple numeric transformations using the standard programming language notation for addition, subtraction, multiplication, division, exponents, and so on.

```
*numeric_transformations.sps.
DATA LIST FREE /var1.
BEGIN DATA
1 2 3 4 5
END DATA.
COMPUTE var2 = 1.
COMPUTE var3 = var1*2.
COMPUTE var4 = ((var1*2)**2)/2.
EXECUTE.
```

- COMPUTE var2 = 1 creates a constant with a value of 1.

- COMPUTE var3 = var1*2 creates a new variable that is twice the value of *var1*.

- COMPUTE var4 = ((var1*2)**2)/2 first multiplies *var1* by 2, then squares that value, and finally divides the result by 2.

Arithmetic and Statistical Functions

In addition to simple arithmetic operators, you can also transform data with a wide variety of functions, including arithmetic and statistical functions.

```
*numeric_functions.sps.
DATA LIST LIST (",") /var1 var2 var3 var4.
BEGIN DATA
1, , 3, 4
5, 6, 7, 8
9, , , 12
END DATA.
COMPUTE Square_Root = SQRT(var4).
COMPUTE Remainder = MOD(var4, 3).
COMPUTE Average = MEAN.3(var1, var2, var3, var4).
COMPUTE Valid_Values = NVALID(var1 TO var4).
COMPUTE Trunc_Mean = TRUNC(MEAN(var1 TO var4)).
EXECUTE.
```

- All functions take one or more arguments, enclosed in parentheses. Depending on the function, the arguments can be constants, expressions, and/or variable names—or various combinations thereof.

- `SQRT(var4)` returns the square root of the value of *var4* for each case.

- `MOD(var4, 3)` returns the remainder (modulus) from dividing the value of *var4* by 3.

- `MEAN.3(var1, var2, var3, var4)` returns the mean of the four specified variables, provided that at least three of them have nonmissing values. The divisor for the calculation of the mean is the number of nonmissing values.

- `NVALID(var1 TO var4)` returns the number of valid, nonmissing values for the inclusive range of specified variables. For example, if only two of the variables have nonmissing values for a particular case, the value of the computed variable is 2 for that case.

- `TRUNC(MEAN(var1 TO var4))` computes the mean of the values for the inclusive range of variables and then truncates the result. Since no minimum number of nonmissing values is specified for the `MEAN` function, a mean will be calculated (and truncated) as long as at least one of the variables has a nonmissing value for that case.

Figure 6-3
Variables computed with arithmetic and statistical functions

For a complete list of arithmetic and statistical functions, see "Transformation Expressions" in the "Universals" section of the *Command Syntax Reference*.

Random Value and Distribution Functions

Random value and distribution functions generate random values based on the specified type of distribution and parameters, such as mean, standard deviation, or maximum value.

```
*random_functons.sps.
NEW FILE.
SET SEED 987987987.
*create 1,000 cases with random values.
INPUT PROGRAM.
- LOOP #I=1 TO 1000.
-    COMPUTE Uniform_Distribution = UNIFORM(100).
-    COMPUTE Normal_Distribution = RV.NORMAL(50,25).
-    COMPUTE Poisson_Distribution = RV.POISSON(50).
-    END CASE.
- END LOOP.
- END FILE.
```

```
END INPUT PROGRAM.
FREQUENCIES VARIABLES = ALL
  /HISTOGRAM /FORMAT = NOTABLE.
```

- The `INPUT PROGRAM` uses a `LOOP` structure to generate 1,000 cases.
- For each case, `UNIFORM(100)` returns a random value from a uniform distribution with values that range from 0 to 100.
- `RV.NORMAL(50,25)` returns a random value from a normal distribution with a mean of 50 and a standard deviation of 25.
- `RV.POISSON(50)` returns a random value from a Poisson distribution with a mean of 50.
- The `FREQUENCIES` command produces histograms of the three variables that show the distributions of the randomly generated values.

Figure 6-4
Histograms of randomly generated values for different distributions
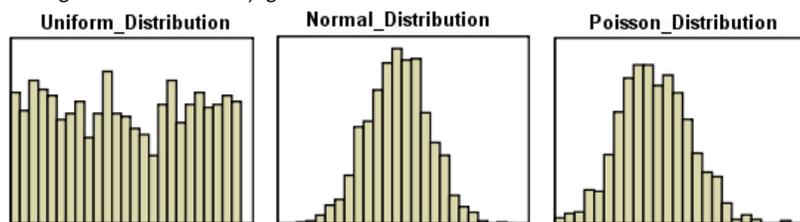
Random variable functions are available for a variety of distributions, including Bernoulli, Cauchy, and Weibull. For a complete list of random variable functions, see "Random Variable and Distribution Functions" in the "Universals" section of the *Command Syntax Reference*.

String Manipulation

Since just about the only restriction you can impose on string variables is the maximum number of characters, string values may often be recorded in an inconsistent manner and/or contain important bits of information that would be more useful if they could be extracted from the rest of the string.

Changing the Case of String Values

Perhaps the most common problem with string values is inconsistent capitalization. Since string values are case sensitive, a value of "male" is *not* the same as a value of "Male." This example converts all values of a string variable to lowercase letters.

```
*string_case.sps.
DATA LIST FREE /gender (A6).
BEGIN DATA
Male Female
male female
MALE FEMALE
END DATA.
COMPUTE gender=LOWER(gender).
EXECUTE.
```

- The LOWER function converts all uppercase letters in the value of *gender* to lowercase letters, resulting in consistent values of "male" and "female."

- You can use the UPCASE function to convert string values to all uppercase letters.

Combining String Values

You can combine multiple string and/or numeric values to create new string variables. For example, you could combine three numeric variables for area code, exchange, and number into one string variable for telephone number with dashes between the values.

```
*concat_string.sps.
DATA LIST FREE /tel1 tel2 tel3 (3F4).
BEGIN DATA
111 222 3333
222 333 4444
333 444 5555
555 666  707
END DATA.
STRING telephone (A12).
COMPUTE telephone =
  CONCAT((STRING(tel1, N3)), "-",
         (STRING(tel2, N3)), "-",
         (STRING(tel3, N4))).
EXECUTE.
```

- The STRING command defines a new string variable that is 12 characters long. Unlike new numeric variables, which can be created by transformation commands, you must define new string variables before using them in any transformations.

- The COMPUTE command combines two string manipulation functions to create the new telephone number variable.

- The CONCAT function concatenates two or more string values. The general form of the function is CONCAT(string1, string2, ...). Each argument can be a variable name, an expression, or a literal string enclosed in quotes.

- Each argument of the CONCAT function must evaluate to a string; so we use the STRING function to treat the numeric values of the three original variables as strings. The general form of the function is STRING(value, format). The value argument can be a variable name, a number, or an expression. The format argument must be a valid numeric format. In this example, we use N format to support leading zeros in values (for example, 0707).

- The dashes in quotes are literal strings that will be included in the new string value; a dash will be displayed between the area code and exchange and between the exchange and number.

Figure 6-5

Original numeric values and concatenated string values

Taking Strings Apart

In addition to being able to combine strings, you can also take them apart.

Example

A dataset contains telephone numbers recorded as strings. You want to create separate variables for the three values that comprise the phone number. You know that each number contains 10 digits—but some contain spaces and/or dashes between the three portions of the number, and some do not.

```
*replace_substr.sps.
***Create some inconsistent sample numbers***.
DATA LIST FREE (",") /telephone (A16).
BEGIN DATA
111-222-3333
222 - 333 - 4444
333 444 5555
4445556666
555-666-0707
END DATA.
*First remove all extraneous spaces and dashes.
STRING #telstr (A16).
COMPUTE #telstr=REPLACE(telephone, " ", "").
COMPUTE #telstr=REPLACE(#telstr, "-", "").
*Now extract the parts.
COMPUTE tel1=NUMBER(CHAR.SUBSTR(#telstr, 1, 3), F5).
COMPUTE tel2=NUMBER(CHAR.SUBSTR(#telstr, 4, 3), F5).
COMPUTE tel3=NUMBER(CHAR.SUBSTR(#telstr, 7), F5).
EXECUTE.
FORMATS tel1 tel2 (N3) tel3 (N4).
```

- The first task is to remove any spaces or dashes from the values, which is accomplished with the two REPLACE functions. The spaces and dashes are replaced with null strings, and the telephone number without any dashes or spaces is stored in the temporary variable *#telstr*.

- The NUMBER function converts a number expressed as a string to a numeric value. The basic format is NUMBER(value, format). The value argument can be a variable name, a number expressed as a string in quotes, or an expression. The format argument must be a valid numeric format; this format is used to determine the numeric value of the string. In other words, the format argument says, "Read the string as if it were a number in this format."

- The value argument for the NUMBER function for all three new variables is an expression using the CHAR.SUBSTR function. The general form of the function is CHAR.SUBSTR(value, position, length). The value argument can be a variable name, an expression, or a literal string enclosed in quotes. The position argument is a number that indicates the starting character position within the string. The optional length argument is a number that specifies how many characters to read starting at the value specified on the position argument. Without the length argument, the string is read from the specified starting position to the end of the string value. So CHAR.SUBSTR("abcd", 2, 2) would return "bc," and CHAR.SUBSTR("abcd", 2) would return "bcd."

- For *tel1*, SUBSTR(#telstr, 1, 3) defines a substring three characters long, starting with the first character in the original string.

- For *tel2*, CHAR.SUBSTR(#telstr, 4, 3) defines a substring three characters long, starting with the fourth character in the original string.

- For *tel3*, CHAR.SUBSTR(#telstr, 7) defines a substring that starts with the seventh character in the original string and continues to the end of the value.

- FORMATS assigns N format to the three new variables for numbers with leading zeros (for example, 0707).

Figure 6-6
Substrings extracted and converted to numbers

Example

This example takes a single variable containing first, middle, and last name and creates three separate variables for each part of the name. Unlike the example with telephone numbers, you can't identify the start of the middle or last name by an absolute position number, because you don't know how many characters are contained in the preceding parts of the name. Instead, you need to find the location of the spaces in the value to determine the end of one part and the start of the next—and some values only contain a first and last name, with no middle name.

```
*substr_index.sps.
DATA LIST FREE (",") /name (A20).
BEGIN DATA
Hugo Hackenbush
Rufus T. Firefly
Boris Badenoff
Rocket J. Squirrel
END DATA.
STRING #n fname mname lname(a20).
```

```
COMPUTE #n = name.
VECTOR vname=fname TO lname.
LOOP #i = 1 to 2.
- COMPUTE #space = CHAR.INDEX(#n," ").
- COMPUTE vname(#i) = CHAR.SUBSTR(#n,1,#space-1).
- COMPUTE #n = CHAR.SUBSTR(#n,#space+1).
END LOOP.
COMPUTE lname=#n.
DO IF lname="".
- COMPUTE lname=mname.
- COMPUTE mname="".
END IF.
EXECUTE.
```

- A temporary (scratch) variable, *#n*, is declared and set to the value of the original variable. The three new string variables are also declared.

- The VECTOR command creates a vector *vname* that contains the three new string variables (in file order).

- The LOOP structure iterates twice to produce the values for *fname* and *mname*.

- COMPUTE #space = CHAR.INDEX(#n," ") creates another temporary variable, *#space*, that contains the position of the first space in the string value.

- On the first iteration, COMPUTE vname(#i) = CHAR.SUBSTR(#n,1,#space-1) extracts everything prior to the first dash and sets *fname* to that value.

- COMPUTE #n = CHAR.SUBSTR(#n,#space+1) then sets *#n* to the remaining portion of the string value *after* the first space.

- On the second iteration, COMPUTE #space... sets *#space* to the position of the "first" space in the modified value of *#n*. Since the first name and first space have been removed from *#n*, this is the position of the space between the middle and last names.

  Note: If there is no middle name, then the position of the "first" space is now the first space after the end of the last name. Since string values are right-padded to the defined width of the string variable, and the defined width of *#n* is the same as the original string variable, there should always be at least one blank space at the end of the value after removing the first name.

- COMPUTE vname(#i)... sets *mname* to the value of everything up to the "first" space in the modified version of *#n*, which is everything after the first space and before the second space in the original string value. If the original value doesn't contain a middle name, then the last name will be stored in *mname*. (We'll fix that later.)

- COMPUTE #n... then sets *#n* to the remaining segment of the string value—everything after the "first" space in the modified value, which is everything after the second space in the original value.

- After the two loop iterations are complete, COMPUTE lname=#n sets *lname* to the final segment of the original string value.

- The DO IF structure checks to see if the value of *lname* is blank. If it is, then the name had only two parts to begin with, and the value currently assigned to *mname* is moved to *lname*.

Figure 6-7
Substring extraction using CHAR.INDEX function

Changing Data Types and String Widths

In release 16.0 or later, you can use the ALTER TYPE command to:

- Change the fundamental data type (numeric or string) of a variable
- Automatically change the width of string variables based on the width of observed values
- Conditionally change variable format for multiple variables based on the current format

Example: Change String Variables to Numeric

```
*string_to_number.sps.
DATA LIST FREE /StringNumber (A3) StringDate(A10).
BEGIN DATA
123 10/28/2007
abc 10/29/2008
END DATA.
VALUE LABELS
  StringNumber '123' 'Numeric value'
  'abc' 'String Value'.
MISSING VALUES StringNumber ('999' 'def').
ALTER TYPE StringNumber (F3) StringDate (ADATE10).
DISPLAY DICTIONARY.
```

- *StringNumber* is converted from a string A3 format to a numeric F3 format.
- *StringDate* is converted from a string A10 format to a numeric ADATE10 date format.

When converting from string to numeric, values containing characters that are invalid in the specified numeric format are converted to system-missing, and value labels or missing values definitions for invalid values are deleted. In this example:

- The string value 'abc' is converted to numeric system-missing.
- The value label for the string value '123' is preserved as the value label for the number 123, while the value label for 'abc' is deleted.
- The user-missing value for the string '999' is converted to a user-missing value for the number 999, while the user-missing value of 'def' is deleted.

Example: Change String Widths Based on Observed Values

```
*change_string_width.sps.
DATA LIST LIST /FName (A10) LName (A20).
BEGIN DATA
Robert Terwilliger
Edna Krabappel
Joe Quimby
END DATA.
ALTER TYPE ALL (A=AMIN).
DISPLAY DICTIONARY.
```

- ALTER TYPE ALL (A=AMIN) changes the width of all string variables to the minimum width required to preserve the observed values without truncation.

- *FName* is changed from A10 to A6, and *LName* is changed from A20 to A11.

Example: Conditionally Change Type and Format

```
*change_string_conditional.sps.
DATA LIST LIST
 /ID (F3) Gender (A1) Grade (A1) BirthDate (A10) ExamDate (A10).
BEGIN DATA
123 F B 10/28/1986 5/20/2007
456 M C 6/20/1986 8/13/2007
789 F A 10/29/1987 9/10/2007
END DATA.
ALTER TYPE ALL (A10=ADATE10).
DISPLAY DICTIONARY.
```

- In the previous example (A=AMIN) applied to all A format variables. In this example (A10=ADATE10) applies only to A10 format variables.

- The string variables *BirthDate* and *ExamDate* are converted to numeric date variables, while the string variables *Gender* and *Grade* are unchanged.

Working with Dates and Times

Dates and times come in a wide variety of formats, ranging from different display formats (for example, 10/28/1986 versus 28-OCT-1986) to separate entries for each component of a date or time (for example, a day variable, a month variable, and a year variable). Various features are available for dealing with dates and times, including:

- Support for multiple input and display formats for dates and times.

- Storing dates and times internally as consistent numbers regardless of the input format, making it possible to compare date/time values and calculate the difference between values even if they were not entered in the same format.

- Functions that can convert string dates to real dates; extract portions of date values (such as simply the month or year) or other information that is associated with a date (such as day of the week); and create calendar dates from separate values for day, month, and year.

Date Input and Display Formats

IBM® SPSS® Statistics automatically converts date information from databases, Excel files, and SAS files to equivalent SPSS Statistics date format variables. SPSS Statistics can also recognize dates in text data files stored in a variety of formats. All you need to do is specify the appropriate format when reading the text data file.

Date format	General form	Example	Date format specification
International date	dd-mmm-yyyy	28-OCT-2003	DATE
American date	mm/dd/yyyy	10/28/2003	ADATE
Sortable date	yyyy/mm/dd	2003/10/28	SDATE
Julian date	yyyyddd	2003301	JDATE
Time	hh:mm:ss	11:35:43	TIME
Days and time	dd hh:mm:ss	15 08:27:12	DTIME
Date and time	dd-mmm-yyyy hh:mm:ss	20-JUN-2003 12:23:01	DATETIME
Day of week	(name of day)	Tuesday	WKDAY
Month of year	(name of month)	January	MONTH

Note: For a complete list of date and time formats, see "Date and Time" in the "Universals" section of the *Command Syntax Reference*.

Example

```
DATA LIST FREE(" ")
 /StartDate(ADATE) EndDate(DATE).
BEGIN DATA
10/28/2002 28-01-2003
10-29-02 15,03,03
01.01.96 01/01/97
1/1/1997 01-JAN-1998
END DATA.
```

■ Both two- and four-digit year specifications are recognized. Use SET EPOCH to set the starting year for two-digit years.

■ Dashes, periods, commas, slashes, or blanks can be used as delimiters in the day-month-year input.

■ Months can be represented in digits, Roman numerals, or three-character abbreviations, and they can be fully spelled out. Three-letter abbreviations and fully spelled out month names must be English month names; month names in other languages are not recognized.

■ In time specifications, colons can be used as delimiters between hours, minutes, and seconds. Hours and minutes are required, but seconds are optional. A period is required to separate seconds from fractional seconds. Hours can be of unlimited magnitude, but the maximum value for minutes is 59 and for seconds is 59.999….

■ Internally, dates and date/times are stored as the number of seconds from October 14, 1582, and times are stored as the number of seconds from midnight.

Note: SET EPOCH has no effect on existing dates in the file. You must set this value before reading or entering date values. The actual date stored internally is determined when the date is read; changing the epoch value afterward will not change the century for existing date values in the file.

Using FORMATS to Change the Display of Dates

Dates in IBM® SPSS® Statistics are often referred to as date format variables because the dates you see are really just display formats for underlying numeric values. Using the FORMATS command, you can change the display formats of a date format variable, including changing to a format that displays only a certain portion of the date, such as the month or day of the week.

Example

```
FORMATS StartDate(DATE11).
```

- A date originally displayed as 10/28/02 would now be displayed as 28-OCT-2002.
- The number following the date format specifies the display width. DATE9 would display as 28-OCT-02.

Some of the other format options are shown in the following table:

Original display format	New format specification	New display format
10/28/02	SDATE11	2002/10/28
10/28/02	WKDAY7	MONDAY
10/28/02	MONTH12	OCTOBER
10/28/02	MOYR9	OCT 2002
10/28/02	QYR6	4 Q 02

The underlying values remain the same; only the display format changes with the FORMATS command.

Converting String Dates to Date Format Numeric Variables

Under some circumstances, IBM® SPSS® Statistics may read valid date formats as string variables instead of date format numeric variables. For example, if you use the Text Wizard to read text data files, the wizard reads dates as string variables by default. If the string date values conform to one of the recognized date formats, it is easy to convert the strings to date format numeric variables.

Example

```
COMPUTE numeric_date = NUMBER(string_date, ADATE)
FORMATS numeric_date (ADATE10).
```

- The NUMBER function indicates that any numeric string values should be converted to those numbers.

- `ADATE` tells the program to assume that the strings represent dates of the general form mm/dd/yyyy. It is important to specify the date format that corresponds to the way the dates are represented in the string variable, since string dates that do not conform to that format will be assigned the system-missing value for the new numeric variable.

- The `FORMATS` command specifies the date display format for the new numeric variable. Without this command, the values of the new variable would be displayed as very large integers.

In version 16.0 or later, `ALTER TYPE` provides an alternative solution that can be applied to multiple variables and doesn't require creating new variables for the converted values. For more information, see the topic "Changing Data Types and String Widths" on p. 95.

Date and Time Functions

Many date and time functions are available, including:

- Aggregation functions to create a single date variable from multiple other variables representing day, month, and year.

- Conversion functions to convert from one date/time measurement unit to another—for example, converting a time interval expressed in seconds to number of days.

- Extraction functions to obtain different types of information from date and time values—for example, obtaining just the year from a date value, or the day of the week associated with a date.

Note: Date functions that take date values or year values as arguments interpret two-digit years based on the century defined by `SET EPOCH`. By default, two-digit years assume a range beginning 69 years prior to the current date and ending 30 years after the current date. When in doubt, use four-digit year values.

Aggregating Multiple Date Components into a Single Date Format Variable

Sometimes, dates and times are recorded as separate variables for each unit of the date. For example, you might have separate variables for day, month, and year or separate hour and minute variables for time. You can use the `DATE` and `TIME` functions to combine the constituent parts into a single date/time variable.

Example

```
COMPUTE datevar=DATE.MDY(month, day, year).
COMPUTE monthyear=DATE.MOYR(month, year).
COMPUTE time=TIME.HMS(hours, minutes).
FORMATS datevar (ADATE10) monthyear (MOYR9) time(TIME9).
```

- `DATE.MDY` creates a single date variable from three separate variables for month, day, and year.

- `DATE.MOYR` creates a single date variable from two separate variables for month and year. Internally, this is stored as the same value as the first day of that month.

- TIME.HMS creates a single time variable from two separate variables for hours and minutes.
- The FORMATS command applies the appropriate display formats to each of the new date variables.

For a complete list of DATE and TIME functions, see "Date and Time" in the "Universals" section of the *Command Syntax Reference*.

Calculating and Converting Date and Time Intervals

Since dates and times are stored internally in seconds, the result of date and time calculations is also expressed in seconds. But if you want to know how much time elapsed between a start date and an end date, you probably do not want the answer in seconds. You can use CTIME functions to calculate and convert time intervals from seconds to minutes, hours, or days.

Example

```
*date_functions.sps.
DATA LIST FREE (",")
  /StartDate (ADATE12) EndDate (ADATE12)
  StartDateTime(DATETIME20) EndDateTime(DATETIME20)
  StartTime (TIME10) EndTime (TIME10).
BEGIN DATA
3/01/2003, 4/10/2003
01-MAR-2003 12:00, 02-MAR-2003 12:00
09:30, 10:15
END DATA.
COMPUTE days = CTIME.DAYS(EndDate-StartDate).
COMPUTE hours = CTIME.HOURS(EndDateTime-StartDateTime).
COMPUTE minutes = CTIME.MINUTES(EndTime-StartTime).
EXECUTE.
```

- CTIME.DAYS calculates the difference between *EndDate* and *StartDate* in days—in this example, 40 days.
- CTIME.HOURS calculates the difference between *EndDateTime* and *StartDateTime* in hours—in this example, 24 hours.
- CTIME.MINUTES calculates the difference between *EndTime* and *StartTime* in minutes—in this example, 45 minutes.

Calculating Number of Years between Dates

You can use the DATEDIFF function to calculate the difference between two dates in various duration units. The general form of the function is

```
DATEDIFF(datetime2, datetime1, "unit")
```

where *datetime2* and *datetime1* are both date or time format variables (or numeric values that represent valid date/time values), and "unit" is one of the following string literal values enclosed in quotes: years, quarters, months, weeks, hours, minutes, or seconds.

Example

```
*datediff.sps.
DATA LIST FREE /BirthDate StartDate EndDate (3ADATE).
```

```
BEGIN DATA
8/13/1951 11/24/2002 11/24/2004
10/21/1958 11/25/2002 11/24/2004
END DATA.
COMPUTE Age=DATEDIFF($TIME, BirthDate, 'years').
COMPUTE DurationYears=DATEDIFF(EndDate, StartDate, 'years').
COMPUTE DurationMonths=DATEDIFF(EndDate, StartDate, 'months').
EXECUTE.
```

■ Age in years is calculated by subtracting *BirthDate* from the current date, which we obtain from the system variable *$TIME*.

■ The duration of time between the start date and end date variables is calculated in both years and months.

■ The DATEDIFF function returns the truncated integer portion of the value in the specified units. In this example, even though the two start dates are only one day apart, that results in a one-year difference in the values of *DurationYears* for the two cases (and a one-month difference for *DurationMonths*).

Adding to or Subtracting from a Date to Find Another Date

If you need to calculate a date that is a certain length of time before or after a given date, you can use the TIME.DAYS function.

Example

Prospective customers can use your product on a trial basis for 30 days, and you need to know when the trial period ends—just to make it interesting, if the trial period ends on a Saturday or Sunday, you want to extend it to the following Monday.

```
*date_functions2.sps.
DATA LIST FREE (" ") /StartDate (ADATE10).
BEGIN DATA
10/29/2003 10/30/2003
10/31/2003 11/1/2003
11/2/2003 11/4/2003
11/5/2003 11/6/2003
END DATA.
COMPUTE expdate = StartDate + TIME.DAYS(30).
FORMATS expdate (ADATE10).
***if expdate is Saturday or Sunday, make it Monday***.
DO IF (XDATE.WKDAY(expdate) = 1).
-  COMPUTE expdate = expdate + TIME.DAYS(1).
ELSE IF (XDATE.WKDAY(expdate) = 7).
-  COMPUTE expdate = expdate + TIME.DAYS(2).
END IF.
EXECUTE.
```

■ TIME.DAYS(30) adds 30 days to *StartDate*, and then the new variable *expdate* is given a date display format.

■ The DO IF structure uses an XDATE.WKDAY extraction function to see if *expdate* is a Sunday (1) or a Saturday (7), and then adds one or two days, respectively.

Example

You can also use the DATESUM function to calculate a date that is a specified length of time before or after a specified date.

```
*datesum.sps.
DATA LIST FREE /StartDate (ADATE).
BEGIN DATA
10/21/2003
10/28/2003
10/29/2004
END DATA.
COMPUTE ExpDate=DATESUM(StartDate, 3, 'years').
EXECUTE.
FORMATS ExpDate(ADATE10).
```

■ *ExpDate* is calculated as a date three years after *StartDate*.

■ The DATESUM function returns the date value in standard numeric format, expressed as the number of seconds since the start of the Gregorian calendar in 1582; so, we use FORMATS to display the value in one of the standard date formats.

Extracting Date Information

A great deal of information can be extracted from date and time variables. In addition to using XDATE functions to extract the more obvious pieces of information, such as year, month, day, hour, and so on, you can obtain information such as day of the week, week of the year, or quarter of the year.

Example

```
*date_functions3.sps.
DATA LIST FREE (",")
  /StartDateTime (datetime25).
BEGIN DATA
29-OCT-2003 11:23:02
1 January 1998 1:45:01
21/6/2000 2:55:13
END DATA.
COMPUTE dateonly=XDATE.DATE(StartDateTime).
FORMATS dateonly(ADATE10).
COMPUTE hour=XDATE.HOUR(StartDateTime).
COMPUTE DayofWeek=XDATE.WKDAY(StartDateTime).
COMPUTE WeekofYear=XDATE.WEEK(StartDateTime).
COMPUTE quarter=XDATE.QUARTER(StartDateTime).
EXECUTE.
```

Figure 6-8
Extracted date information

	StartDateTime	dateonly	hour	DayofWeek	WeekofYear	quarter
1	29-OCT-2003	10/29/2003	11.00	4.00	44.00	4.00
2	01-JAN-1998	01/01/1998	1.00	5.00	1.00	1.00
3	21-JUN-2000	06/21/2000	2.00	4.00	25.00	2.00
4						
5						

- The date portion extracted with XDATE.DATE returns a date expressed in seconds; so, we also include a FORMATS command to display the date in a readable date format.

- Day of the week is an integer between 1 (Sunday) and 7 (Saturday).

- Week of the year is an integer between 1 and 53 (January 1–7 = 1).

For a complete list of XDATE functions, see "Date and Time" in the "Universals" section of the *Command Syntax Reference*.

Cleaning and Validating Data

Invalid—or at least questionable—data values can include anything from simple out-of-range values to complex combinations of values that should not occur.

Finding and Displaying Invalid Values

The first step in cleaning and validating data is often to simply identify and investigate questionable values.

Example

All of the variables in a file may have values that appear to be valid when examined individually, but certain combinations of values for different variables may indicate that at least one of the variables has either an invalid value or at least one that is suspect. For example, a pregnant male clearly indicates an error in one of the values, whereas a pregnant female older than 55 may not be invalid but should probably be double-checked.

```
*invalid_data3.sps.
DATA LIST FREE /age gender pregnant.
BEGIN DATA
25 0 0
12 1 0
80 1 1
47 0 0
34 0 1
9 1 1
19 0 0
27 0 1
END DATA.
VALUE LABELS gender 0 'Male' 1 'Female'
  /pregnant 0 'No' 1 'Yes'.
DO IF pregnant = 1.
- DO IF gender = 0.
-   COMPUTE valueCheck = 1.
- ELSE IF gender = 1.
-   DO IF age > 55.
-     COMPUTE valueCheck = 2.
-   ELSE IF age < 12.
-     COMPUTE valueCheck = 3.
-   END IF.
- END IF.
ELSE.
- COMPUTE valueCheck=0.
END IF.
VALUE LABELS valueCheck
  0 'No problems detected'
  1 'Male and pregnant'
  2 'Age > 55 and pregnant'
  3 'Age < 12 and pregnant'.
FREQUENCIES VARIABLES = valueCheck.
```

■ The variable *valueCheck* is first set to 0.

- The outer DO IF structure restricts the actions for all transformations within the structure to cases recorded as pregnant (pregnant = 1).

- The first nested DO IF structure checks for males (gender = 0) and assigns those cases a value of 1 for *valueCheck*.

- For females (gender = 1), a second nested DO IF structure, nested within the previous one, is initiated, and *valueCheck* is set to 2 for females over the age of 55 and 3 for females under the age of 12.

- The VALUE LABELS command assigns descriptive labels to the numeric values of *valueCheck*, and the FREQUENCIES command generates a table that summarizes the results.

Figure 7-1
Frequency table summarizing detected invalid or suspect values

valueCheck

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	No problems detected	4	50.0	50.0	50.0
	Male and pregnant	2	25.0	25.0	75.0
	Age > 55 and pregnant	1	12.5	12.5	87.5
	Age < 12 and pregnant	1	12.5	12.5	100.0
	Total	8	100.0	100.0	

Example

A data file contains a variable *quantity* that represents the number of products sold to a customer, and the only valid values for this variable are integers. The following command syntax checks for and then reports all cases with non-integer values.

```
*invalid_data.sps.
*First we provide some simple sample data.
DATA LIST FREE /quantity.
BEGIN DATA
1 1.1 2 5 8.01
END DATA.
*Now we look for non-integers values
 in the sample data.
COMPUTE filtervar=(MOD(quantity,1)>0).
FILTER BY filtervar.
SUMMARIZE
  /TABLES=quantity
  /FORMAT=LIST CASENUM NOTOTAL
  /CELLS=COUNT.
FILTER OFF.
```

Figure 7-2
Table listing all cases with non-integer values

	Case Number	quantity
1	2	1.10
2	5	8.01
N		2

- The COMPUTE command creates a new variable, *filtervar*. If the remainder (the MOD function) of the original variable (*quantity*) divided by 1 is greater than 0, then the expression is true and *filtervar* will have a value of 1, resulting in all non-integer values of *quantity* having a value of 1 for *filtervar*. For integer values, *filtervar* is set to 0.

- The FILTER command filters out any cases with a value of 0 for the specified filter variable. In this example, it will filter out all of the cases with integer values for *quantity*, since they have a value of 0 for *filtervar*.

- The SUMMARIZE command simply lists all of the nonfiltered cases, providing the case number and the value of *quantity* for each case, as well as a table listing all of the cases with non-integer values.

- The second FILTER command turns off filtering, making all cases available for subsequent procedures.

Excluding Invalid Data from Analysis

With a slight modification, you can change the computation of the filter variable in the above example to filter out cases with invalid values:

```
COMPUTE filtrvar=(MOD(quantity,1)=0).
FILTER BY filtrvar.
```

- Now all cases with integer values for *quantity* have a value of 1 for the filter variable, and all cases with non-integer values for *quantity* are filtered out because they now have a value of 0 for the filter variable.

- This solution filters out the entire case, including valid values for other variables in the data file. If, for example, another variable recorded total purchase price, any case with an invalid value for *quantity* would be excluded from computations involving total purchase price (such as average total purchase price), even if that case has a valid value for total purchase price.

A better solution is to assign invalid values to a user-missing category, which identifies values that should be excluded or treated in a special manner for that specific variable, leaving other variables for cases with invalid values for *quantity* unaffected.

```
*invalid_data2.sps.
DATA LIST FREE /quantity.
BEGIN DATA
1 1.1 2 5 8.01
END DATA.
IF (MOD(quantity,1) > 0) quantity = (-9).
MISSING VALUES quantity (-9).
VALUE LABELS quantity -9 "Non-integer values".
```

- The IF command assigns a value of −9 to all non-integer values of *quantity*.

- The MISSING VALUES command flags *quantity* values of −9 as user-missing, which means that these values will either be excluded or treated in a special manner by most procedures.

- The VALUE LABELS command assigns a descriptive label to the user-missing value.

Finding and Filtering Duplicates

Duplicate cases may occur in your data for many reasons, including:

- Data-entry errors in which the same case is accidently entered more than once.

■ Multiple cases that share a common primary ID value but have different secondary ID values, such as family members who live in the same house.

■ Multiple cases that represent the same case but with different values for variables other than those that identify the case, such as multiple purchases made by the same person or company for different products or at different times.

The Identify Duplicate Cases dialog box (Data menu) provides a number of useful features for finding and filtering duplicate cases. You can paste the command syntax from the dialog box selections into a command syntax window and then refine the criteria used to define duplicate cases.

Example

In the data file *duplicates.sav*, each case is identified by two ID variables: *ID_house*, which identifies each household, and *ID_person*, which identifies each person within the household. If multiple cases have the same value for both variables, then they represent the same case. In this example, that is not necessarily a coding error, since the same person may have been interviewed on more than one occasion.

   The interview date is recorded in the variable *int_date*, and for cases that match on both ID variables, we want to ignore all but the most recent interview.

```
* duplicates_filter.sps.
GET FILE='/examples/data/duplicates.sav'.
SORT CASES BY ID_house(A) ID_person(A) int_date(A) .
MATCH FILES  /FILE = *
  /BY ID_house ID_person /LAST = MostRecent .
FILTER BY MostRecent .
EXECUTE.
```

■ `SORT CASES` sorts the data file by the two ID variables and the interview date. The end result is that all cases with the same household ID are grouped together, and within each household, cases with the same person ID are grouped together. Those cases are sorted by ascending interview date; for any duplicates, the last case will be the most recent interview date.

■ Although `MATCH FILES` is typically used to merge two or more data files, you can use `FILE = *` to match the active dataset with itself. In this case, that is useful not because we want to merge data files but because we want another feature of the command—the ability to identify the `LAST` case for each value of the key variables specified on the `BY` subcommand.

■ `BY ID_house ID_person` defines a match as cases having the same values for those two variables. The order of the `BY` variables must match the sort order of the data file. In this example, the two variables are specified in the same order on both the `SORT CASES` and `MATCH FILES` commands.

■ `LAST = MostRecent` assigns a value of 1 for the new variable *MostRecent* to the last case in each matching group and a value of 0 to all other cases in each matching group. Since the data file is sorted by ascending interview date within the two ID variables, the most recent

interview date is the last case in each matching group. If there is only one case in a group, then it is also considered the last case and is assigned a value of 1 for the new variable *MostRecent*.

■ `FILTER BY MostRecent` filters out any cases with a value of 0 for *MostRecent*, which means that all but the case with the most recent interview date in each duplicate group will be excluded from reports and analyses. Filtered-out cases are indicated with a slash through the row number in Data View in the Data Editor.

Figure 7-3
Filtered duplicate cases in Data View

Example

You may not want to automatically exclude duplicates from reports; you may want to examine them before deciding how to treat them. You could simply omit the `FILTER` command at the end of the previous example and look at each group of duplicates in the Data Editor, but if there are many variables and you are interested in examining only the values of a few key variables, that might not be the optimal approach.

This example counts the number of duplicates in each group and then displays a report of a selected set of variables for all duplicate cases, sorted in descending order of the duplicate count, so the cases with the largest number of duplicates are displayed first.

```
*duplicates_count.sps.
GET FILE='/examples/data/duplicates.sav'.
AGGREGATE OUTFILE = * MODE = ADDVARIABLES
  /BREAK = ID_house ID_person
  /DuplicateCount = N.
SORT CASES BY DuplicateCount (D).
COMPUTE filtervar=(DuplicateCount > 1).
FILTER BY filtervar.
SUMMARIZE
  /TABLES=ID_house ID_person int_date DuplicateCount
  /FORMAT=LIST NOCASENUM TOTAL
  /TITLE='Duplicate Report'
  /CELLS=COUNT.
```

■ The `AGGREGATE` command is used to create a new variable that represents the number of cases for each pair of ID values.

- ■ `OUTFILE = * MODE = ADDVARIABLES` writes the aggregated results as new variables in the active dataset. (This is the default behavior.)

- ■ The `BREAK` subcommand aggregates cases with matching values for the two ID variables. In this example, that simply means that each case with the same two values for the two ID variables will have the same values for any new variables based on aggregated results.

- ■ `DuplicateCount = N` creates a new variable that represents the number of cases for each pair of ID values. For example, the *DuplicateCount* value of 3 is assigned to the three cases in the active dataset with the values of 102 and 1 for *ID_house* and *ID_person*, respectively.

- ■ The `SORT CASES` command sorts the data file in descending order of the values of *DuplicateCount*, so cases with the largest numbers of duplicates will be displayed first in the subsequent report.

- ■ `COMPUTE filtervar=(DuplicateCount > 1)` creates a new variable with a value of 1 for any cases with a *DuplicateCount* value greater than 1 and a value of 0 for all other cases. So all cases that are considered duplicates have a value of 1 for *filtervar*, and all unique cases have a value of 0.

- ■ `FILTER BY filtervar` selects all cases with a value of 1 for *filtervar* and filters out all other cases. So subsequent procedures will include only duplicate cases.

- ■ The `SUMMARIZE` command produces a report of the two ID variables, the interview date, and the number of duplicates in each group for all duplicate cases. It also displays the total number of duplicates. The cases are displayed in the current file order, which is in descending order of the duplicate count value.

Figure 7-4
Summary report of duplicate cases

Duplicate Report

	Household ID	Person ID	Interview date	DuplicateCount
1	102	1	07/07/2002	3
2	102	1	10/12/2002	3
3	102	1	01/15/2003	3
4	101	4	12/31/2002	2
5	101	4	10/29/2003	2
Total N	5	5	5	5

Data Preparation Option

The Data Preparation option provides two validation procedures:

- ■ `VALIDATEDATA` provides the ability to define and apply validation rules that identify invalid data values. You can create rules that flag out-of-range values, missing values, or blank values. You can also save variables that record individual rule violations and the total number of rule violations per case.

- ■ `DETECTANOMALY` finds unusual observations that could adversely affect predictive models. The procedure is designed to quickly detect unusual cases for data-auditing purposes in the exploratory data analysis step, prior to any inferential data analysis. This algorithm is designed for generic anomaly detection; that is, the definition of an anomalous case is not specific to any particular application, such as detection of unusual payment patterns in the healthcare industry or detection of money laundering in the finance industry, in which the definition of an anomaly can be well-defined.

Example

This example illustrates how you can use the Data Preparation procedures to perform a simple, initial evaluation of any dataset, without defining any special rules for validating the data. The procedures provide many features not covered here (including the ability to define and apply custom rules).

```
*data_validation.sps
***create some sample data***.
INPUT PROGRAM.
SET SEED 123456789.
LOOP #i=1 to 1000.
- COMPUTE notCategorical=RV.NORMAL(200,40).
- DO IF UNIFORM(100) < 99.8.
-   COMPUTE mostlyConstant=1.
-   COMPUTE mostlyNormal=RV.NORMAL(50,10).
- ELSE.
-   COMPUTE mostlyConstant=2.
-   COMPUTE mostlyNormal=500.
- END IF.
- END CASE.
END LOOP.
END FILE.
END INPUT PROGRAM.
VARIABLE LEVEL notCategorical mostlyConstant(nominal).
****Here's the real job****.
VALIDATEDATA VARIABLES=ALL.
DETECTANOMALY.
```

■ The input program creates some sample data with a few notable anomalies, including a variable that is normally distributed, with the exception of a small proportion of cases with a value far greater than all of the other cases, and a variable where almost all of the cases have the same value. Additionally, the scale variable *notCategorical* has been assigned the nominal measurement level.

■ VALIDATEDATA performs the default data validation routines, including checking for categorical (nominal, ordinal) variables where more than 95% of the cases have the same value or more than 90% of the cases have unique values.

■ DETECTANOMALY performs the default anomaly detection on all variables in the dataset.

Figure 7-5
Results from VALIDATEDATA

Variable Checks

| Categorical | Cases Constant > 95.0% | mostlyConstant |
| | Categories Containing One Case > 90.0% | notCategorical |

Each variable is reported with every check it fails.

Figure 7-6
Results from DETECTANOMALY

Anomaly Case Index List

Case	Anomaly Index
81	16.296
483	16.296
871	16.296

Anomaly Case Reason List

Case	Reason Variable	Variable Impact	Variable Value	Variable Norm
81	mostlyNormal	.900	500.00	51.89
483	mostlyNormal	.900	500.00	51.89
871	mostlyNormal	.900	500.00	51.89

■ The default VALIDATEDATA evaluation detects and reports that more than 95% of cases for the categorical variable *mostlyConstant* have the same value and more than 90% of cases for the categorical variable *notCategorical* have unique values. The default evaluation, however, found nothing unusual to report in the scale variable *mostlyNormal*.

■ The default DETECTANOMALY analysis reports any case with an anomaly index of 2 or more. In this example, three cases have an anomaly index of over 16. The *Anomaly Case Reason List* table reveals that these three cases have a value of 500 for the variable *mostlyNormal*, while the mean value for that variable is only 52.

Conditional Processing, Looping, and Repeating

As with other programming languages, IBM® SPSS® Statistics contains standard programming structures that can be used to do many things. These include the ability to:

- Perform actions only if some condition is true (if/then/else processing)
- Repeat actions
- Create an array of elements
- Use loop structures

Indenting Commands in Programming Structures

Indenting commands nested within programming structures is a fairly common convention that makes code easier to read and debug. For compatibility with batch production mode, however, each IBM® SPSS® Statistics command should begin in the first column of a new line. You can indent nested commands by inserting a plus (+) or minus (−) sign or a period (.) in the first column of each indented command, as in:

```
DO REPEAT tempvar = var1, var2, var3.
+  COMPUTE tempvar = tempvar/10.
+  DO IF (tempvar >= 100).   /*Then divide by 10 again.
+     COMPUTE tempvar = tempvar/10.
+  END IF.
END REPEAT.
```

Conditional Processing

Conditional processing with IBM® SPSS® Statistics commands is performed on a **casewise** basis—each case is evaluated to determine if the condition is met. This is well suited for tasks such as setting the value of a new variable or creating a subset of cases based on the value(s) of one or more existing variables.

Note: Conditional processing or flow control on a **jobwise** basis—such as running different procedures for different variables based on data type or level of measurement or determining which procedure to run next based on the results of the last procedure—typically requires the type of functionality available only with the programmability features discussed in the second part of this book.

Conditional Transformations

There are a variety of methods for performing conditional transformations, including:

- Logical variables

- One or more IF commands, each defining a condition and an outcome
- If/then/else logic in a DO IF structure

Example

```
*if_doif1.sps.
DATA LIST FREE /var1.
BEGIN DATA
1 2 3 4
END DATA.
COMPUTE newvar1=(var1<3).
IF (var1<3) newvar2=1.
IF (var1>=3) newvar2=0.
DO IF var1<3.
- COMPUTE newvar3=1.
ELSE.
- COMPUTE newvar3=0.
END IF.
EXECUTE.
```

- The logical variable *newvar1* will have a value of 1 if the condition is true, a value of 0 if it is false, and system-missing if the condition cannot be evaluated due to missing data. While it requires only one simple command, logical variables are limited to numeric values of 0, 1, and system-missing.
- The two IF commands return the same result as the single COMPUTE command that generated the logical variable. Unlike the logical variable, however, the result of an IF command can be virtually any numeric or string value, and you are not limited to two outcome results. Each IF command defines a single conditional outcome, but there is no limit to the number of IF commands you can specify.
- The DO IF structure also returns the same result, and like the IF commands, there is no limit on the value of the outcome or the number of possible outcomes.

Example

As long as all the conditions are mutually exclusive, the choice between IF and DO IF may often be a matter of preference, but what if the conditions are not mutually exclusive?

```
*if_doif2.sps
DATA LIST FREE /var1 var2.
BEGIN DATA
1 1
2 1
END DATA.
IF (var1=1) newvar1=1.
IF (var2=1) newvar1=2.
DO IF var1=1.
- COMPUTE newvar2=1.
ELSE IF var2=1.
- COMPUTE newvar2=2.
END IF.
EXECUTE.
```

- The two IF statements are not mutually exclusive, since it's possible for a case to have a value of 1 for both *var1* and *var2*. The first IF statement will assign a value of 1 to *newvar1* for the first case, and then the second IF statement will change the value of *newvar1* to 2 for the same case. In IF processing, the general rule is "the last one wins."

- The DO IF structure evaluates the same two conditions, with different results. The first case meets the first condition and the value of *newvar2* is set to 1 for that case. At this point, the DO IF structure moves on to the next case, because once a condition is met, no further conditions are evaluated for that case. So the value of *newvar2* remains 1 for the first case, even though the second condition (which would set the value to 2) is also true.

Missing Values in DO IF Structures

Missing values can affect the results from DO IF structures because if the expression evaluates to missing, then control passes immediately to the END IF command at that point. To avoid this type of problem, you should attempt to deal with missing values first in the DO IF structure before evaluating any other conditions.

```
* doif_elseif_missing.sps.

*create sample data with missing data.
DATA LIST FREE (",") /a.
BEGIN DATA
1, , 1 , ,
END DATA.

COMPUTE b=a.

* The following does NOT work since the second condition is never evaluated.
DO IF a=1.
- COMPUTE a1=1.
ELSE IF MISSING(a).
- COMPUTE a1=2.
END IF.

* On the other hand the following works.
DO IF MISSING(b).
- COMPUTE b1=2.
ELSE IF b=1.
- COMPUTE b1=1.
END IF.
EXECUTE.
```

- The first DO IF will never yield a value of 2 for *a1*, because if *a* is missing, then DO IF a=1 evaluates as missing, and control passes immediately to END IF. So *a1* will either be 1 or missing.

- In the second DO IF, however, we take care of the missing condition first; so if the value of *b* is missing, DO IF MISSING(b) evaluates as *true* and *b1* is set to 2; otherwise, *b1* is set to 1.

In this example, DO IF MISSING(b) will always evaluate as either *true* or *false*, never as missing, thereby eliminating the situation in which the first condition might evaluate as missing and pass control to END IF without evaluating the other condition(s).

Figure 8-1

DO IF results with missing values displayed in Data Editor

Conditional Case Selection

If you want to select a subset of cases for analysis, you can either filter or delete the unselected cases.

Example

```
*filter_select_if.sps.
DATA LIST FREE /var1.
BEGIN DATA
1 2 3 2 3
END DATA.
DATASET NAME filter.
DATASET COPY temporary.
DATASET COPY select_if.
*compute and apply a filter variable.
COMPUTE filterVar=(var1 ~=3).
FILTER By filterVar.
FREQUENCIES VARIABLES=var1.
*delete unselected cases from active dataset.
DATASET ACTIVATE select_if.
SELECT IF (var1~=3).
FREQUENCIES VARIABLES=var1.
*temporarily exclude unselected cases.
DATASET ACTIVATE temporary.
TEMPORARY.
SELECT IF (var1~=3).
FREQUENCIES VARIABLES=var1.
FREQUENCIES VARIABLES=var1.
```

■ The COMPUTE command creates a new variable, *filterVar*. If *var1* is not equal to 3, *filterVar* is set to 1; if *var1* is 3, *filterVar* is set to 0.

■ The FILTER command filters cases based on the value of *filterVar*. Any case with a value other than 1 for *filterVar* is filtered out and is not included in subsequent statistical and charting procedures. The cases remain in the dataset and can be reactivated by changing the filter condition or turning filtering off (FILTER OFF). Filtered cases are marked in the Data Editor with a diagonal line through the row number.

- `SELECT IF` deletes unselected cases from the active dataset, and those cases are no longer available in that dataset.

- The combination of `TEMPORARY` and `SELECT IF` temporarily deletes the unselected cases. `SELECT IF` is a transformation, and `TEMPORARY` signals the beginning of temporary transformations that are in effect only for the next command that reads the data. For the first `FREQUENCIES` command following these commands, cases with a value of 3 for *var1* are excluded. For the second `FREQUENCIES` command, however, cases with a value of 3 are now included again.

Simplifying Repetitive Tasks with DO REPEAT

A `DO REPEAT` structure allows you to repeat the same group of transformations multiple times, thereby reducing the number of commands that you need to write. The basic format of the command is:

```
DO REPEAT stand-in variable = variable or value list
    /optional additional stand-in variable(s) …
transformation commands
END REPEAT PRINT.
```

- The transformation commands inside the `DO REPEAT` structure are repeated for each variable or value assigned to the stand-in variable.

- Multiple stand-in variables and values can be specified in the same `DO REPEAT` structure by preceding each additional specification with a forward slash.

- The optional `PRINT` keyword after the `END REPEAT` command is useful when debugging command syntax, since it displays the actual commands generated by the `DO REPEAT` structure.

- Note that when a stand-in variable is set equal to a list of variables, the variables do not have to be consecutive in the data file. So `DO REPEAT` may be more useful than `VECTOR` in some circumstances. For more information, see the topic "Vectors" on p. 119.

Example

This example sets two variables to the same value.

```
* do_repeat1.sps.

***create some sample data***.
DATA LIST LIST /var1 var3 id var2.
BEGIN DATA
3 3 3 3
2 2 2 2
END DATA.
***real job starts here***.
DO REPEAT v=var1 var2.
- COMPUTE v=99.
END REPEAT.
EXECUTE.
```

Figure 8-2
Two variables set to the same constant value

- The two variables assigned to the stand-in variable *v* are assigned the value 99.
- If the variables don't already exist, they are created.

Example

You could also assign different values to each variable by using two stand-in variables: one that specifies the variables and one that specifies the corresponding values.

```
* do_repeat2.sps.
***create some sample data***.
DATA LIST LIST /var1 var3 id var2.
BEGIN DATA
3 3 3 3
2 2 2 2
END DATA.
***real job starts here***.
DO REPEAT v=var1 TO var2 /val=1 3 5 7.
- COMPUTE v=val.
END REPEAT PRINT.
EXECUTE.
```

Figure 8-3
Different value assigned to each variable

- The COMPUTE command inside the structure is repeated four times, and each value of the stand-in variable *v* is associated with the corresponding value of the variable *val*.
- The PRINT keyword displays the generated commands in the log item in the Viewer.

Figure 8-4
Commands generated by DO REPEAT displayed in the log
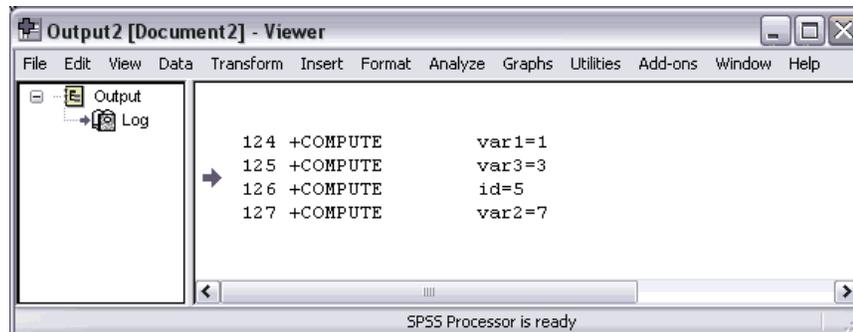
ALL Keyword and Error Handling

You can use the keyword ALL to set the stand-in variable to all variables in the active dataset; however, since not all variables are created equal, actions that are valid for some variables may not be valid for others, resulting in errors. For example, some functions are valid only for numeric variables, and other functions are valid only for string variables.

You can suppress the display of error messages with the command SET ERRORS = NONE, which can be useful if you know your command syntax will create a certain number of harmless error conditions for which the error messages are mostly noise. This does not, however, tell the program to ignore error conditions; it merely prevents error messages from being displayed in the output. This distinction is important for command syntax run via an INCLUDE command, which will terminate on the first error encountered regardless of the setting for displaying error messages.

Vectors

Vectors are a convenient way to sequentially refer to consecutive variables in the active dataset. For example, if *age*, *sex*, and *salary* are three consecutive numeric variables in the data file, we can define a vector called *VectorVar* for those three variables. We can then refer to these three variables as *VectorVar(1)*, *VectorVar(2)*, and *VectorVar(3)*. This is often used in LOOP structures but can also be used without a LOOP.

Example

You can use the MAX function to find the highest value among a specified set of variables. But what if you also want to know which variable has that value, and if more than one variable has that value, how many variables have that value? Using VECTOR and LOOP, you can get the information you want.

```
*vectors.sps.

***create some sample data***.
DATA LIST FREE
  /FirstVar SecondVar ThirdVar FourthVar FifthVar.
BEGIN DATA
1 2 3 4 5
10 9 8 7 6
```

```
1 4 4 4 2
END DATA.

***real job starts here***.
COMPUTE MaxValue=MAX(FirstVar TO FifthVar).
COMPUTE MaxCount=0.

VECTOR VectorVar=FirstVar TO FifthVar.
LOOP #cnt=5 to 1 BY -1.
- DO IF MaxValue=VectorVar(#cnt).
-    COMPUTE MaxVar=#cnt.
-    COMPUTE MaxCount=MaxCount+1.
- END IF.
END LOOP.
EXECUTE.
```

■ For each case, the MAX function in the first COMPUTE command sets the variable *MaxValue* to the maximum value within the inclusive range of variables from *FirstVar* to *FifthVar*. In this example, that happens to be five variables.

■ The second COMPUTE command initializes the variable *MaxCount* to 0. This is the variable that will contain the count of variables with the maximum value.

■ The VECTOR command defines a vector in which *VectorVar(1) = FirstVar*, *VectorVar(2) =* the next variable in the file order, ..., *VectorVar(5) = FifthVar*.

Note: Unlike some other programming languages, vectors in IBM® SPSS® Statistics start at 1, not 0.

■ The LOOP structure defines a loop that will be repeated five times, decreasing the value of the temporary variable *#cnt* by 1 for each loop. On the first loop, *VectorVar(#cnt)* equals *VectorVar(5),* which equals *FifthVar*; on the last loop, it will equal *VectorVar(1)*, which equals *FirstVar*.

■ If the value of the current variable equals the value of *MaxValue*, then the value of *MaxVar* is set to the current loop number represented by *#cnt*, and *MaxCount* is incremented by 1.

■ The final value of *MaxVar* represents the position of the first variable in file order that contains the maximum value, and *MaxCount* is the number of variables that have that value. (LOOP #cnt = 1 TO 5 would set *MaxVar* to the position of the *last* variable with the maximum value.)

■ The vector exists only until the next EXECUTE command or procedure that reads the data.

Figure 8-5
Highest value across variables identified with VECTOR and LOOP

Creating Variables with VECTOR

You can use the short form of the VECTOR command to create multiple new variables. The short form is VECTOR followed by a variable name prefix and, in parentheses, the number of variables to create. For example,

```
VECTOR newvar(100).
```

will create 100 new variables, named *newvar1, newvar2, ..., newvar100.*

Disappearing Vectors

Vectors have a short life span; a vector lasts only until the next command that reads the data, such as a statistical procedure or the EXECUTE command. This can lead to problems under some circumstances, particularly when you are testing and debugging a command file. When you are creating and debugging long, complex command syntax jobs, it is often useful to insert EXECUTE commands at various stages to check intermediate results. Unfortunately, this kills any defined vectors that might be needed for subsequent commands, making it necessary to redefine the vector(s). However, redefining the vectors sometimes requires special consideration.

```
* vectors_lifespan.sps.

GET FILE='/examples/data/employee data.sav'.
VECTOR vec(5).
LOOP #cnt=1 TO 5.
- COMPUTE vec(#cnt)=UNIFORM(1).
END LOOP.
EXECUTE.

*Vector vec no longer exists; so this will cause an error.
LOOP #cnt=1 TO 5.
- COMPUTE vec(#cnt)=vec(#cnt)*10.
END LOOP.

*This also causes error because variables vec1 - vec5 now exist.
VECTOR vec(5).
LOOP #cnt=1 TO 5.
- COMPUTE vec(#cnt)=vec(#cnt)*10.
END LOOP.

* This  redefines vector without error.
VECTOR vec=vec1 TO vec5.
LOOP #cnt=1 TO 5.
- COMPUTE vec(#cnt)=vec(#cnt)*10.
END LOOP.
EXECUTE.
```

- The first VECTOR command uses the **short form** of the command to create five new variables as well as a vector named *vec* containing those five variable names: *vec1* to *vec5*.
- The LOOP assigns a random number to each variable of the vector.
- EXECUTE completes the process of assigning the random numbers to the new variables (transformation commands like COMPUTE aren't run until the next command that reads the data). Under normal circumstances, this may not be necessary at this point. However, you might do this when debugging a job to make sure that the correct values are assigned. At this

point, the five variables defined by the VECTOR command exist in the active dataset, but the vector that defined them is gone.

- Since the vector *vec* no longer exists, the attempt to use the vector in the subsequent LOOP will cause an error.

- Attempting to redefine the vector in the same way it was originally defined will also cause an error, since the short form will attempt to create new variables using the names of existing variables.

- VECTOR vec=vec1 TO vec5 redefines the vector to contain the same series of variable names as before without generating any errors, because this form of the command defines a vector that consists of a range of contiguous variables that already exist in the active dataset.

Loop Structures

The LOOP-END LOOP structure performs repeated transformations specified by the commands within the loop until it reaches a specified cutoff. The cutoff can be determined in a number of ways:

```
*loop1.sps.
*create sample data, 4 vars = 0.
DATA LIST FREE /var1 var2 var3 var4 var5.
BEGIN DATA
0 0 0 0 0
END DATA.
***Loops start here***.
*Loop that repeats until MXLOOPS value reached.
SET MXLOOPS=10.
LOOP.
- COMPUTE var1=var1+1.
END LOOP.
*Loop that repeats 9 times, based on indexing clause.
LOOP #I = 1 to 9.
- COMPUTE var2=var2+1.
END LOOP.
*Loop while condition not encountered.
LOOP IF (var3 < 8).
- COMPUTE var3=var3+1.
END LOOP.
*Loop until condition encountered.
LOOP.
- COMPUTE var4=var4+1.
END LOOP IF (var4 >= 7).
*Loop until BREAK condition.
LOOP.
- DO IF  (var5 < 6).
-    COMPUTE var5=var5+1.
- ELSE.
-    BREAK.
- END IF.
END LOOP.
EXECUTE.
```

- An unconditional loop with no indexing clause will repeat until it reaches the value specified on the SET MXLOOPS command. The default value is 40.

- LOOP #I = 1 to 9 specifies an indexing clause that will repeat the loop nine times, incrementing the value of *#I* by 1 for each loop. LOOP #tempvar = 1 to 10 BY 2 would repeat five times, incrementing the value of *#tempvar* by 2 for each loop.

- ◾ `LOOP IF` continues as long as the specified condition is not encountered. This corresponds to the programming concept of "do while."

- ◾ `END LOOP IF` continues until the specified condition is encountered. This corresponds to the programming concept of "do until."

- ◾ A `BREAK` command in a loop ends the loop. Since `BREAK` is unconditional, it is typically used only inside of conditional structures in the loop, such as `DO IF-END IF`.

Indexing Clauses

The indexing clause limits the number of iterations for a loop by specifying the number of times the program should execute commands within the loop structure. The indexing clause is specified on the `LOOP` command and includes an indexing variable followed by initial and terminal values.

The indexing variable can do far more than simply define the number of iterations. The current value of the indexing variable can be used in transformations and conditional statements within the loop structure. So it is often useful to define indexing clauses that:

- ◾ Use the `BY` keyword to increment the value of the indexing variable by some value other than the default of 1, as in: `LOOP #i = 1 TO 100 BY 5`.

- ◾ Define an indexing variable that decreases in value for each iteration, as in: `LOOP #j = 100 TO 1 BY -1`.

Loops that use an indexing clause are not constrained by the `MXLOOPS` setting. An indexing clause that defines 1,000 iterations will be iterated 1,000 times even if the `MXLOOPS` setting is only 40.

The loop structure described in "Vectors" on p. 119 uses an indexing variable that decreases for each iteration. The loop structure described in "Using XSAVE in a Loop to Build a Data File" on p. 126 has an indexing clause that uses an arithmetic function to define the ending value of the index. Both examples use the current value of the indexing variable in transformations in the loop structure.

Nested Loops

You can nest loops inside of other loops. A nested loop is run for every iteration of the parent loop. For example, a parent loop that defines 5 iterations and a nested loop that defines 10 iterations will result in a total of 50 iterations for the nested loop (10 times for each iteration of the parent loop).

Example

Many statistical tests rely on assumptions of normal distributions and the **Central Limit Theorem**, which basically states that even if the distribution of the population is not normal, repeated random samples of a sufficiently large size will yield a distribution of sample means that is normal.

We can use an input program and nested loops to demonstrate the validity of the Central Limit Theorem. For this example, we'll assume that a sample size of 100 is "sufficiently large."

```
*loop_nested.sps.
NEW FILE.
SET RNG=MC SEED=987987987.
INPUT PROGRAM.
- VECTOR UniformVar(100).
- *parent loop creates cases.
```

```
- LOOP #I=1 TO 100.
- *nested loop creates values for each variable in each case.
- LOOP #J=1 to 100.
-  COMPUTE UniformVar(#J)=RV.UNIFORM(1,1000).
- END LOOP.
- END CASE.
- END LOOP.
- END FILE.
END INPUT PROGRAM.
COMPUTE UniformMean=mean(UniformVar1 TO UniformVar100).
COMPUTE NormalVar=RV.NORMAL(500,100).
FREQUENCIES
  VARIABLES=NormalVar UniformVar1 UniformMean
  /FORMAT=NOTABLE
  /HISTOGRAM  NORMAL
  /ORDER = ANALYSIS.
```

- The first two commands simply create a new, empty active dataset and set the random number seed to consistently duplicate the same results.

- `INPUT PROGRAM-END INPUT PROGRAM` is used to generate cases in the data file.

- The `VECTOR` command creates a vector called *UniformVar*, and it also creates 100 variables, named *UniformVar1, UniformVar2, ..., UniformVar100*.

- The outer `LOOP` creates 100 cases via the `END CASE` command, which creates a new case for each iteration of the loop. `END CASE` is part of the input program and can be used only within an `INPUT PROGRAM-END INPUT PROGRAM` structure.

- For each case created by the outer loop, the nested `LOOP` creates values for the 100 variables. For each iteration, the value of *#J* increments by one, setting *UniformVar(#J)* to *UniformVar(1)*, then *UniformVar(2)*, and so forth, which in turn stands for *UniformVar1, UniformVar2*, and so forth.

- The `RV.UNIFORM` function assigns each variable a random value based on a uniform distribution. This is repeated for all 100 cases, resulting in 100 cases and 100 variables, all containing random values based on a uniform distribution. So the distribution of values within each variable and across variables within each case is non-normal.

- The `MEAN` function creates a variable that represents the mean value across all variables for each case. This is essentially equivalent to the distribution of sample means for 100 random samples, each containing 100 cases.

- For comparison purposes, we use the `RV.NORMAL` function to create a variable with a normal distribution.

- Finally, we create histograms to compare the distributions of the variable based on a normal distribution (*NormalVar*), one of the variables based on a uniform distribution (*UniformVar1*), and the variable that represents the distribution of sample means (*UniformMean*).

Figure 8-6
Demonstrating the Central Limit Theorem with nested loops
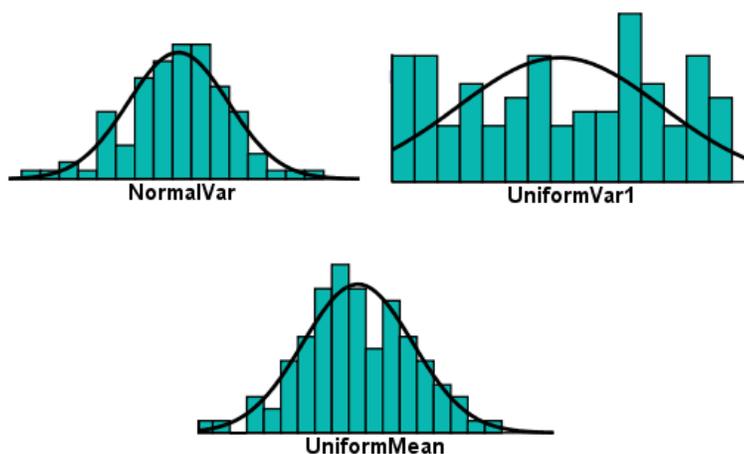
As you can see from the histograms, the distribution of sample means represented by *UniformMean* is approximately normal, despite the fact that it was generated from samples with uniform distributions similar to *UniformVar1*.

Conditional Loops

You can define conditional loop processing with LOOP IF or END LOOP IF. The main difference between the two is that, given equivalent conditions, END LOOP IF will produce one more iteration of the loop than LOOP IF.

Example

```
*loop_if1.sps.
DATA LIST FREE /X.
BEGIN DATA
1 2 3 4 5
END DATA.
SET MXLOOPS=10.
COMPUTE Y=0.
LOOP IF (X~=3).
- COMPUTE Y=Y+1.
END LOOP.
COMPUTE Z=0.
LOOP.
- COMPUTE Z=Z+1.
END LOOP IF (X=3).
EXECUTE.
```

- LOOP IF (X~=3) does nothing when *X* is 3, so the value of *Y* is not incremented and remains 0 for that case.

- END LOOP IF (X=3) will iterate once when *X* is 3, incrementing *Z* by 1, yielding a value of 1.

- For all other cases, the loop is iterated the number of times specified on SET MXLOOPS, yielding a value of 10 for both *Y* and *Z*.

Using XSAVE in a Loop to Build a Data File

You can use XSAVE in a loop structure to build a data file, writing one case at a time to the new data file.

Example

This example constructs a data file of casewise data from aggregated data. The aggregated data file comes from a table that reports the number of males and females by age. Since IBM® SPSS® Statistics works best with raw (casewise) data, we need to disaggregate the data, creating one case for each person and a new variable that indicates gender for each case.

In addition to using XSAVE to build the new data file, this example also uses a function in the indexing clause to define the ending index value.

```
*loop_xsave.sps.
DATA LIST FREE
 /Age Female Male.
BEGIN DATA
20 2 2
21 0 0
22 1 4
23 3 0
24 0 1
END DATA.
LOOP #cnt=1 to SUM(Female, Male).
- COMPUTE Gender = (#cnt > Female).
- XSAVE OUTFILE="/temp/tempdata.sav"
    /KEEP Age Gender.
END LOOP.
EXECUTE.
GET FILE='/temp/tempdata.sav'.
COMPUTE IdVar=$CASENUM.
FORMATS Age Gender (F2.0) IdVar(N3).
EXECUTE.
```

- DATA LIST is used to read the aggregated, tabulated data. For example, the first case (record) represents two females and two males aged 20.

- The SUM function in the LOOP indexing clause defines the number of loop iterations for each case. For example, for the first case, the function returns a value of 4, so the loop will iterate four times.

- On the first two iterations, the value of the indexing variable *#cnt* is not greater than the number of females. So the new variable *Gender* takes a value of 0 for each of those iterations, and the values 20 and 0 (for *Age* and *Gender*) are saved to the new data file for the first two cases.

- During the subsequent two iterations, the comparison #cnt > Female is true, returning a value of 1, and the next two variables are saved to the new data file with the values of 20 and 1.

- This process is repeated for each case in the aggregated data file. The second case results in no loop iterations and consequently no cases in the new data file, the third case produces five new cases, and so on.

■ Since XSAVE is a transformation, we need an EXECUTE command after the loop ends to finish the process of saving the new data file.

■ The FORMATS command specifies a format of N3 for the ID variable, displaying leading zeros for one- and two-digit values. GET FILE opens the data file that we created, and the subsequent COMPUTE command creates a sequential ID variable based on the system variable $CASENUM, which is the current row number in the data file.

Figure 8-7
Tabular source data and new disaggregated data file

Age	Female	Male
20.00	2.00	2.00
21.00	.00	.00
22.00	1.00	4.00
23.00	3.00	.00
24.00	.00	1.00

Age	Gender	IdVar
20	0	001
20	0	002
20	1	003
20	1	004
22	0	005
22	1	006
22	1	007
22	1	008
22	1	009
23	0	010
23	0	011
23	0	012
24	1	013

Calculations Affected by Low Default MXLOOPS Setting

A LOOP with an end point defined by a logical condition (for example, END LOOP IF varx > 100) will loop until the defined end condition is reached or until the number of loops specified on SET MXLOOPS is reached, whichever comes first. The default value of MXLOOPS is only 40, which may produce undesirable results or errors that can be hard to locate for looping structures that require a larger number of loops to function properly.

Example

This example generates a data file with 1,000 cases, where each case contains the number of random numbers—uniformly distributed between 0 and 1—that have to be drawn to obtain a number less than 0.001. Under normal circumstance, you would expect the mean value to be around 1,000 (randomly drawing numbers between 0 and 1 will result in a value of less than 0.001 roughly once every thousand numbers), but the low default value of MXLOOPS would give you misleading results.

```
* set_mxloops.sps.

SET MXLOOPS=40.     /* Default value. Change to 10000 and compare.
SET SEED=02051242.
INPUT PROGRAM.
LOOP cnt=1 TO 1000. /*LOOP with indexing clause not affected by MXLOOPS.
- COMPUTE n=0.
- LOOP.
-    COMPUTE n=n+1.
- END LOOP IF UNIFORM(1)<.001. /*Loops limited by MXLOOPS setting.
- END CASE.
END LOOP.
END FILE.
```

```
END INPUT PROGRAM.

DESCRIPTIVES VARIABLES=n
  /STATISTICS=MEAN MIN MAX .
```

- All of the commands are syntactically valid and produce no warnings or error messages.

- `SET MXLOOPS=40` simply sets the maximum number of loops to the default value.

- The seed is set so that the same result occurs each time the commands are run.

- The outer `LOOP` generates 1,000 cases. Since it uses an indexing clause (`cnt=1 TO 1000`), it is unconstrained by the `MXLOOPS` setting.

- The nested `LOOP` is *supposed* to iterate until it produces a random value of less than 0.001.

- Each case includes the case number (*cnt*) and *n*, where *n* is the number of times we had to draw a random number before getting a number less than 0.001. There is 1 chance in 1,000 of getting such a number.

- The `DESCRIPTIVES` command shows that the mean value of *n* is only 39.2—far below the expected mean of close to 1,000. Looking at the maximum value gives you a hint as to why the mean is so low. The maximum is only 40, which is remarkably close to the mean of 39.2; and if you look at the values in the Data Editor, you can see that nearly all of the values of *n* are 40, because the `MXLOOPS` limit of 40 was almost always reached before a random uniform value of 0.001 was obtained.

- If you change the `MXLOOPS` setting to 10,000 (`SET MXLOOPS=10000`), however, you get very different results. The mean is now 980.9, fairly close to the expected mean of 1,000.

Figure 8-8
Different results with different MXLOOPS settings

MXLOOPS = 40

	N	Minimum	Maximum	Mean
n	1000	1.00	40.00	39.2100
Valid N (listwise)	1000			

cnt	n
1.00	40.00
2.00	40.00
3.00	40.00
4.00	40.00
5.00	40.00
6.00	40.00
7.00	40.00
8.00	29.00
9.00	40.00
10.00	40.00

MXLOOPS = 10000

	N	Minimum	Maximum	Mean
n	1000	2.00	8223.00	980.9090
Valid N (listwise)	1000			

cnt	n
1.00	309.00
2.00	2261.00
3.00	800.00
4.00	2595.00
5.00	1850.00
6.00	281.00
7.00	244.00
8.00	1064.00
9.00	386.00
10.00	1718.00

Exporting Data and Results

You can export and save both data and results in a variety of formats for use by other applications, including:

- Export data in formats that can be read by other data analysis applications
- Write data to databases
- Export output in Word, Excel, PDF, HTML, and text format
- Export output in IBM® SPSS® Statistics format and then use it as input for subsequent analysis
- Read SPSS Statistics format data files into other applications using the SPSS Statistics data file driver.

Exporting Data to Other Applications and Formats

You can save the contents of the active dataset in a variety of formats, including SAS, Stata, and Excel. You can also write data to a database.

Saving Data in SAS Format

With the `SAVE TRANSLATE` command, you can save data as SAS v6, SAS v7, and SAS transport files. A SAS transport file is a sequential file written in SAS transport format and can be read by SAS with the XPORT engine and `PROC COPY` or the `DATA` step.

- Certain characters that are allowed in IBM® SPSS® Statistics variable names are not valid in SAS, such as @, #, and $. These illegal characters are replaced with an underscore when the data are exported.
- Variable labels containing more than 40 characters are truncated when exported to a SAS v6 file.
- Where they exist, SPSS Statistics variable labels are mapped to the SAS variable labels. If no variable label exists in the SPSS Statistics data, the variable name is mapped to the SAS variable label.
- SAS allows only one value for missing, whereas SPSS Statistics allows the definition of numerous missing values. As a result, all missing values in SPSS Statistics are mapped to a single missing value in the SAS file.

Example

```
*save_as_SAS.sps.
GET FILE='/examples/data/employee data.sav'.
SAVE TRANSLATE OUTFILE='/examples/data/sas7datafile.sas7bdat'
  /TYPE=SAS  /VERSION=7  /PLATFORM=WINDOWS
    /VALFILE='/examples/data/sas7datafile_labels.sas' .
```

- The active data file will be saved as a SAS v7 data file.

- `PLATFORM=WINDOWS` creates a data file that can be read by SAS running on Windows operating systems. For UNIX operating systems, use `PLATFORM=UNIX`. For platform-independent data files, use `VERSION=X` to create a SAS transport file.

- The `VALFILE` subcommand saves defined value labels in a SAS syntax file. This file contains `proc format` and `proc datasets` commands that can be run in SAS to create a SAS format catalog file.

For more information, see the `SAVE TRANSLATE` command in the *Command Syntax Reference*.

Saving Data in Stata Format

To save data in Stata format, use the `SAVE TRANSLATE` command with `/TYPE=STATA`.

Example

```
*save_as_Stata.sps.
GET FILE='/examples/data/employee data.sav'.
SAVE TRANSLATE
  OUTFILE='/examples/data/statadata.dta'
  /TYPE=STATA
  /VERSION=8
  /EDITION=SE.
```

- Data can be written in Stata 5–8 format and in both Intercooled and SE format (versions 7 and 8 only).

- Data files that are saved in Stata 5 format can be read by Stata 4.

- The first 80 bytes of variable labels are saved as Stata variable labels.

- For numeric variables, the first 80 bytes of value labels are saved as Stata value labels. Value labels are dropped for string variables, non-integer numeric values, and numeric values greater than an absolute value of 2,147,483,647.

- For versions 7 and 8, the first 32 bytes of variable names in case-sensitive form are saved as Stata variable names. For earlier versions, the first eight bytes of variable names are saved as Stata variable names. Any characters other than letters, numbers, and underscores are converted to underscores.

- IBM® SPSS® Statistics variable names that contain multibyte characters (for example, Japanese or Chinese characters) are converted to variable names of the general form *Vnnn*, where *nnn* is an integer value.

- For versions 5–6 and Intercooled versions 7–8, the first 80 bytes of string values are saved. For Stata SE 7–8, the first 244 bytes of string values are saved.

- For versions 5–6 and Intercooled versions 7–8, only the first 2,047 variables are saved. For Stata SE 7–8, only the first 32,767 variables are saved.

SPSS Statistics Variable Type	Stata Variable Type	Stata Data Format
Numeric	Numeric	g
Comma	Numeric	g
Dot	Numeric	g

SPSS Statistics Variable Type	Stata Variable Type	Stata Data Format
Scientific Notation	Numeric	g
Date*, Datetime	Numeric	D_m_Y
Time, DTime	Numeric	g (number of seconds)
Wkday	Numeric	g (1–7)
Month	Numeric	g (1–12)
Dollar	Numeric	g
Custom Currency	Numeric	g
String	String	s

*Date, Adate, Edate, SDate, Jdate, Qyr, Moyr, Wkyr

Saving Data in Excel Format

To save data in Excel format, use the SAVE TRANSLATE command with /TYPE=XLS.

Example

```
*save_as_excel.sps.
GET FILE='/examples/data/employee data.sav'.
SAVE TRANSLATE OUTFILE='/examples/data/exceldata.xls'
  /TYPE=XLS /VERSION=8
  /FIELDNAMES
  /CELLS=VALUES .
```

■ VERSION=8 saves the data file in Excel 97–2000 format.

■ FIELDNAMES includes the variable names as the first row of the Excel file.

■ CELLS=VALUES saves the actual data values. If you want to save descriptive value labels instead, use CELLS=LABELS.

Writing Data Back to a Database

SAVE TRANSLATE can also write data back to an existing database. You can create new database tables or replace or modify existing ones. As with reading database tables, writing back to a database uses ODBC, so you need to have the necessary ODBC database drivers installed.

The command syntax for writing back to a database is fairly simple, but just like reading data from a database, you need the somewhat cryptic CONNECT string. The easiest way to get the CONNECT string is to use the Export to Database wizard (File menu in the Data Editor window, Export to Database), and then paste the generated command syntax at the last step of the wizard.

For more information on ODBC drivers and CONNECT strings, see "Getting Data from Databases" on p. 19 in Chapter 3.

Example: Create a New Database Table

This example reads a table from an Access database, creates a subset of cases and variables, and then writes a new table to the database containing that subset of data.

```
*write_to_access.sps.
GET DATA /TYPE=ODBC /CONNECT=
 'DSN=MS Access Database;DBQ=C:\examples\data\dm_demo.mdb;'+
 'DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
 /SQL = 'SELECT * FROM CombinedTable'.
EXECUTE.
DELETE VARIABLES Income TO Response.
N OF CASES 50.
SAVE TRANSLATE
  /TYPE=ODBC /CONNECT=
    'DSN=MS Access Database;DBQ=C:\examples\data\dm_demo.mdb;'+
    'DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
  /TABLE='CombinedSubset'
  /REPLACE
  /UNSELECTED=RETAIN
  /MAP.
```

■ The CONNECT string in the SAVE TRANSLATE command is exactly the same as the one used in the GET DATA command, and that CONNECT string was obtained by pasting command syntax from the Database Wizard. TYPE=ODBC indicates that the data will be saved in a database. The database must already exist; you cannot use SAVE TRANSLATE to create a database.

■ The TABLE subcommand specifies the name of the database table. If the table does not already exist in the database, it will be added to the database.

■ If a table with the name specified on the TABLE subcommand already exists, the REPLACE subcommand specifies that this table should be overwritten.

■ You can use APPEND instead of REPLACE to append data to an existing table, but there must be an exact match between variable and field names and corresponding data types. The table can contain more fields than variables being written to the table, but every variable must have a matching field in the database table.

■ UNSELECTED=RETAIN specifies that any filtered, but not deleted, cases should be included in the table. This is the default. To exclude filtered cases, use UNSELECTED=DELETE.

■ The MAP subcommand provides a summary of the data written to the database. In this example, we deleted all but the first three variables and first 50 cases before writing back to the database, and the output displayed by the MAP subcommand indicates that three variables and 50 cases were written to the database.

```
Data written to CombinedSubset.
3 variables and 50 cases written.
Variable: ID                    Type: Number   Width: 11   Dec: 0
Variable: AGE                   Type: Number   Width:  8   Dec: 2
Variable: MARITALSTATUS         Type: Number   Width:  8   Dec: 2
```

Example: Append New Columns to a Database Table

The SQL subcommand provides the ability to issue any SQL directives that are needed in the target database. For example, the APPEND subcommand only appends rows to an existing table. If you want to append columns to an existing table, you can do so using SQL directives with the SQL subcommand.

```
*append_to_table.sps.
GET DATA /TYPE=ODBC /CONNECT=
 'DSN=MS Access Database;DBQ=C:\examples\data\dm_demo.mdb;'+
 'DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
```

```
  /SQL = 'SELECT * FROM CombinedTable'.
CACHE.
AUTORECODE VARIABLES=income
  /INTO income_rank
  /DESCENDING.
SAVE TRANSLATE /TYPE=ODBC
 /CONNECT=
 'DSN=MS Access Database;DBQ=C:\examples\data\dm_demo.mdb;'
 'DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
 /TABLE = 'NewColumn'
 /KEEP ID income_rank
 /REPLACE
 /SQL='ALTER TABLE CombinedTable ADD COLUMN income_rank REAL'
 /SQL='UPDATE CombinedTable INNER JOIN NewColumn ON ' +
 'CombinedTable.ID=NewColumn.ID SET ' +
 'CombinedTable.income_rank=NewColumn.income_rank'.
```

- The TABLE, KEEP, and REPLACE subcommands create or replace a table named *NewColumn* that contains two variables: a key variable (*ID*) and a calculated variable (*income_rank*).

- The first SQL subcommand, specified on a single line, adds a column to an existing table that will contain values of the computed variable *income_rank*. At this point, all we have done is create an empty column in the existing database table, and the fact that both database tables and the active dataset use the same name for that column is merely a convenience for simplicity and clarity.

- The second SQL subcommand, specified on multiple lines with the quoted strings concatenated with plus signs, adds the *income_rank* values from the new table to the existing table, matching rows (cases) based on the value of the key variable *ID*.

The end result is that an existing table is modified to include a new column containing the values of the computed variable.

Example: Specifying Data Types and Primary Keys for a New Table

The TABLE subcommand creates a database table with default database types. This example demonstrates how to create (or replace) a table with specific data types and primary keys.

```
*write_db_key.sps
DATA LIST LIST /
  ID (F3) numVar (f8.2) intVar (f3) dollarVar (dollar12.2).
BEGIN DATA
123 123.45 123 123.45
456 456.78 456 456.78
END DATA.

SAVE TRANSLATE /TYPE=ODBC
 /CONNECT='DSN=Microsoft Access;'+
 ' DBQ=c:\examples\data\dm_demo.mdb;DriverId=25;'+
 ' FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
 /SQL='CREATE TABLE NewTable(ID counter, numVar double, intVar smallint,'+
 ' dollarVar currency, primary key(ID))'
 /REPLACE
 /TABLE='tempTable'
 /SQL='INSERT INTO NewTable(ID, numVar, intVar, dollarVar) '+
 '  SELECT ID, numVar, intVar, dollarVar FROM tempTable'
 /SQL='DROP TABLE tempTable'.
```

- The first SQL subcommand creates a new table with data types explicitly defined for each field and also specifies that *ID* is the primary key. For compound primary keys, simply include all the variables that define the primary key in parentheses after primary key, as in: primary key (idVar1, idVar2). At this point, this new table contains no data.

- The TABLE subcommand creates another new table that contains variables in the active dataset with the default database data types. In this example, the original variables have variable formats of F3, F8.2, F3, and Dollar12.2 respectively, but the default database type for all four is double.

- The second SQL subcommand inserts the data from *tempTable* into *NewTable*. This does not affect the data types or primary key designation previously defined for *NewTable*, so *intVar* will have a data type of integer, *dollarVar* will have a data type of currency, and *ID* will be designated as the primary key.

- The last SQL subcommand deletes *tempTable*, since it is no longer needed.

You can use the same basic method to replace an existing table with a table that contains specified database types and primary key attributes. Just add a SQL subcommand that specifies DROP TABLE prior to the SQL subcommand that specifies CREATE TABLE.

Saving Data in Text Format

You use the SAVE TRANSLATE command to save data as tab-delimited or CSV-format text or the WRITE command to save data as fixed-width text. See the *Command Syntax Reference* for more information.

Reading IBM SPSS Statistics Data Files in Other Applications

The IBM® SPSS® Statistics data file driver allows you to read SPSS Statistics (*.sav*) data files in applications that support Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC). SPSS Statistics itself supports ODBC in the Database Wizard, providing you with the ability to leverage the Structured Query Language (SQL) when reading *.sav* data files in SPSS Statistics.

There are three flavors of the SPSS Statistics data file driver, all of which are available for Windows, UNIX, and Linux:

- **Standalone driver**. The standalone driver provides ODBC support without requiring installation of additional components. After the standalone driver is installed, you can immediately set up an ODBC data source and use it to read *.sav* files.

- **Service driver**. The service driver provides both ODBC and JDBC support. The service driver handles data requests from the service client driver, which may be installed on the same computer or on one or more remote computers. Thus you can configure one service driver that may be used by many clients. If you put your data files on the same computer on which the service driver is installed, the service driver can reduce network traffic because all the queries occur on the server. Only the resulting cases are sent to the service client. If the server has a faster processor or more RAM compared to service client machines, there may also be performance improvements.

- **Service client driver**. The service client driver provides an interface between the client application that needs to read the *.sav* data file and the service driver that handles the request for the data. Unlike the standalone driver, it supports both ODBC and JDBC. The operating system of the service client driver does not need to match the operating system of the service

driver. For example, you can install the service driver on a UNIX machine and the service client driver on a Windows machine.

Using the standalone and service client drivers is similar to connecting to a database with any other ODBC or JDBC driver. After configuring the driver, creating data sources, and connecting to the SPSS Statistics data file, you will see that the data file is represented as a collection of tables. In other words, the data file looks like a database source.

Installing the IBM SPSS Statistics Data File Driver

You can download and install the IBM® SPSS® Statistics data file driver from *http://www.spss.com/drivers/*. The *SPSS Statistics Data File Driver Guide*, available from the same location, contains information on installing and configuring the driver.

Example: Using the Standalone Driver with Excel

This section describes how to use the standalone ODBC driver to read IBM® SPSS® Statistics data files into another application. For information on using the driver to read SPSS Statistics data files into SPSS Statistics, see "Using the Standalone Driver" in Chapter 3.

This example describes how to use the standalone ODBC driver with Excel on a Windows operating system. You can, of course, use SAVE TRANSLATE /TYPE=XLS to save data in Excel format (as described previously in this chapter), but the ODBC driver enables someone without access to SPSS Statistics to read SPSS Statistics data files into Excel.

Defining a Data Source

To read SPSS Statistics data files into Excel using the standalone ODBC driver, you first need to define a data source for each data file.

▶ In the Windows Control Panel, choose Administrative Tools, and then choose Data Sources (ODBC).

▶ In the ODBC Data Source Administrator, click Add.

▶ In the Create New Data Source dialog, select SPSS Statistics 17 Data File Driver - Standalone and then click Finish.

▶ In the Standalone Setup dialog, enter the data source name. In this example, we'll simply use the name of the SPSS Statistics data file: *demo.sav*.

▶ For Statistics Data File Name, enter the full path and name of the data file. In this example, we'll use *c:\examples\data\demo.sav*.

Figure 9-1
Standalone data source setup dialog
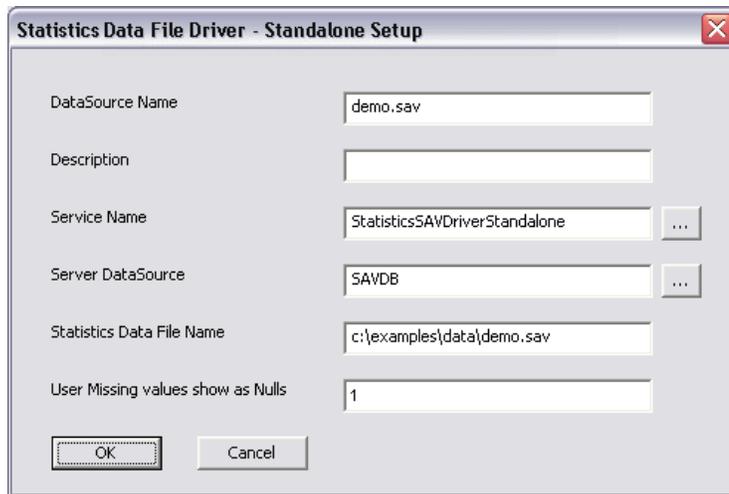
Note that User Missing Values Show as Nulls is set to 1. This means that user-missing values will be converted to null or blank. If you change the setting to 0, user-missing values are read as valid values. System-missing values are always converted to null or blank.

Reading the Data File in Excel

▶ In Excel 2003, from the menus choose:
Data > Import External Data > Import Data

▶ In the Select Data Source dialog, click New Source.

▶ In Excel 2007, from the menu tabs and ribbons, choose:
Data > From Other Sources > From Data Connection Wizard

▶ In the Data Connection Wizard (Excel 2003 and 2007), select ODBC DSN, and then click Next.

▶ In the Connect to ODBC Data Source step, select the data source name you defined for the SPSS Statistics data file from the list of ODBC data sources (in this example, demo.sav), and then click Next.

▶ In the Select Database and Table step, select the *Cases* table to read the data values or select the *CasesView* table to read the defined value labels. For any values without defined value labels, the data values are displayed.

Figure 9-2

CasesView table with value labels displayed in Excel

B	C	D	E	F
age	marital	address	income	inccat
55	Married	12	72	$50 - $74
56	Unmarried	29	153	$75+
28	Married	9	28	$25 - $49
24	Married	4	26	$25 - $49
25	Unmarried	2	23	Under $25
45	Married	9	76	$75+
42	Unmarried	19	40	$25 - $49
35	Unmarried	15	57	$50 - $74
46	Unmarried	26	24	Under $25

Exporting Results

You can export output in many formats, including: Word, Excel, PowerPoint, PDF, HTML, XML, and IBM® SPSS® Statistics data file format. This section covers only a few of the available formats and possibilities. For more information, see the descriptions of OMS and OUTPUT EXPORT in the *Command Syntax Reference*.

OMS vs. OUTPUT EXPORT

Both OMS and OUTPUT EXPORT provide the ability to export output in numerous formats, and in many respects both commands offer similar functionality. Here are some guidelines for determining which command is best for your needs:

- To export to XML or SPSS Statistics data file format, use OMS.
- To export to PowerPoint (Windows operating systems only), use OUTPUT EXPORT.
- To specify sheet names in Excel or append results to existing sheets in a workbook, use OUTPUT EXPORT.
- To control the treatment of wide tables in Word (wrap, shrink to fit) or control page size and margins, use OUTPUT EXPORT.
- To selectively export output based on object or table type, use OMS.
- To export output with the Batch Facility (available with SPSS Statistics Server), use OMS.

Exporting Output to Word/RTF

Both OUTPUT EXPORT and OMS can export output in Word/RTF format. OUTPUT EXPORT provides the ability to control the display of wide tables and set page dimensions. OMS provides the ability to control which types of output objects are included in the exported document.

Export to Word with OUTPUT EXPORT

```
*export_output_word.sps.
GET FILE='/examples/data/demo.sav'.
OUTPUT NEW.
CROSSTABS TABLES=inccat by ed.
OUTPUT EXPORT
  /DOC
```

```
        DOCUMENTFILE='/temp/wrap.doc'
      WIDETABLES=WRAP.
OUTPUT EXPORT
    /DOC
    DOCUMENTFILE='/temp/shrink.doc'
    WIDETABLES=SHRINK.
```

■ All output after the OUTPUT NEW command is exported to the Word document.

■ WIDETABLES=WRAP in the first OUTPUT EXPORT command specifies that wide tables should be wrapped. The table is divided into sections that will fit within the defined document width. Row labels are repeated for each section of the table.

■ WIDETABLES=SHRINK in the second OUTPUT EXPORT command specifies that wide tables should be scaled down to fit on the page. Font size and column width are reduced so that the table fits within the document width.

Not coincidentally, the crosstabulation produced by this job is too wide to fit in the default Word document width.

Figure 9-3
Wide table wrapped in Word

		Level of education			
		Did not complete high school	High school degree	Some college	College degree
Income category in thousands	Under $25	322	378	241	196
	$25 - $49	537	730	511	490
	$50 - $74	224	326	248	259
	$75+	307	502	360	410
	Total	1390	1936	1360	1355

		Level of education	
		Post-undergraduate degree	Total
Income category in thousands	Under $25	37	1174
	$25 - $49	120	2388
	$50 - $74	63	1120
	$75+	139	1718
	Total	359	6400

Figure 9-4
Wide table scaled down to fit Word page width

		Level of education					
		Did not complete high school	High school degree	Some college	College degree	Post- undergraduate degree	Total
Income category in thousands	Under $25	322	378	241	196	37	1174
	$25 - $49	537	730	511	490	120	2388
	$50 - $74	224	326	248	259	63	1120
	$75+	307	502	360	410	139	1718
Total		1390	1936	1360	1355	359	6400

The DOC subcommand also provides keywords for changing the page size and margins. For more information, see the OUTPUT EXPORT command in the *Command Syntax Reference*.

Export to Word with OMS

```
*oms_word.sps.
GET FILE='/examples/data/demo.sav'.
OMS SELECT TABLES
  /IF SUBTYPES=['Crosstabulation']
  /DESTINATION FORMAT=DOC
   OUTFILE='/temp/oms_word.doc'.
CROSSTABS TABLES=inccat by wireless.
DESCRIPTIVES VARIABLES=age.
CROSSTABS TABLES=marital by retire.
OMSEND.
```

- The OMS request encompasses all output created by any commands between the OMS and OMSEND commands.

- SELECT TABLES includes only pivot tables in the OMS output file. Logs, titles, charts, and other object types will not be included.

- IF SUBTYPES=['Crosstabulation'] narrows the selection down to crosstabulation tables. Notes tables and case processing summary tables will not be included, nor will any tables created by the DESCRIPTIVES command.

- DESTINATION FORMAT=DOC creates an OMS output file in Word/RTF format.

The end result is a Word document that contains only crosstabulation tables and no other output.

Figure 9-5
OMS results in Word

Income category in thousands * Wireless service Crosstabulation

Count

		Wireless service		
		No	Yes	Total
Income category in thousands	Under $25	822	352	1174
	$25 - $49	1480	908	2388
	$50 - $74	644	476	1120
	$75+	907	811	1718
	Total	3853	2547	6400

Marital status * Retired Crosstabulation

Count

		Retired		
		No	Yes	Total
Marital status	Unmarried	3069	155	3224
	Married	3023	153	3176
	Total	6092	308	6400

Exporting Output to Excel

Both OUTPUT EXPORT and OMS can export output in Excel format. OUTPUT EXPORT provides the ability to specify sheet names within the workbook, create a new sheet in an existing workbook, and append output to an existing sheet in a workbook. OMS provides the ability to control which types of output objects are included in the exported document.

Export to Excel with OUTPUT EXPORT

```
*output_export_excel.sps.
PRESERVE.
SET PRINTBACK OFF.
GET FILE='/examples/data/demo.sav'.
OUTPUT NEW.
FREQUENCIES VARIABLES=inccat.
OUTPUT EXPORT
  /CONTENTS EXPORT=VISIBLE
  /XLS DOCUMENTFILE='/temp/output_export.xls'
  OPERATION=CREATEFILE
  SHEET='Frequencies'.
OUTPUT NEW.
DESCRIPTIVES VARIABLES=age.
OUTPUT EXPORT
  /CONTENTS EXPORT=VISIBLE
  /XLS DOCUMENTFILE='/temp/output_export.xls'
  OPERATION=CREATESHEET
  SHEET='Descriptives'.
OUTPUT NEW.
FREQUENCIES VARIABLES=ownpda.
OUTPUT EXPORT
```

```
   /CONTENTS EXPORT=VISIBLE
   /XLS DOCUMENTFILE='/temp/output_export.xls'
   OPERATION=MODIFYSHEET
   SHEET='Frequencies'
   LOCATION=LASTROW.
RESTORE.
```

■ PRESERVE preserves the current SET command specifications.

■ SET PRINTBACK OFF turns off the display of commands in the Viewer. If commands are displayed in the Viewer, then they would also be included in the output exported to Excel.

■ All output after each OUTPUT NEW command is exported based on the specifications in the next OUTPUT EXPORT command.

■ CONTENTS=VISIBLE in all the OUTPUT EXPORT commands includes only visible objects in the Viewer. Since Notes tables are hidden by default, they will not be included.

■ XLS DOCUMENTFILE='/temp/output_export.xls' creates or modifies the named Excel file.

■ OPERATION=CREATEFILE in the first OUTPUT EXPORT command creates a new file.

■ SHEET='Frequencies' creates a named sheet in the file.

■ The output from all commands after the previous OUTPUT NEW commands is included on the sheet named *Frequencies*. This includes all the output from the FREQUENCIES command. If commands are displayed in the Viewer, it also includes the log of the OUTPUT NEW command—but we have turned off display of commands in the Viewer.

■ OPERATION=CREATESHEET combined with SHEET='Descriptives' in the second OUTPUT EXPORT command modifies the named Excel file to add the named sheet. In this example, it modifies the same Excel file we created with the previous OUTPUT EXPORT command, which will now have two sheets: *Frequencies* and *Descriptives*.

■ The new sheet in the Excel file will contain the output from the DESCRIPTIVES command.

■ OPERATION=MODIFYSHEET combined with SHEET='Frequencies' in the last OUTPUT EXPORT command modifies the content of the *Frequencies* sheet.

■ LOCATION=LASTROW specifies that the output should be appended at the bottom of the sheet, after the last row that contains any non-empty cells. In this example, the output from the last FREQUENCIES command will be inserted to the sheet below the output from the first FREQUENCIES command.

■ The job ends with a RESTORE command that restores your previous SET specifications.

Figure 9-6
Output Export results in Excel

Export to Excel with OMS

```
*oms_excel.sps.
GET FILE='/examples/data/demo.sav'.
OMS SELECT TABLES
  /IF SUBTYPES=['Frequencies']
  /DESTINATION FORMAT=XLS
  OUTFILE='/temp/frequencies.xls'.
OMS SELECT TABLES
  /IF SUBTYPES=['Descriptive Statistics']
  /DESTINATION FORMAT=XLS
  OUTFILE='/temp/descriptives.xls'.
FREQUENCIES VARIABLES=inccat.
DESCRIPTIVES VARIABLES=age.
FREQUENCIES VARIABLES=ownpda.
OMSEND.
```

■ The two OMS requests encompass all output created by any commands between the OMS
and OMSEND commands.

- SELECT TABLES in both OMS commands includes only pivot tables in the OMS output file. Logs, titles, charts, and other object types will not be included.

- IF SUBTYPES=['Frequencies'] in the first OMS command selects only frequency tables.

- IF SUBTYPES=['Descriptive Statistcs'] in the second OMS command selects only descriptive statistics tables.

- DESTINATION FORMAT=XLS creates an OMS output file in Excel.

The end result is two Excel files: one that contains the two frequency tables from the two FREQUENCIES commands, and one that contains the descriptive statistics table from the DESCRIPTIVES command.

Figure 9-7
OMS results in Excel

Using Output as Input with OMS

Using the OMS command, you can save pivot table output to IBM® SPSS® Statistics data files and then use that output as input in subsequent commands or sessions. This can be useful for many purposes. This section provides examples of two possible ways to use output as input:

- Generate a table of group summary statistics (percentiles) not available with the AGGREGATE command and then merge those values into the original data file.

- Draw repeated random samples with replacement from a data file, calculate regression coefficients for each sample, save the coefficient values in a data file, and then calculate confidence intervals for the coefficients (bootstrapping).

The command syntax files for these examples are installed with the product.

Adding Group Percentile Values to a Data File

Using the `AGGREGATE` command, you can compute various group summary statistics and then include those values in the active dataset as new variables. For example, you could compute mean, minimum, and maximum income by job category and then include those values in the dataset. Some summary statistics, however, are not available with the `AGGREGATE` command. This example uses `OMS` to write a table of group percentiles to a data file and then merges the data in that file with the original data file.

The command syntax used in this example is *oms_percentiles.sps*.

```
GET
  FILE='Employee data.sav'.
PRESERVE.
SET TVARS NAMES TNUMBERS VALUES.
DATASET DECLARE freq_table.

***split file by job category to get group percentiles.
SORT CASES BY jobcat.
SPLIT FILE LAYERED BY jobcat.

OMS
  /SELECT TABLES
  /IF COMMANDS=['Frequencies'] SUBTYPES=['Statistics']
  /DESTINATION FORMAT=SAV
   OUTFILE='freq_table'
  /COLUMNS SEQUENCE=[L1 R2].

FREQUENCIES
  VARIABLES=salary
  /FORMAT=NOTABLE
  /PERCENTILES= 25 50 75.

OMSEND.

***restore previous  SET settings.
RESTORE.

MATCH FILES FILE=*
  /TABLE='freq_table'
  /rename (Var1=jobcat)
  /BY jobcat
 /DROP command_ TO salary_Missing.
EXECUTE.
```

- The `PRESERVE` command saves your current `SET` command specifications.

- `SET TVARS NAMES TNUMBERS VALUES` specifies that variable names and data values, not variable or value labels, should be displayed in tables. Using variable names instead of labels is not technically necessary in this example, but it makes the new variable names constructed from column labels somewhat easier to work with. Using data values instead of value labels, however, is required to make this example work properly because we will use the job category values in the two files to merge them together.

- `SORT CASES` and `SPLIT FILE` are used to divide the data into groups by job category (*jobcat*). The `LAYERED` keyword specifies that results for each split-file group should be displayed in the same table rather than in separate tables.

- The `OMS` command will select all statistics tables from subsequent `FREQUENCIES` commands and write the tables to a data file.

- The `COLUMNS` subcommand will put the first layer dimension element and the second row dimension element in the columns.

■ The FREQUENCIES command produces a statistics table that contains the 25th, 50th, and 75th percentile values for salary. Since split-file processing is on, the table will contain separate percentile values for each job category.

Figure 9-8
Default and pivoted statistics tables

■ In the statistics table, the variable *salary* is the only layer dimension element, so the L1 specification in the OMS COLUMNS subcommand will put *salary* in the column dimension.

■ The table statistics are the second (inner) row dimension element in the table, so the R2 specification in the OMS COLUMNS subcommand will put the statistics in the column dimension, nested under the variable *salary*.

■ The data values 1, 2, and 3 are used for the categories of the variable *jobcat* instead of the descriptive text value labels because of the previous SET command specifications.

■ OMSEND ends all active OMS commands. Without this, we could not access the data file *temp.sav* in the subsequent MATCH FILES command because the file would still be open for writing.

Figure 9-9
Data file created from pivoted table

- The `MATCH FILES` command merges the contents of the dataset created from the statistics table with the original dataset. New variables from the data file created by `OMS` will be added to the original data file.

- `FILE=*` specifies the current active dataset, which is still the original data file.

- `TABLE='freq_table'` identifies the dataset created by `OMS` as a **table lookup file**. A table lookup file is a file in which data for each "case" can be applied to multiple cases in the other data file(s). In this example, the table lookup file contains only three cases—one for each job category.

- In the data file created by `OMS`, the variable that contains the job category values is named *Var1*, but in the original data file, the variable is named *jobcat*. `RENAME (Var1=jobcat)` compensates for this discrepancy in the variable names.

- `BY jobcat` merges the two files together by values of the variable *jobcat*. The three cases in the table lookup file will be merged with every case in the original data file with the same value for *jobcat* (also known as *Var1* in the table lookup file).

- Since we don't want to include the three table identifier variables (automatically included in every data file created by `OMS`) or the two variables that contain information on valid and missing cases, we use the `DROP` subcommand to omit these from the merged data file.

The end result is three new variables containing the 25th, 50th, and 75th percentile salary values for each job category.

Figure 9-10
Percentiles added to original data file

Bootstrapping with OMS

Bootstrapping is a method for estimating population parameters by repeatedly resampling the same sample—computing some test statistic on each sample and then looking at the distribution of the test statistic over all the samples. Cases are selected randomly, with replacement, from the original sample to create each new sample. Typically, each new sample has the same number of cases as the original sample; however, some cases may be randomly selected multiple times and others not at all. In this example, we

- use a macro to draw repeated random samples with replacement;

- run the `REGRESSION` command on each sample;

- use the `OMS` command to save the regression coefficients tables to a data file;
- produce histograms of the coefficient distributions and a table of confidence intervals, using the data file created from the coefficient tables.

The command syntax file used in this example is *oms_bootstrapping.sps*.

OMS Commands to Create a Data File of Coefficients

Although the command syntax file *oms_bootstrapping.sps* may seem long and/or complicated, the `OMS` commands that create the data file of sample regression coefficients are really very short and simple:

```
PRESERVE.
SET TVARS NAMES.
DATASET DECLARE bootstrap_example.
OMS /DESTINATION VIEWER=NO /TAG='suppressall'.
OMS
  /SELECT TABLES
  /IF COMMANDS=['Regression'] SUBTYPES=['Coefficients']
  /DESTINATION FORMAT=SAV OUTFILE='bootstrap_example'
  /COLUMNS DIMNAMES=['Variables'  'Statistics']
  /TAG='reg_coeff'.
```

- The `PRESERVE` command saves your current `SET` command specifications, and `SET TVARS NAMES` specifies that variable names—not labels—should be displayed in tables. Since variable names in data files created by `OMS` are based on table column labels, using variable names instead of labels in tables tends to result in shorter, less cumbersome variable names.

- `DATASET DECLARE` defines a dataset name that will then be used in the `REGRESSION` command.

- The first `OMS` command prevents subsequent output from being displayed in the Viewer until an `OMSEND` is encountered. This is not technically necessary, but if you are drawing hundreds or thousands of samples, you probably don't want to see the output of the corresponding hundreds or thousands of `REGRESSION` commands.

- The second `OMS` command will select coefficients tables from subsequent `REGRESSION` commands.

- All of the selected tables will be saved in a dataset named *bootstrap_example*. This dataset will be available for the rest of the current session but will be deleted automatically at the end of the session unless explicitly saved. The contents of this dataset will be displayed in a separate Data Editor window.

- The `COLUMNS` subcommand specifies that both the 'Variables' and 'Statistics' dimension elements of each table should appear in the columns. Since a regression coefficients table is a simple two-dimensional table with 'Variables' in the rows and 'Statistics' in the columns, if both dimensions appear in the columns, then there will be only one row (case) in the generated data file for each table. This is equivalent to pivoting the table in the Viewer so that both 'Variables' and 'Statistics' are displayed in the column dimension.

Figure 9-11

Variables dimension element pivoted into column dimension

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients		
		B	Std. Error	Beta	t	Sig.
1	(Constant)	-12120.813	3082.981		-3.932	.000
	Beginning Salary	1.914	.046	.882	41.271	.000
	Months since Hire	172.297	36.276	.102	4.750	.000

a. Dependent Variable: Current Salary

Coefficients[a]

	(Constant)				Beginning Salary		
	Unstandardized Coefficients				Unstandardized Coefficients		Standardized Coefficients
Mode	B	Std. Error	t	Sig.	B	Std. Error	Beta
1	-1.212E4	3082.981	-3.932	.000	1.914	.046	.882	41

a. Dependent Variable: Current Salary
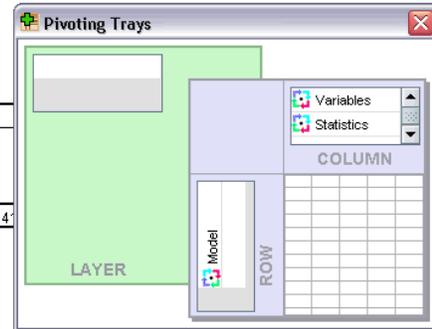
Sampling with Replacement and Regression Macro

The most complicated part of the OMS bootstrapping example has nothing to do with the OMS command. A macro routine is used to generate the samples and run the REGRESSION commands. Only the basic functionality of the macro is discussed here.

```
DEFINE regression_bootstrap (samples=!TOKENS(1)
                            /depvar=!TOKENS(1)
                            /indvars=!CMDEND)

COMPUTE dummyvar=1.
AGGREGATE
  /OUTFILE=* MODE=ADDVARIABLES
  /BREAK=dummyvar
  /filesize=N.
!DO !other=1 !TO !samples
SET SEED RANDOM.
WEIGHT OFF.
FILTER OFF.
DO IF $casenum=1.
- COMPUTE #samplesize=filesize.
- COMPUTE #filesize=filesize.
END IF.
DO IF (#samplesize>0 and #filesize>0).
- COMPUTE sampleWeight=rv.binom(#samplesize, 1/#filesize).
- COMPUTE #samplesize=#samplesize-sampleWeight.
- COMPUTE #filesize=#filesize-1.
ELSE.
- COMPUTE sampleWeight=0.
END IF.
WEIGHT BY sampleWeight.
FILTER BY sampleWeight.
REGRESSION
  /STATISTICS COEFF
  /DEPENDENT !depvar
  /METHOD=ENTER !indvars.
!DOEND
!ENDDEFINE.

GET FILE='/examples/data/Employee data.sav'.

regression_bootstrap
```

```
      samples=100
      depvar=salary
      indvars=salbegin jobtime.
```

- A macro named *regression_bootstrap* is defined. It is designed to work with arguments similar to IBM® SPSS® Statistics subcommands and keywords.

- Based on the user-specified number of samples, dependent variable, and independent variable, the macro will draw repeated random samples with replacement and run the REGRESSION command on each sample.

- The samples are generated by randomly selecting cases with replacement and assigning weight values based on how many times each case is selected. If a case has a value of 1 for *sampleWeight*, it will be treated like one case. If it has a value of 2, it will be treated like two cases, and so on. If a case has a value of 0 for *sampleWeight*, it will not be included in the analysis.

- The REGRESSION command is then run on each weighted sample.

- The macro is invoked by using the macro name like a command. In this example, we generate 100 samples from the *employee data.sav* file. You can substitute any file, number of samples, and/or analysis variables.

Ending the OMS Requests

Before you can use the generated dataset, you need to end the OMS request that created it, because the dataset remains open for writing until you end the OMS request. At that point, the basic job of creating the dataset of sample coefficients is complete, but we've added some histograms and a table that displays the 2.5th and 97.5th percentiles values of the bootstrapped coefficient values, which indicate the 95% confidence intervals of the coefficients.

```
OMSEND.
DATASET ACTIVATE bootstrap_example.
FREQUENCIES
  VARIABLES=salbegin_B salbegin_Beta jobtime_B jobtime_Beta
  /FORMAT NOTABLE
  /PERCENTILES= 2.5 97.5
  /HISTOGRAM  NORMAL.
RESTORE.
```

- OMSEND without any additional specifications ends all active OMS requests. In this example, there were two: one to suppress all Viewer output and one to save regression coefficients in a data file. If you don't end both OMS requests, either you won't be able to open the data file or you won't see any results of your subsequent analysis.

- The job ends with a RESTORE command that restores your previous SET specifications.

Figure 9-12

95% confidence interval (2.5th and 97.5th percentiles) and coefficient histograms

Statistics

		salbegin_B	salbegin_Beta	jobtime_B	jobtime_Beta
N	Valid	100	100	100	100
	Missing	0	0	0	0
Percentiles	2.5	1.71305	.83828	87.69077	.05271
	97.5	2.10343	.90552	254.97741	.14664

jobtime Unstandardized
Coefficients B

jobtime Standardized
Coefficients Beta
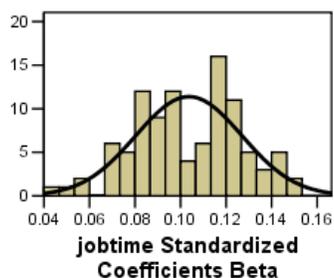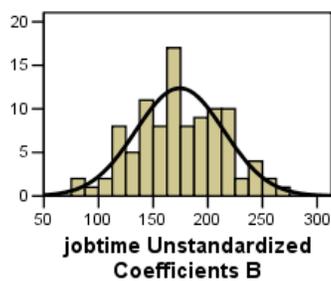
Transforming OXML with XSLT

Using the OMS command, you can route output to OXML, which is XML that conforms to the Output XML schema. This section provides a few basic examples of using XSLT to transform OXML.

■ These examples assume some basic understanding of XML and XSLT. If you have not used XML or XSLT before, this is not the place to start. There are numerous books and Internet resources that can help you get started.

■ All of the XSLT stylesheets presented here are installed with IBM® SPSS® Statistics in the *Samples* folder.

■ The Output XML schema is documented in the SPSS Statistics help system.

OMS Namespace

Output XML produced by OMS contains a namespace declaration:

xmlns="http://xml.spss.com/spss/oms"

In order for XSLT stylesheets to work properly with OXML, the XSLT stylesheets must contain a similar namespace declaration that also defines a prefix that is used to identify that namespace in the stylesheet. For example:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0" xmlns:oms="http://xml.spss.com/spss/oms">
```

This defines "oms" as the prefix that identifies the namespace; therefore, all of the XPath expressions that refer to OXML elements by name must use "oms:" as a prefix to the element name references. All of the examples presented here use the "oms:" prefix, but you could define and use a different prefix.

"Pushing" Content from an XML File

In the "push" approach, the structure and order of elements in the transformed results are usually defined by the source XML file. In the case of OXML, the structure of the XML mimics the nested tree structure of the Viewer outline, and we can construct a very simple XSLT transformation to reproduce the outline structure.

This example generates the outline in HTML, but it could just as easily generate a simple text file. The XSLT stylesheet is *oms_simple_outline_example.xsl*.

Figure 9-13
Viewer Outline Pane
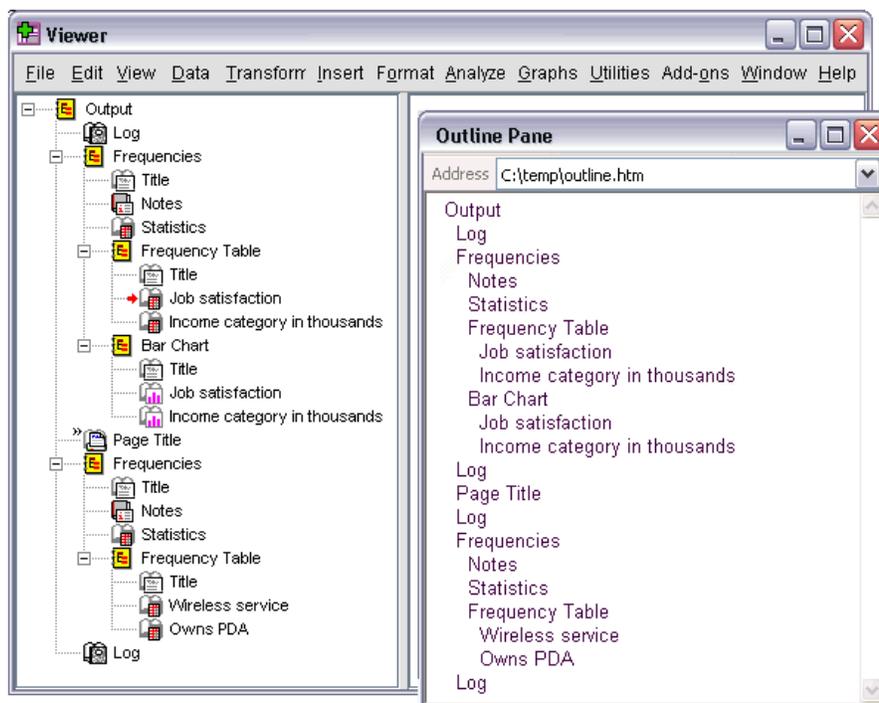
Figure 9-14
XSLT stylesheet oms_simple_outline_example.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
 xmlns:oms="http://xml.spss.com/spss/oms">

<xsl:template match="/">
 <HTML>
  <HEAD>
   <TITLE>Outline Pane</TITLE>
  </HEAD>
  <BODY>
  <br/>Output
   <xsl:apply-templates/>
  </BODY>
 </HTML>
</xsl:template>

<xsl:template match="oms:command|oms:heading">
 <xsl:call-template name="displayoutline"/>
  <xsl:apply-templates/>
</xsl:template>
<xsl:template match="oms:textBlock|oms:pageTitle|oms:pivotTable|oms:chartTitle">
 <xsl:call-template name="displayoutline"/>
</xsl:template>

<!--indent based on number of ancestors:
two spaces for each ancestor-->
<xsl:template name="displayoutline">
 <br/>
 <xsl:for-each select="ancestor::*">
  <xsl:text>  </xsl:text>
 </xsl:for-each>
 <xsl:value-of select="@text"/>
 <xsl:if test="not(@text)">
  <!--no text attribute, must be page title-->
  <xsl:text>Page Title</xsl:text>
 </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

- xmlns:oms="http://xml.spss.com/spss/oms" defines "oms" as the prefix that identifies the namespace, so all element names in XPath expressions need to include the prefix "oms:".
- The stylesheet consists mostly of two template elements that cover each type of element that can appear in the outline—command, heading, textBlock, pageTitle, pivotTable, and chartTitle.
- Both of those templates call another template that determines how far to indent the text attribute value for the element.

- The `command` and `heading` elements can have other outline items nested under them, so the template for those two elements also includes `<xsl:apply-templates/>` to apply the template for the other outline items.

- The template that determines the outline indentation simply counts the number of "ancestors" the element has, which indicates its nesting level, and then inserts two spaces (` ` is a "nonbreaking" space in HTML) before the value of the `text` attribute value.

- `<xsl:if test="not(@text)">` selects `<pageTitle>` elements because this is the only specified element that doesn't have a `text` attribute. This occurs wherever there is a `TITLE` command in the command file. In the Viewer, it inserts a page break for printed output and then inserts the specified page title on each subsequent printed page. In OXML, the `<pageTitle>` element has no attributes, so we use `<xsl:text>` to insert the text "Page Title" as it appears in the Viewer outline.

Viewer Outline "Titles"

You may notice that there are a number of "Title" entries in the Viewer outline that don't appear in the generated HTML. These should not be confused with page titles. There is no corresponding element in OXML because the actual "title" of each output block (the text object selected in the Viewer if you click the "Title" entry in the Viewer outline) is exactly the same as the text of the entry directly above the "Title" in the outline, which is contained in the `text` attribute of the corresponding `command` or `heading` element in OXML.

"Pulling" Content from an XML File

In the "pull" approach, the structure and order of elements in the source XML file may not be relevant for the transformed results. Instead, the source XML is treated like a data repository from which selected pieces of information are extracted, and the structure of the transformed results is defined by the XSLT stylesheet.

The "pull" approach typically uses `<xsl:for-each>` to select and extract information from the XML.

Simple xsl:for-each "Pull" Example

This example uses `<xsl:for-each>` to "pull" selected information out of OXML output and create customized HTML tables.

Although you can easily generate HTML output using `DESTINATION FORMAT=HTML` on the `OMS` command, you have very little control over the HTML generated beyond the specific object types included in the HTML file. Using OXML, however, you can create customized tables. This example

- selects only frequency tables in the OXML file;

- displays only valid (nonmissing) values;

- displays only the *Frequency* and *Valid Percent* columns;

- replaces the default column labels with *Count* and *Percent*.

The XSLT stylesheet used in this example is *oms_simple_frequency_tables.xsl*.

Note: This stylesheet is not designed to work with frequency tables generated with layered split-file processing.

Figure 9-15
Frequencies pivot tables in Viewer

Variable One

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	One	19	18.1	26.0	26.0
	Two	28	26.7	38.4	64.4
	3.00	26	24.8	35.6	100.0
	Total	73	69.5	100.0	
Missing	99.00	17	16.2		
	System	15	14.3		
	Total	32	30.5		
Total		105	100.0		

var2

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	Female	63	60.0	60.0	60.0
	Male	42	40.0	40.0	100.0
	Total	105	100.0	100.0	

Figure 9-16
Customized HTML frequency tables

Variable One

Category	Count	Percent
One	19	26.0
Two	28	38.4
3.00	26	35.6
Total	73	100.0

var2

Category	Count	Percent
Female	63	60.0
Male	42	40.0
Total	105	100.0

Figure 9-17
XSLT stylesheet: oms_simple_frequency_tables.xsl

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0" xmlns:oms="http://xml.spss.com/spss/oms">
<!--enclose everything in a template, starting at the root node-->
<xsl:template match="/">
<HTML>
<HEAD>
<TITLE>Modified Frequency Tables</TITLE>
</HEAD>
```

```
<BODY>
<!-Find all Frequency Tables->
<xsl:for-each select="//oms:pivotTable[@subType='Frequencies']">
<xsl:for-each select="oms:dimension[@axis='row']">
 <h3>
  <xsl:value-of select="@text"/>
 </h3>
</xsl:for-each>
<!-create the HTML table->
<table border="1">
 <tbody align="char" char="." charoff="1">
 <tr>
 <!-
 table header row; you could extract headings from
 the XML but in this example we're using different header text
 ->
 <th>Category</th><th>Count</th><th>Percent</th>
 </tr>
<!-find the columns of the pivot table->
<xsl:for-each select="descendant::oms:dimension[@axis='column']">
  <!-select only valid, skip missing->
   <xsl:if test="ancestor::oms:group[@text='Valid']">
    <tr>
    <td>
     <xsl:choose>
      <xsl:when test="not((parent::*)[@text='Total'])">
       <xsl:value-of select="parent::*/@text"/>
      </xsl:when>
      <xsl:when test="((parent::*)[@text='Total'])">
       <b><xsl:value-of select="parent::*/@text"/></b>
      </xsl:when>
     </xsl:choose>
    </td>
    <td>
     <xsl:value-of select="oms:category[@text='Frequency']/oms:cell/@text"/>
    </td>
    <td>
     <xsl:value-of select="oms:category[@text='Valid Percent']/oms:cell/@text"/>
    </td>
   </tr>
  </xsl:if>
 </xsl:for-each>
</tbody>
</table>
<!-Don't forget possible footnotes for split files->
<xsl:if test="descendant::*/oms:note">
<p><xsl:value-of select="descendant::*/oms:note/@text"/></p>
</xsl:if>
</xsl:for-each>
</BODY>
</HTML>
</xsl:template>
```

```
</xsl:stylesheet>
```

- `xmlns:oms="http://xml.spss.com/spss/oms"` defines "oms" as the prefix that identifies the namespace, so all element names in XPath expressions need to include the prefix "oms:".

- The XSLT primarily consists of a series of nested `<xsl:for-each>` statements, each drilling down to a different element and attribute of the table.

- `<xsl:for-each select="//oms:pivotTable[@subType='Frequencies']">` selects all tables of the subtype 'Frequencies'.

- `<xsl:for-each select="oms:dimension[@axis='row']">` selects the row dimension of each table.

- `<xsl:for-each select="descendant::oms:dimension[@axis='column']">` selects the column elements from each row. OXML represents tables row by row, so column elements are nested within row elements.

- `<xsl:if test="ancestor::oms:group[@text='Valid']">` selects only the section of the table that contains valid, nonmissing values. If there are no missing values reported in the table, this will include the entire table. This is the first of several XSLT specifications in this example that rely on attribute values that differ for different output languages. If you don't need solutions that work for multiple output languages, this is often the simplest, most direct way to select certain elements. Many times, however, there are alternatives that don't rely on localized text strings. For more information, see the topic "Advanced xsl:for-each "Pull" Example" on p. 157.

- `<xsl:when test="not((parent::*)[@text='Total'])">` selects column elements that aren't in the 'Total' row. Once again, this selection relies on localized text, and the only reason we make the distinction between total and nontotal rows in this example is to make the row label 'Total' bold.

- `<xsl:value-of select="oms:category[@text='Frequency']/oms:cell/@text"/>` gets the content of the cell in the 'Frequency' column of each row.

- `<xsl:value-of select="oms:category[@text='Valid Percent']/oms:cell/@text"/>` gets the content of the cell in the 'Valid Percent' column of each row. Both this and the previous code for obtaining the value from the 'Frequency' column rely on localized text.

Figure 9-18
XPath expressions for selected frequency table elements
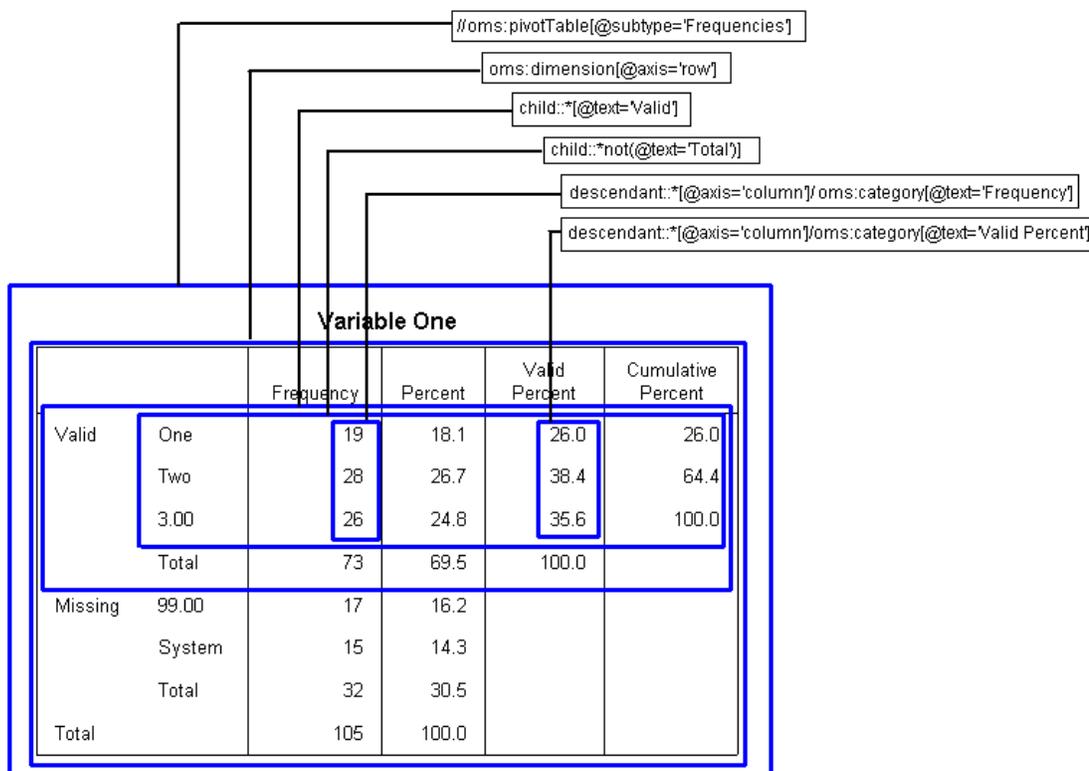
Advanced xsl:for-each "Pull" Example

In addition to selecting and displaying only selected parts of each frequency table in HTML format, this example

- doesn't rely on any localized text;
- always shows both variable names and labels;
- always shows both values and value labels;
- rounds decimal values to integers.

The XSLT stylesheet used in this example is *customized_frequency_tables.xsl*.

Note: This stylesheet is not designed to work with frequency tables generated with layered split-file processing.

Figure 9-19
Customized HTML with value rounded to integers

Variable Name: var1

Variable Label: Variable One

Category	Count	Percent
1: One	19	26
2: Two	28	38
3	26	36
Total	73	100

Variable Name: var2

Category	Count	Percent
f: Female	63	60
m: Male	42	40
Total	105	100

The simple example contained a single XSLT <template> element. This stylesheet contains multiple templates:

- A main template that selects the table elements from the OXML
- A template that defines the display of variable names and labels
- A template that defines the display of values and value labels
- A template that defines the display of cell values as rounded integers

The following sections explain the different templates used in the stylesheet.

Main Template for Advanced xsl:for-each Example

Since this XSLT stylesheet produces tables with essentially the same structure as the simple <xsl:for-each> example, the main template is similar to the one used in the simple example.

Figure 9-20
Main template of customized_frequency_tables.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0" xmlns:oms="http://xml.spss.com/spss/oms">

<!--enclose everything in a template, starting at the root node-->
<xsl:template match="/">
<HTML>
<HEAD>
<TITLE>Modified Frequency Tables</TITLE>
</HEAD>
<BODY>
<xsl:for-each select="//oms:pivotTable[@subType='Frequencies']">
<xsl:for-each select="oms:dimension[@axis='row']">
```

```
 <h3>
  <xsl:call-template name="showVarInfo"/>
 </h3>
</xsl:for-each>
<!--create the HTML table-->
<table border="1">
 <tbody align="char" char="." charoff="1">
 <tr>  <th>Category</th><th>Count</th><th>Percent</th>
 </tr>
 <xsl:for-each select="descendant::oms:dimension[@axis='column']">
  <xsl:if test="oms:category[3]">
   <tr>
    <td>
     <xsl:choose>
      <xsl:when test="parent::*/@varName">
       <xsl:call-template name="showValueInfo"/>
      </xsl:when>
      <xsl:when test="not(parent::*/@varName)">
       <b><xsl:value-of select="parent::*/@text"/></b>
      </xsl:when>
      </xsl:choose>
    </td>
    <td>
     <xsl:apply-templates select="oms:category[1]/oms:cell/@number"/>
    </td>
    <td>
     <xsl:apply-templates select="oms:category[3]/oms:cell/@number"/>
    </td>
   </tr>
  </xsl:if>
  </xsl:for-each>
 </tbody>
</table>
<xsl:if test="descendant::*/oms:note">
<p><xsl:value-of select="descendant::*/oms:note/@text"/></p>
</xsl:if>
</xsl:for-each>
</BODY>
</HTML>
</xsl:template>
```

This template is very similar to the one for the simple example. The main differences are:

■   <xsl:call-template name="showVarInfo"/> calls another template to determine what to
    show for the table title instead of simply using the text attribute of the row dimension
    (oms:dimension[@axis='row']). For more information, see the topic "Controlling Variable and
    Value Label Display" on p. 160.

■   <xsl:if test="oms:category[3]"> selects only the data in the 'Valid' section of the table instead
    of <xsl:if test="ancestor::oms:group[@text='Valid']">. The positional argument used in this
    example doesn't rely on localized text. It relies on the fact that the basic structure of a
    frequency table is always the same and the fact that OXML does not include elements for

empty cells. Since the 'Missing' section of a frequency table contains values only in the first two columns, there are no oms:category[3] column elements in the 'Missing' section, so the test condition is not met for the 'Missing' rows. For more information, see the topic "XPath Expressions in Multiple Language Environments" on p. 162.

- <xsl:when test="parent::*/@varName"> selects the nontotal rows instead of <xsl:when test="not((parent::*)[@text='Total'])">. Column elements in the nontotal rows in a frequency table contain a varName attribute that identifies the variable, whereas column elements in total rows do not. So this selects nontotal rows without relying on localized text.

- <xsl:call-template name="showValueInfo"/> calls another template to determine what to show for the row labels instead of <xsl:value-of select="parent::*/@text"/>. For more information, see the topic "Controlling Variable and Value Label Display" on p. 160.

- <xsl:apply-templates select="oms:category[1]/oms:cell/@number"/> selects the value in the 'Frequency' column instead of <xsl:value-of select="oms:category[@text='Frequency']/oms:cell/@text"/>. A positional argument is used instead of localized text (the 'Frequency' column is always the first column in a frequency table), and a template is applied to determine how to display the value in the cell. Percentage values are handled the same way, using oms:category[3] to select the values from the 'Valid Percent' column. For more information, see the topic "Controlling Decimal Display" on p. 161.

Controlling Variable and Value Label Display

The display of variable names and/or labels and values and/or value labels in pivot tables is determined by the current settings for SET TVARS and SET TNUMBERS—the corresponding text attributes in the OXML also reflect those settings. The system default is to display labels when they exist and names or values when they don't. The settings can be changed to always show names or values and never show labels or always show both.

The XSLT templates *showVarInfo* and *showValueInfo* are designed to ignore those settings and always show both names or values and labels (if present).

Figure 9-21
showVarInfo and showValueInfo templates

```
<!–display both variable names and labels–>
<xsl:template name="showVarInfo">
 <p>
 <xsl:text>Variable Name: </xsl:text>
 <xsl:value-of select="@varName"/>
 </p>
 <xsl:if test="@label">
 <p>
  <xsl:text>Variable Label: </xsl:text>
  <xsl:value-of select="@label"/>
 </p>
 </xsl:if>
</xsl:template>

<!–display both values and value labels–>
<xsl:template name="showValueInfo">
```

```
<xsl:choose>
<!-Numeric vars have a number attribute,
 string vars have a string attribute ->
 <xsl:when test="parent::*/@number">
  <xsl:value-of select="parent::*/@number"/>
 </xsl:when>
 <xsl:when test="parent::*/@string">
  <xsl:value-of select="parent::*/@string"/>
 </xsl:when>
</xsl:choose>
<xsl:if test="parent::*/@label">
  <xsl:text>: </xsl:text>
  <xsl:value-of select="parent::*/@label"/>
 </xsl:if>
</xsl:template>
```

- <xsl:text>Variable Name: </xsl:text> and <xsl:value-of select="@varName"/> display the text "Variable Name:" followed by the variable name.

- <xsl:if test="@label"> checks to see if the variable has a defined label.

- If the variable has a defined label, <xsl:text>Variable Label: </xsl:text> and <xsl:value-of select="@label"/> display the text "Variable Label:" followed by the defined variable label.

- Values and value labels are handled in a similar fashion, except instead of a varName attribute, values will have either a number attribute or a string attribute.

Controlling Decimal Display

The text attribute of a <cell> element in OXML displays numeric values with the default number of decimal positions for the particular type of cell value. For most table types, there is little or no control over the default number of decimals displayed in cell values in pivot tables, but OXML can provide some flexibility not available in default pivot table display.

In this example, the cell values are rounded to integers, but we could just as easily display five or six or more decimal positions because the number attribute may contain up to 15 significant digits.

Figure 9-22
Rounding cell values

```
<!-round decimal cell values to integers->
<xsl:template match="@number">
 <xsl:value-of select="format-number(.,'#')"/>
</xsl:template>
```

- This template is invoked whenever <apply-templates select="..."/> contains a reference to a number attribute.

- <xsl:value-of select="format-number(.,'#')"/> specifies that the selected values should be rounded to integers with no decimal positions.

XPath Expressions in Multiple Language Environments

Text Attributes

Most table elements contain a `text` attribute that contains the information as it would appear in a pivot table in the current output language. For example, the column in a frequency table that contains counts is labeled *Frequency* in English but *Frecuencia* in Spanish.

For XPath expressions that need to work in a multiple language environment, it is recommended to use the `text_eng` attribute, whose value is the English value of the `text` attribute regardless of the output language. For example, in the case of *Frequency* discussed above the associated `text_eng` attribute would always have the value `'Frequency'`, so your XPath expression would contain `@text_eng='Frequency'` instead of `@text='Frequency'`. The `OATTRS` subcommand of the `SET` command specifies whether `text_eng` attributes are included in OXML output.

Positional Arguments

For many table types you can use positional arguments that are not affected by output language. For example, in a frequency table the column that contains counts is always the first column, so a positional argument of `category[1]` at the appropriate level of the tree structure should always select information in the column that contains counts.

In some table types, however, the elements in the table and order of elements in the table can vary. For example, the order of statistics in the columns or rows of table subtype "Report" generated by the `MEANS` command is determined by the specified order of the statistics on the `CELLS` subcommand. In fact, two tables of this type may not even display the same statistics at all. So `category[1]` might select the category that contains mean values in one table, median values in another table, and nothing at all in another table.

Layered Split-File Processing

Layered split-file processing can alter the basic structure of tables that you might otherwise assume have a fixed default structure. For example, a standard frequency table has only one row dimension (`dimension axis="row"`), but a frequency table of the same variable when layered split-file processing is in effect will have multiple row dimensions, and the total number of dimensions—and row label columns in the table—depends on the number of split-file variables and unique split-file values.

Figure 9-23
Standard and layered frequencies tables

Standard Frequency Table

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	One	19	18.1	26.0	26.0
	Two	28	26.7	38.4	64.4
	3.00	26	24.8	35.6	100.0
	Total	73	69.5	100.0	
Missing	99.00	17	16.2		
	System	15	14.3		
	Total	32	30.5		
Total		105	100.0		

Frequency Table with Layered Split-File Processing

var2			Frequency	Percent	Valid Percent	Cumulative Percent
Female	Valid	One	14	22.2	29.2	29.2
		Two	20	31.7	41.7	70.8
		3.00	14	22.2	29.2	100.0
		Total	48	76.2	100.0	
	Missing	System	15	23.8		
	Total		63	100.0		
Male	Valid	One	5	11.9	20.0	20.0
		Two	8	19.0	32.0	52.0
		3.00	12	28.6	48.0	100.0
		Total	25	59.5	100.0	
	Missing	99.00	17	40.5		
	Total		42	100.0		

Controlling and Saving Output Files

In addition to exporting results in external formats for use in other applications, you can also control how output is routed to different output windows using the OUTPUT commands.

The OUTPUT commands (OUTPUT NEW, OUTPUT NAME, OUTPUT ACTIVATE, OUTPUT OPEN, OUTPUT SAVE, OUTPUT CLOSE) provide the ability to programmatically manage one or many output documents. These functions allow you to:

- Save an output document through syntax.

- Programmatically partition output into separate output documents (for example, results for males in one output document and results for females in a separate one).

- Work with multiple open output documents in a given session, selectively appending new results to the appropriate document.

Example

```
*save_output.sps.
OUTPUT CLOSE NAME=ALL.
DATA LIST LIST /GroupVar SummaryVar.
BEGIN DATA
1 1
1 2
1 3
2 4
2 5
```

```
2 6
END DATA.
OUTPUT NEW NAME=group1.
COMPUTE filterVar=(GroupVar=1).
FILTER BY filterVar.
FREQUENCIES VARIABLES=SummaryVar.
OUTPUT SAVE OUTFILE='/temp/group1.spv'.
OUTPUT NEW NAME=group2.
COMPUTE filterVar=(GroupVar=2).
FILTER BY filterVar.
FREQUENCIES VARIABLES=SummaryVar.
OUTPUT SAVE OUTFILE='/temp/group2.spv'.
FILTER OFF.
```

- ■ OUTPUT CLOSE NAME=ALL closes all currently open output documents. (It does not save output documents; anything in those documents not previously saved is gone.)

- ■ OUTPUT NEW creates a new output document and makes it the active output document. Subsequent output will be sent to that document. Specifying names for the output documents allows you to switch between open output documents (using OUTPUT ACTIVATE, which is not used in this example).

- ■ OUTPUT SAVE saves the currently active output document to a file.

- ■ In this example, output for the two groups defined by *GroupVar* is sent to two different output documents, and then those two output documents are saved.

Scoring data with predictive models

The process of applying a predictive model to a set of data is referred to as **scoring** the data. IBM® SPSS® Statistics has procedures for building predictive models such as regression, clustering, tree, and neural network models. Once a model has been built, the model specifications can be saved in a file that contains all of the information necessary to reconstruct the model. You can then use that model file to generate predictive scores in other datasets. (*Note:* Some procedures produce a model XML file, and some procedures produce a ZIP archive file.)

Example. The direct marketing division of a company uses results from a test mailing to assign propensity scores to the rest of their contact database, using various demographic characteristics to identify contacts most likely to respond and make a purchase.

Scoring is treated as a transformation of the data. The model is expressed internally as a set of numeric transformations to be applied to a given set of fields (variables)—the predictors specified in the model—in order to obtain a predicted result. In this sense, the process of scoring data with a given model is inherently the same as applying any function, such as a square root function, to a set of data.

The scoring process consists of two basic steps:

▶ Build the model and save the model file. You build the model using a dataset for which the outcome of interest (often referred to as the **target**) is known. For example, if you want to build a model that will predict who is likely to respond to a direct mail campaign, you need to start with a dataset that already contains information on who responded and who did not respond. For example, this might be the results of a test mailing to a small group of customers or information on responses to a similar campaign in the past.

Note: For some model types there is no target outcome of interest. Clustering models, for example, do not have a target, and some nearest neighbor models do not have a target.

▶ Apply that model to a different dataset (for which the outcome of interest is not known) to obtain predicted outcomes.

Note: Starting with version 19, scoring is available in the core system. In prior releases, scoring required access to SPSS Statistics Server.

Building a predictive model

IBM® SPSS® Statistics provides numerous procedures for building predictive models. This example builds a binary logistic regression model in which the target outcome of interest has only two possible outcomes. In this example, the contacts who received the test mailing either responded (made a purchase) or did not respond (did not make a purchase).

This example uses two data files: *dmdata2.sav* is used to build the model, and then that model is applied to *dmdata3.sav*.

To build the model, we use the LOGISTIC REGRESSION procedure.

```
*build_logistic_model.sps.
GET FILE '/examples/data/dmdata2.sav'.
SET RNG=MT MTINDEX=2000000.
COMPUTE Partition=RV.BERNOULLI(.5).
LOGISTIC REGRESSION VARIABLES Responded
  /METHOD=ENTER Age Income Education Reside Gender Married Region
  /CONTRAST (Income)=INDICATOR
  /CONTRAST (Education)=INDICATOR
  /CONTRAST (Gender)=INDICATOR
  /CONTRAST (Married)=INDICATOR
  /CONTRAST (Region)=INDICATOR
  /CRITERIA CUT(0.05)
  /SELECT Partition EQ 1
  /OUTFILE = PARAMETER ("/temp/logistic_model.xml").
```

- The SET command isn't necessary. It's only included to replicate the results of the following COMPUTE command.

- The COMPUTE command creates a new variable *Partition*, and the BERNOULLI function randomly assigns the value 1 to half the cases and 0 to the other half. The values of *Partition* will be used to split cases into training and testing samples in the LOGISTIC REGRESSION command.

- The LOGISTIC REGRESSION command uses the variable *Responded* as the target (dependent) variable. Contacts in the test mailing who responded positively (made a purchase) have a value of 1; those who did not respond have a value of 0.

- The CRITERIA subcommand specifies a cut-off value of 0.05. This does not affect the actual scoring model that will be created. It affects only the results of the classification table which we will use to evaluate the model. Contacts with a predicted probability of responding that is greater than or equal to 0.05 (5%) will be classified as positive responders.

- The SELECT subcommand uses the values of Partition to split the contacts into training and testing groups. The model will be built with contacts in the training group, and then tested on contacts in the testing group.

- The OUTFILE subcommand saves the model information in an XML file that conforms to PMML standards. This model file will be used to calculate predicted probabilities for contacts in the other data file.

Evaluating the model

Before you use the model for scoring purposes, you should evaluate how good the model is. The kind of results available for evaluating the model depend on the technique used to generate the model. This example uses the results available with the LOGISTIC REGRESSION procedure, available in the Regression add-on option.

The direct marketing division has determined that they need a positive response rate of at least 5% for the campaign to be profitable; so they want to know if the model will identify a group of contacts likely to result in a 5% positive response rate. The classification table provides useful information in this regard.

Figure 10-1
Classification table

Classification Table^c

			Predicted					
			Selected Cases^a			Unselected Cases^b		
			Responded to test offer			Responded to test offer		
Observed			No	Yes	Percentage Correct	No	Yes	Percentage Correct
Step 1	Responded to test offer	No	651	249	72.3	653	267	71.0
		Yes	19	20	51.3	36	22	37.9
	Overall Percentage				71.5			69.0

a. Selected cases Partition EQ 1
b. Unselected cases Partition NE 1
c. The cut value is .050

■ The classification table is divided into two sections: *Selected Cases* are the contacts in the test mailing used to build the model, and *Unselected Cases* are the contacts in the test mailing used to test the model.

■ In the *Selected Cases* group, 269 (249+20) were classified in the "Yes" group, but only 20 of those actually responded. That makes the correct classification rate for predicted positive responders 7.3% (20/269), which is higher than the minimum required response rate of 5%.

■ The model built with the *Selected Cases* is then tested on the remaining cases. For the *Unselected Cases*, the correct classification rate for predicted positive responders is 22/(267+22) or 7.6%. This is fairly close to correct classification rate for the cases used to build the model, and is also over the 5% minimum response rate. This suggests that the model should be appropriate to identify the group of contacts likely to yield a 5% or higher positive response rate.

Applying the model

The model can be applied to any data file that contains the same measures used as predictors in the model file. The variable names don't have to be exactly the same. You can map model variable names to variable names in the data file you want to score. Since the purpose of applying the model is to predict values associated with the target variable, the target variable does not need to be present in the data file you want to score.

In this example, the data file to be scored, *dmdata3.sav*, contains all the measures used to build the model, except the variable named *Income* in the data file used to build the model (and consequently named *Income* in the model file) is named *IncomeCategory* in the data file to be scored. (*Note*: This variable represents the same income categories in both files. Otherwise it would not be valid to map *IncomeCategory* to *Income*.)

```
*score_logistic_model.sps.
GET FILE='/examples/data/dmdata3.sav'.
MODEL HANDLE NAME=logistic_model FILE='/temp/logistic_model.xml'
 /MAP VARIABLES=IncomeCategory  MODELVARIABLES=Income.
COMPUTE ProbabilityOfResponding=APPLYMODEL(logistic_model, 'PROBABILITY', 1).
EXECUTE.
MODEL CLOSE NAME=logistic_model.
```

- The MODEL HANDLE command specifies the model file to use, which in this example is the model file we created with the LOGISTIC REGRESSION command.

- The NAME assigned to the model is used in the subsequent COMPUTE command to identify which model to use when computing scores (since you can have multiple models open simultaneously).

- The MAP subcommand maps the variable *IncomeCategory* in the active dataset to *Income* in the model. The data file being scored must contain variables that match all variables used in the model, otherwise scores cannot be computed.

- The COMPUTE command specifies the APPLYMODEL function, which has two required arguments: model name and scoring function.

- With the PROBABILITY function used in this example, you can also specify an optional third argument, which is the target value for which you want the predicted probability. Without this third argument, the PROBABILITY function returns the probability of the predicted value for each case. But we are specifically interested in the probability of a positive response, which is represented by the value 1 in the model. (The available scoring functions depend on the model. For details, search for "applymodel" in the help system.)

- The optional MODEL CLOSE command closes the model.

A new variable, *ProbabilityofResponding*, is appended to the end of the active dataset. This contains the probability, expressed as a proportion, that each contact will respond to the campaign. You could then create a new dataset that contains only those contacts with a predicted probability of equal to or greater than 5%, as in:

```
DATASET COPY MailList.
DATASET ACTIVATE MailList.
SELECT IF ProbabilityOfResponding >=.05.
EXECUTE.
```

Part II:
Programming with Python

Introduction

The IBM® SPSS® Statistics - Integration Plug-In for Python is one of a family of Integration Plug-Ins that also includes .NET and R. It provides two separate interfaces for programming with the Python language within IBM® SPSS® Statistics on Windows, Linux, and Mac OS, as well as for SPSS Statistics Server.

spss module. The spss module provides functions that operate on the SPSS Statistics processor and extend the SPSS Statistics command syntax language with the full capabilities of the Python programming language. This interface allows you to access SPSS Statistics variable dictionary information, case data, and procedure output, from within Python code. You can also submit command syntax to SPSS Statistics for processing, create new variables and new cases in the active dataset, create new datasets, and create output in the form of pivot tables and text blocks, all from within Python code. Python code that utilizes the spss module is referred to as a **Python program**.

SpssClient module. The SpssClient module provides functions that operate on user interface and output objects. With this interface, you can customize pivot tables, export items such as charts and tables in a variety of formats, invoke SPSS Statistics dialog boxes, and manage connections to instances of SPSS Statistics Server, all from within Python code. The SpssClient module provides functionality similar to what is available with Windows-only Basic scripts. A Python module that, directly or indirectly, utilizes the SpssClient module, without the presence of the spss module, is referred to as a **Python script**. Scripts are concerned with objects in the user interface and the Viewer.

A wide variety of tasks can be accomplished in a programmatic fashion with these interfaces.

Control the Flow of a Command Syntax Job

You can write Python programs to control the execution of syntax jobs based on variable properties, case data, procedure output, error codes, or conditions such as the presence of specific files or environment variables. With this functionality, you can:

■ Conditionally run command syntax only when a particular variable exists in the active dataset or the case data meet specified criteria.

■ Decide on a course of action if a command fails to produce a meaningful result, such as an iterative process that doesn't converge.

■ Determine whether to proceed with execution or halt a job if an error arises during the execution of command syntax.

Dynamically Create and Submit Command Syntax

Python programs can dynamically construct command syntax and submit it to SPSS Statistics for processing. This allows you to dynamically tailor command specifications to the current variable dictionary, the case data in the active dataset, procedure output, or virtually any other information from the environment. For example, you can create a Python program to:

■ Dynamically create a list of variables from the active dataset that have a particular attribute and then use that list as the variable list for a given syntax command.

■ Perform data management operations on a dynamically selected set of files—for example, combine cases from all SPSS Statistics data files located in a specified directory.

Apply Custom Algorithms to Your Data

Access to case data allows you to use the power of the Python language to perform custom calculations on your data. This opens up the possibility of using the vast set of scientific programming libraries available for the Python language. Python programs can write the results back to the active dataset, to a new dataset, or as pivot table output directed to the Viewer or exported via the Output Management System (OMS). In short, you can write custom procedures in the Python language that have almost the same capabilities as SPSS Statistics procedures, such as `DESCRIPTIVES` and `REGRESSION`.

Create and Manage Multiple Datasets

In addition to accessing the active dataset, Python programs can concurrently access multiple open datasets as well as create new datasets. This allows you to create one or more new datasets from existing datasets, combining the data from the existing datasets in any way you choose. It also allows you to concurrently modify multiple datasets—perhaps with results from an analysis—without having to explicitly activate each one.

Customize Pivot Tables

You can write Python scripts that customize just about any aspect of a pivot table, including labels, data cells, and footnotes. You can run your scripts manually, set them up to run as autoscripts to be triggered by the creation of specified output objects for selected procedures, or call scripting functions from within Python programs by first importing the `SpssClient` module. You can also include your customizations in the base autoscript, which is applied to all new output objects before any autoscripts for specific output objects are applied.

Develop and Debug Code Using Third-Party IDEs

You can use the Python IDE of your choice to develop and debug both Python programs and Python scripts. IDEs typically include a rich set of tools for creating and debugging software, such as editors that do code completion and syntax highlighting and debuggers that allow you to step through your code and inspect variable and attribute values. In fact, you can build entire applications based on Python programs that externally drive SPSS Statistics from a Python IDE or from a separate Python process, such as the Python interpreter.

Prerequisites

The Integration Plug-In for Python works with SPSS Statistics release 14.0.1 or later and requires only the Core system. The Plug-In, along with installation instructions, is available for download from Developer Central at *http://www.spss.com/devcentral/*. For release 18 on Windows, the Plug-In is installed with Python Essentials. For release 19 and higher and for all platforms, the Plug-In is installed with IBM® SPSS® Statistics - Essentials for Python.

The chapters that follow include hands-on examples of Python programs and Python scripts and assume a basic working knowledge of the Python programming language, although aspects of the language are discussed when deemed necessary. For help getting started with the Python programming language, see the Python tutorial, available at *http://docs.python.org/tut/tut.html*.

Note: SPSS Inc. is not the owner or licensor of the Python software. Any user of Python must agree to the terms of the Python license agreement located on the Python Web site. SPSS Inc. does not make any statement about the quality of the Python program. SPSS Inc. fully disclaims all liability associated with your use of the Python program.

Additional Plug-Ins

The IBM® SPSS® Statistics Programmability Extension, included with the Core system, provides a general framework for supporting external languages through Integration Plug-Ins, such as the Integration Plug-In for Python. In particular, there are also freeware Integration Plug-Ins for .NET and R, available from Developer Central. The IBM® SPSS® Statistics - Integration Plug-In for Microsoft® .NET supports development in any .NET language and is intended for applications that interact with the SPSS Statistics backend but present their own user interface and provide their own means for displaying output. The IBM® SPSS® Statistics - Integration Plug-In for R provides access to the extensive set of statistical routines available in R and runs from within the SPSS Statistics client. For more information, see Programming with R on p. 339

Getting Started with Python Programming in IBM SPSS Statistics

The spss Python Module

The `spss` Python module, installed with the IBM® SPSS® Statistics - Integration Plug-In for Python, enables you to:

- Build and run command syntax.
- Get information about data in the current IBM® SPSS® Statistics session.
- Get data, add new variables, and append cases to the active dataset.
- Create new datasets.
- Concurrently access multiple open datasets.
- Get output results.
- Create custom pivot tables and text blocks.
- Create macro variables.
- Get error information.
- Manage multiple versions of the Integration Plug-In for Python.

The functionality available with the module is accessed by including the Python statement `import spss` as the first line in a `BEGIN PROGRAM-END PROGRAM` program block within command syntax, as in:

```
BEGIN PROGRAM PYTHON.
import spss
spss.Submit("SHOW ALL.")
END PROGRAM.
```

- The keyword `PYTHON` on the `BEGIN PROGRAM` command specifies that the program block contains Python code. Within a `PYTHON` program block, you have full access to all of the functionality of the Python programming language. The keyword `PYTHON` is the default for `BEGIN PROGRAM` and can be omitted.
- You need to include the `import spss` statement only once in a given SPSS Statistics session. Repeating an `import` statement in subsequent `BEGIN PROGRAM` blocks essentially has no effect.
- As you'll learn in a subsequent topic, the `Submit` function allows you to send commands to SPSS Statistics for processing. The prefix `spss` in `spss.Submit` specifies that this function can be found in the `spss` module.

Note: To run the above program, simply include the code in the Syntax Editor and run it like any other block of command syntax.

For functions that are commonly used, like `Submit`, you can omit the `spss` prefix by including the statement `from spss import <function name>` before the first call to the function. For example:

```
BEGIN PROGRAM.
import spss
from spss import Submit
Submit("SHOW ALL.")
END PROGRAM.
```

When included in a program block, output from the Python `print` statement is directed to a log item in the SPSS Statistics Viewer if a Viewer is available, as shown in the following program block.

```
BEGIN PROGRAM.
print "Hello, world!"
END PROGRAM.
```

Figure 12-1
Output from BEGIN PROGRAM displayed in a log item
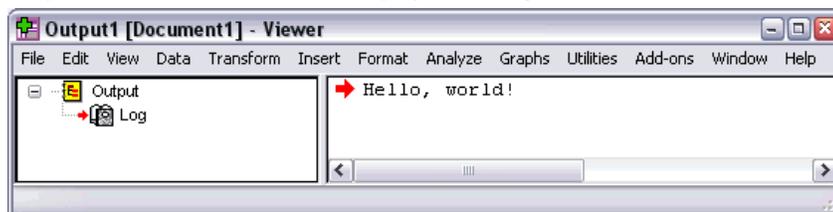
Many of the functions in the `spss` module are used in examples in the sections that follow. A brief description for a particular function is also available using the Python `help` function. For example, adding the statement `help(spss.Submit)` to a program block results in the display of a brief description of the `Submit` function in a log item in the Viewer.

Complete documentation for the `spss` module is available in the SPSS Statistics Help system, under Integration Plug-In for Python Help. The documentation is also available in PDF from Help>Programmability>Python Plug-in, within SPSS Statistics, once the Integration Plug-In for Python is installed.

Running Your Code from a Python IDE

You can run code utilizing the `spss` module from any Python IDE (Integrated Development Environment). IDEs typically include a rich set of tools for creating and debugging software, such as editors that do code completion and syntax highlighting, and debuggers that allow you to step through your code and inspect variable and attribute values. Once you've completed code development in an IDE, you can include it in a `BEGIN PROGRAM-END PROGRAM` block within command syntax.

To run your code from a Python IDE, simply include an `import spss` statement in the IDE's code window. You can follow the `import` statement with calls to any of the functions in the `spss` module, just like with program blocks in command syntax jobs, but you don't include the `BEGIN PROGRAM-END PROGRAM` statements. A sample session using the PythonWin IDE (a freely available IDE for working with the Python programming language on Windows) is

shown below, and it illustrates a nice feature of using an IDE—the ability to run code one line at a time and examine the results.

Figure 12-2
Driving IBM SPSS Statistics from a Python IDE
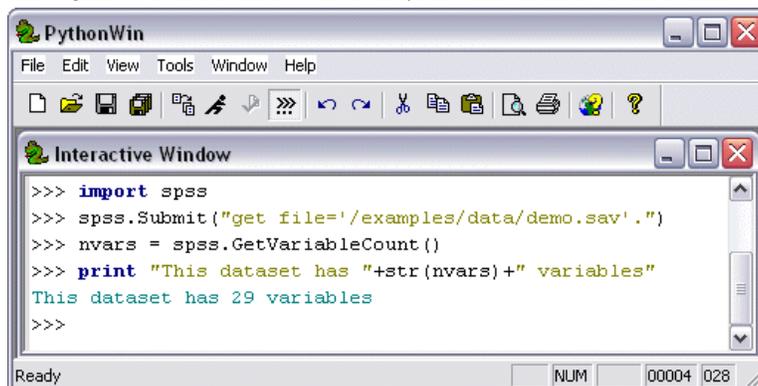
When you submit syntax commands that would normally generate Viewer output, the output is directed to the IDE's output window, as shown below. In that regard, when you run code utilizing the `spss` module from an external process (such as a Python IDE) the code starts up an instance of the IBM® SPSS® Statistics backend, without an associated Viewer.

Figure 12-3
Output from IBM SPSS Statistics command displayed in a Python IDE
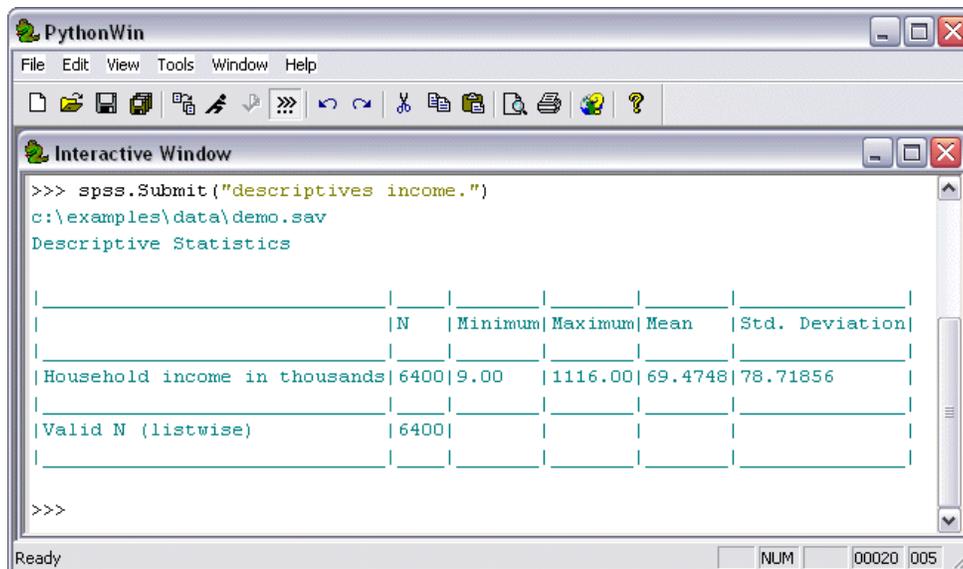
You can suppress output that would normally go to the SPSS Statistics Viewer by calling the `SetOutput` function in the `spss` module. The code `spss.SetOutput("OFF")` suppresses output and `spss.SetOutput("ON")` turns it back on. By default, output is displayed.

It can also be useful to programmatically determine whether the SPSS Statistics backend is being driven by an external Python process. The check is done with the function `spss.PyInvokeSpss.IsXDriven`, which returns 1 if a Python process, such as an IDE, is driving the SPSS Statistics backend and 0 if SPSS Statistics is driving the backend.

Note: You can drive the SPSS Statistics backend with any separate Python process, such as the Python interpreter. Once you've installed the IBM® SPSS® Statistics - Integration Plug-In for Python, you initiate this mode with the `import spss` statement, just like driving the SPSS Statistics backend from a Python IDE.

The SpssClient Python Module

The `SpssClient` Python module, installed with the IBM® SPSS® Statistics - Integration Plug-In for Python, enables you to:

- Customize pivot tables and text output
- Export items, such as charts and tables, in a variety of formats
- Invoke IBM® SPSS® Statistics dialog boxes
- Manage connections to instances of SPSS Statistics Server

Note: The `SpssClient` module provides functionality similar to what is available with Windows-only Basic scripts. For guidance on choosing the right technology for your task, see "Choosing the Best Programming Technology" on p. 205.

The `SpssClient` module can be used within a `BEGIN PROGRAM-END PROGRAM` program block or within a standalone Python module, referred to as a Python script. When used within a `BEGIN PROGRAM-END PROGRAM` block, the module allows you to seamlessly integrate code that customizes output with the code that generates the output. When used as part of a standalone Python module, the `SpssClient` module allows you to create general purpose scripts that can be invoked as needed from the SPSS Statistics client.

Whether used in a program block or in a standalone module, the basic structure of code that utilizes the `SpssClient` module is:

```
import SpssClient
SpssClient.StartClient()
<Python language statements>
SpssClient.StopClient()
```

- The `import SpssClient` statement imports the classes and methods available in the `SpssClient` module.
- `SpssClient.StartClient()` provides a connection to the associated SPSS Statistics client, enabling the code to retrieve information from the client and to perform operations on objects managed by the client, such as pivot tables. If the code is run from the SPSS Statistics client, a connection is established to that client. If the code is run from an external Python process (such as a Python IDE or the Python interpreter), an attempt is made to connect to an existing SPSS Statistics client. If more than one client is found, a connection is made to the most recently launched one. If an existing client is not found, a new and invisible instance of the SPSS Statistics client is started and a connection to it is established.

- `SpssClient.StopClient()` terminates the connection to the SPSS Statistics client and should be called at the completion of the code that utilizes the `SpssClient` module.

Using the SpssClient Module in a Program Block

```
*python_SpssClient_module_in_program_block.sps.
BEGIN PROGRAM.
import spss, SpssClient

# Code utilizing the spss module to generate output
spss.StartProcedure("Demo")
textBlock = spss.TextBlock("Sample text block",
                           "A single line of text.")
spss.EndProcedure()

# Code utilizing the SpssClient module to modify the output
SpssClient.StartClient()
OutputDoc = SpssClient.GetDesignatedOutputDoc()
OutputItems = OutputDoc.GetOutputItems()
OutputItem = OutputItems.GetItemAt(OutputItems.Size()-1)
LogItem = OutputItem.GetSpecificType()
text = LogItem.GetTextContents()
LogItem.SetTextContents("<html><B>" + text + "</B></html>")
SpssClient.StopClient()

END PROGRAM.
```

- The `import` statement includes both the `spss` and `SpssClient` modules.
- The first section of the code uses the `StartProcedure` and `TextBlock` functions from the `spss` module to create a text block in the Viewer.
- The second section of the code uses functions in the `SpssClient` module to change the style of the text in the text block to bold.

Using the SpssClient Module in an Extension Command

Extension commands are custom commands that provide the ability to invoke external functions written in Python or R from command syntax. For extension commands implemented in Python (or implemented in R but wrapped in Python), the underlying code is processed as part of an implicit BEGIN PROGRAM-END PROGRAM block and thus supports inclusion of the `SpssClient` module. In other words, the code that implements an extension command can invoke methods in the `SpssClient` module that act on the output generated by the command. For more information, see the topic "Extension Commands" in Chapter 31 on p. 381.

Using the SpssClient Module in a Standalone Module

This example iterates through the designated output document and changes the text style of all title items to italic.

```
#SamplePythonScript.py
import SpssClient
SpssClient.StartClient()
OutputDoc = SpssClient.GetDesignatedOutputDoc()
OutputItems = OutputDoc.GetOutputItems()
for index in range(OutputItems.Size()):
    OutputItem = OutputItems.GetItemAt(index)
    if OutputItem.GetType() == SpssClient.OutputItemType.TITLE:
        TitleItem = OutputItem.GetSpecificType()
        text = TitleItem.GetTextContents()
        TitleItem.SetTextContents("<html><I>" + text + "</I></html>")
SpssClient.StopClient()
```

Standalone Python modules that directly or indirectly utilize the `SpssClient` module, without the `spss` module, are referred to as Python scripts. The Python script shown above is contained in the module *SamplePythonScript.py*, included with the accompanying examples. Python scripts can be created from File > New > Script (within the SPSS Statistics client) when Python is specified as the default script language. The default script language is set from the Scripts tab in the Options dialog box and is preset to Basic on Windows and Python on Linux and Mac OS.

Invoking Python Scripts

Python scripts can be invoked in the following ways:

■ Interactively from Utilities > Run Script by selecting the Python module (*.py*) file containing the script.

■ Interactively from the Python editor launched from SPSS Statistics (accessed from File > Open > Script) by selecting the Python module (*.py*) file containing the script. Running the script from the Python editor allows you to use the debugging tools available with the editor.

   Note: Python programs (code that utilizes the `spss` module) are not intended to be run from the Python editor launched from SPSS Statistics.

■ Implicitly as an autoscript. Autoscripts are scripts that run automatically when triggered by the creation of specific pieces of output from selected procedures and are typically used to reformat a particular table type beyond what you can do by applying a TableLook. For example, you could set up an autoscript to reformat Statistics tables created by the `FREQUENCIES` command.

■ From an external Python process. You can run a Python script from any external Python process, such as a Python IDE that is not launched from SPSS Statistics, or the Python interpreter.

■ Automatically at the start of each session or each time you switch servers. For more information, see "Scripting Facility" in the Help system.

Getting Help

■ For more general information on Python scripts and autoscripts, see "Scripting Facility" in the Help system.

- For more examples of scripts, see "Modifying and Exporting Output Items" on p. 319.
- Complete documentation for the `SpssClient` module is available in the SPSS Statistics Help system, under Integration Plug-In for Python Help. The documentation is also available in PDF from Help>Programmability>Scripting, within SPSS Statistics, once the Integration Plug-In for Python is installed.

Submitting Commands to IBM SPSS Statistics

The common task of submitting command syntax from a program block is done using the `Submit` function from the `spss` module. In its simplest usage, the function accepts a quoted string representing a syntax command and submits the command text to IBM® SPSS® Statistics for processing. For example,

```
BEGIN PROGRAM.
import spss
spss.Submit("FREQUENCIES VARIABLES=var1, var2, var3.")
END PROGRAM.
```

imports the `spss` module and submits a `FREQUENCIES` command to SPSS Statistics.

Functions in the `spss` module enable you to retrieve information from, or run command syntax on, the active dataset. You can load a dataset prior to a `BEGIN PROGRAM` block as in:

```
GET FILE='/examples/data/Employee data.sav'.
BEGIN PROGRAM.
import spss
spss.Submit("FREQUENCIES VARIABLES=gender, educ, jobcat, minority.")
END PROGRAM.
```

or you can use the `Submit` function to load a dataset from within a program block as in:

```
BEGIN PROGRAM.
import spss
spss.Submit(["GET FILE='/examples/data/Employee data.sav'.",
     "FREQUENCIES VARIABLES=gender, educ, jobcat, minority."])
END PROGRAM.
```

- As illustrated in this example, the `Submit` function can accept a list of strings, each of which consists of a single syntax command. A list in Python is indicated by enclosing the items in square brackets.
- For Windows users, notice that the file specification uses the forward slash (/) instead of the usual backslash (\\). Escape sequences in the Python programming language begin with a backslash (\\), so using a forward slash prevents an unintentional escape sequence. And SPSS Statistics always accepts a forward slash in file specifications. Windows users can include backslashes and avoid escape sequences by using a raw string for the file specification. For more information, see the topic "Using Raw Strings in Python" in Chapter 13 on p. 202.

Command syntax generated within a program block and submitted to SPSS Statistics must follow interactive syntax rules. For most practical purposes, this means that command syntax strings that you build in a programming block must contain a period (.) at the end of each syntax command. The period is optional if the argument to the `Submit` function contains only one command. If you want to include a file of commands in a session and the file contains `BEGIN PROGRAM`

blocks, you must use the INSERT command in interactive mode (the default), as opposed to the INCLUDE command.

When you submit commands for SPSS Statistics procedures from BEGIN PROGRAM blocks, you can embed the procedure calls in Python loops, thus repeating the procedure many times but with specifications that change for each iteration. That's something you can't do with the looping structures (LOOP-END LOOP and DO REPEAT-END REPEAT) available in command syntax because the loop commands are transformation commands, and you can't have procedures inside such structures.

Example

Consider a regression analysis where you want to investigate different scenarios for a single predictor. Each scenario is represented by a different variable, so you need repeated runs of the Regression procedure, using a different variable each time. Setting aside the task of building the list of variables for the different scenarios, you might have something like:

```
for var in varlist:
    spss.Submit("REGRESSION /DEPENDENT res /METHOD=ENTER " + var + ".")
```

- varlist is meant to be a Python list containing the names of the variables for the different scenarios.

- On each iteration of the for loop, var is the name of a different variable in varlist. The value of var is then inserted into the command string for the REGRESSION command.

Dynamically Creating Command Syntax

Using the functions in the spss module, you can dynamically compose command syntax based on dictionary information and/or data values in the active dataset.

Example

Run the DESCRIPTIVES procedure, but only on the scale variables in the active dataset.

```
*python_desc_on_scale_vars.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
varList=[]
for i in range(spss.GetVariableCount()):
    if spss.GetVariableMeasurementLevel(i)=='scale':
        varList.append(spss.GetVariableName(i))
if len(varList):
    spss.Submit("DESCRIPTIVES " + " ".join(varList) + ".")
END PROGRAM.
```

The program block uses four functions from the spss module:

- spss.GetVariableCount returns the number of variables in the active dataset.

- spss.GetVariableMeasurementLevel(i) returns the measurement level of the variable with index value *i*. The index value of a variable is the position of the variable in the dataset, starting with the index value 0 for the first variable in file order. Dictionary information is accessed one variable at a time.

- `spss.GetVariableName(i)` returns the name of the variable with index value *i*, so you can build a list of scale variable names. The list is built with the Python list method `append`.

- `spss.Submit` submits the string containing the syntax for the DESCRIPTIVES command to IBM® SPSS® Statistics. The set of variables included on the DESCRIPTIVES command comes from the Python variable *varList*, which is a Python list, but the argument to the `Submit` function in this case is a string. The list is converted to a string using the Python string method `join`, which creates a string from a list by concatenating the elements of the list, using a specified string as the separator between elements. In this case, the separator is `" "`, a single space. In the present example, *varList* has the value `['id','bdate','salary','salbegin','jobtime','prevexp']`. The completed string is:

  `DESCRIPTIVES id bdate salary salbegin jobtime prevexp.`

When you're submitting a single command to SPSS Statistics, it's usually simplest to call the `Submit` function with a string representing the command, as in the above example. You can submit multiple commands with a single call to `Submit` by passing to `Submit` a list of strings, each of which represents a single syntax command. You can also submit a block of commands as a single string that spans multiple lines, resembling the way you might normally write command syntax. For more information, see the topic "Creating Blocks of Command Syntax within Program Blocks" in Chapter 13 on p. 199.

Capturing and Accessing Output

Functionality provided with the `spss` module allows you to access IBM® SPSS® Statistics procedure output in a programmatic fashion. This is made possible through an in-memory workspace—referred to as the **XML workspace**—that can contain an XML representation of procedural output. Output is directed to the workspace with the OMS command and retrieved from the workspace with functions that employ XPath expressions. For the greatest degree of control, you can work with OMS or XPath explicitly or you can use utility functions, available in supplementary modules, that construct appropriate OMS commands and XPath expressions for you, given a few simple inputs.

Example

In this example, we'll run the Descriptives procedure on a set of variables, direct the output to the XML workspace, and retrieve the mean value of one of the variables. The example assumes that variables in labels in Pivot Tables are displayed as the associated variable label (as set from the Output Labels tab on the Options dialog box).

```
*python_retrieve_output_value.sps.
BEGIN PROGRAM.
import spss,spssaux
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
cmd="DESCRIPTIVES VARIABLES=salary,salbegin,jobtime,prevexp."
desc_table,errcode=spssaux.CreateXMLOutput(
                          cmd,
                          omsid="Descriptives")
meansal=spssaux.GetValuesFromXMLWorkspace(
                desc_table,
                tableSubtype="Descriptive Statistics",
                rowCategory="Current Salary",
                colCategory="Mean",
                cellAttrib="text")
if meansal:
    print "The mean salary is: ", meansal[0]
END PROGRAM.
```

- The `BEGIN PROGRAM` block starts with an `import` statement for two modules: `spss` and
  `spssaux`. `spssaux` is a supplementary module that is installed with the IBM® SPSS®
  Statistics - Integration Plug-In for Python. Among other things, it contains two functions
  for working with procedure output: `CreateXMLOutput` generates an `OMS` command to
  route output to the XML workspace, and it submits both the `OMS` command and the original
  command to SPSS Statistics; and `GetValuesFromXMLWorkspace` retrieves output from the
  XML workspace without the explicit use of XPath expressions.

- The call to `CreateXMLOutput` includes the command as a quoted string to be submitted to
  SPSS Statistics and the associated OMS identifier (available from the OMS Identifiers dialog
  box on the Utilities menu). In this example, we're submitting a `DESCRIPTIVES` command,
  and the associated OMS identifier is "Descriptives." Output generated by `DESCRIPTIVES`
  will be routed to the XML workspace and associated with an identifier whose value is
  stored in the variable *desc_table*. The variable *errcode* contains any error level from the
  `DESCRIPTIVES` command—0 if no error occurs.

- In order to retrieve information from the XML workspace, you need to provide the identifier
  associated with the output—in this case, the value of *desc_table*. That provides the first
  argument to the `GetValuesFromXMLWorkspace` function.

- We're interested in the mean value of the variable for current salary. If you were to look at
  the Descriptives output in the Viewer, you would see that this value can be found in the
  Descriptive Statistics table on the row for the variable *Current Salary* and under the *Mean*
  column. These same identifiers—the table name, row name, and column name—are used to
  retrieve the value from the XML workspace, as you can see in the arguments used for the
  `GetValuesFromXMLWorkspace` function.

- In the general case, `GetValuesFromXMLWorkspace` returns a list of values—for example,
  the values in a particular row or column in an output table. Even when only one value is
  retrieved, as in this example, the function still returns a list structure, albeit a list with a single
  element. Since we are interested in only this single value (the value with index position 0 in
  the list), we extract it from the list. *Note*: If the XPath expression does not match anything in
  the workspace object, you will get back an empty list.

For more information, see the topic "Retrieving Output from Syntax Commands" in Chapter 17
on p. 279.

Modifying Pivot Table Output

The `SpssClient` module provides methods that allow you to customize pivot tables in output documents.

Example

This example illustrates code that accesses each pivot table in the designated output document and changes the text style to bold.

```
#ChangePivotTableTextStyle.py
import SpssClient
SpssClient.StartClient()

OutputDoc = SpssClient.GetDesignatedOutputDoc()
OutputItems = OutputDoc.GetOutputItems()

for index in range(OutputItems.Size()):
   OutputItem = OutputItems.GetItemAt(index)
   if OutputItem.GetType() == SpssClient.OutputItemType.PIVOT:
      PivotTable = OutputItem.GetSpecificType()
      PivotTable.SelectTable()
      PivotTable.SetTextStyle(SpssClient.SpssTextStyleTypes.SpssTSBold)
SpssClient.StopClient()
```

■ The `GetDesignatedOutputDoc` method of the `SpssClient` class returns an object representing the designated output document (the current document to which output is routed). The `GetOutputItems` method of the output document object returns a list of objects representing the items in the output document, such as pivot tables, charts, and log items.

■ The `for` loop iterates through the list of items in the output document. Pivot tables are identified as an output item type of `SpssClient.OutputItemType.PIVOT`.

■ Once an output item has been identified as a pivot table, you get an object representing the pivot table by calling the `GetSpecificType` method on the output item object. In this example, *PivotTable* is a pivot table object.

■ The `SelectTable` method of the pivot table object selects all elements of the table and the `SetTextStyle` method is used to set the text style to bold.

You can include this code with the code that generates the pivot tables or use it as a standalone Python script that you can invoke in a variety of ways. For more information, see the topic "The SpssClient Python Module" on p. 178. For more information about the methods used in this example, see "Modifying and Exporting Output Items" on p. 319.

Python Syntax Rules

Within a program block, only statements and functions recognized by the Python processor are allowed. Python syntax rules differ from IBM® SPSS® Statistics command syntax rules in a number of ways:

Python is case-sensitive. This includes Python variable names, function names, and pretty much anything else you can think of. A Python variable name of *myvariable* is not the same as *MyVariable*, and the Python function `spss.GetVariableCount` is not the same as `SPSS.getvariablecount`.

There is no command terminator in Python, and continuation lines come in two flavors:

■ **Implicit.** Expressions enclosed in parentheses, square brackets, or curly braces can continue across multiple lines (at natural break points) without any continuation character. Quoted strings contained in such an expression cannot continue across multiple lines unless they are triple-quoted. The expression continues implicitly until the closing character for the expression is encountered. For example, lists in the Python programming language are enclosed in square brackets, functions contain a pair of parentheses (whether they take any arguments or not), and dictionaries are enclosed in curly braces so that they can all span multiple lines.

■ **Explicit.** All other expressions require a backslash at the end of each line to explicitly denote continuation.

Line indentation indicates grouping of statements. Groups of statements contained in conditional processing and looping structures are identified by indentation. There is no statement or character that indicates the end of the structure. Instead, the indentation level of the statements defines the structure, as in:

```
for i in range(varcount):
    """A multi-line comment block enclosed in a pair of
    triple-quotes."""
    if spss.GetVariableMeasurementLevel(i)=="scale":
        ScaleVarList.append(spss.GetVariableName(i))
    else:
        CatVarList.append(spss.GetVariableName(i))
```

As shown here, you can include a comment block that spans multiple lines by enclosing the text in a pair of triple-quotes. If the comment block is to be part of an indented block of code, the first set of triple quotes must be at the same level of indentation as the rest of the block. Avoid using tab characters in program blocks that are read by SPSS Statistics.

Escape sequences begin with a backslash. The Python programming language uses the backslash (\\) character as the start of an escape sequence; for example, `"\n"` for a newline and `"\t"` for a tab. This can be troublesome when you have a string containing one of these sequences, as when specifying file paths on Windows, for example. The Python programming language offers a number of options for dealing with this. For any string where you just need the backslash character, you can use a double backslash (\\\\). For strings specifying file paths, you can use forward slashes (/) instead of backslashes. You can also specify the string as a raw string by prefacing it with an r or R; for example, `r"c:\temp"`. Backslashes in raw strings are treated as the backslash character, not as the start of an escape sequence. For more information, see the topic "Using Raw Strings in Python" in Chapter 13 on p. 202.

Python Quoting Conventions

■ Strings in the Python programming language can be enclosed in matching single quotes (`'`) or double quotes (`"`), as in SPSS Statistics.

■ To specify an apostrophe (single quote) within a string, enclose the string in double quotes. For example,

```
"Joe's Bar and Grille"
```

is treated as

```
Joe's Bar and Grille
```

- To specify quotation marks (double quotes) within a string, use single quotes to enclose the string, as in

```
'Categories Labeled "UNSTANDARD" in the Report'
```

- The Python programming language treats double quotes of the same type as the outer quotes differently from SPSS Statistics. For example,

```
'Joe''s Bar and Grille'
```

is treated as

```
Joes Bar and Grille
```

in Python; that is, the concatenation of the two strings `'Joe'` and `'s Bar and Grille'`.

Mixing Command Syntax and Program Blocks

Within a given command syntax job, you can intersperse `BEGIN PROGRAM-END PROGRAM` blocks with any other syntax commands, and you can have multiple program blocks in a given job. Python variables assigned in a particular program block are available to subsequent program blocks, as shown in this simple example:

```
*python_multiple_program_blocks.sps.
DATA LIST FREE /var1.
BEGIN DATA
1
END DATA.
DATASET NAME File1.
BEGIN PROGRAM.
import spss
File1N=spss.GetVariableCount()
END PROGRAM.
DATA LIST FREE /var1 var2 var3.
BEGIN DATA
1 2 3
END DATA.
DATASET NAME File2.
BEGIN PROGRAM.
File2N=spss.GetVariableCount()
if File2N > File1N:
   message="File2 has more variables than File1."
elif File1N > File2N:
   message="File1 has more variables than File2."
else:
   message="Both files have the same number of variables."
print message
END PROGRAM.
```

- The first program block contains the `import spss` statement. This statement is not required in the second program block.

- The first program block defines a programmatic variable, *File1N*, with a value set to the number of variables in the active dataset. The Python code in a program block is executed when the `END PROGRAM` statement in that block is reached, so the variable *File1N* has a value prior to the second program block.

- Prior to the second program block, a different dataset becomes the active dataset, and the second program block defines a programmatic variable, *File2N*, with a value set to the number of variables in that dataset.

- The value of *File1N* persists from the first program block, so the two variable counts can be compared in the second program block.

Passing Values from a Program Block to Command Syntax

Within a program block, you can define a macro variable that can be used outside of the block in command syntax. This provides the means to pass values computed in a program block to command syntax that follows the block. Although you can run command syntax from Python using the Submit function, this is not always necessary. The method described here shows you how to use Python statements to compute what you need and then continue on with the rest of your syntax job, making use of the results from Python. As an example, consider building separate lists of the categorical and scale variables in a dataset and then submitting a FREQUENCIES command for any categorical variables and a DESCRIPTIVES command for any scale variables. This example is an extension of an earlier one where only scale variables were considered. For more information, see the topic "Dynamically Creating Command Syntax" on p. 182.

```
*python_set_varlist_macros.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
catlist=[]
scalist=[]
for i in range(spss.GetVariableCount()):
   varName=spss.GetVariableName(i)
   if spss.GetVariableMeasurementLevel(i) in ['nominal', 'ordinal']:
      catlist.append(varName)
   else:
      scalist.append(varName)
if len(catlist):
   categoricalVars = " ".join(catlist)
   spss.SetMacroValue("!catvars", categoricalVars)
if len(scalist):
   scaleVars = " ".join(scalist)
   spss.SetMacroValue("!scavars", scaleVars)
END PROGRAM.

FREQUENCIES !catvars.
DESCRIPTIVES !scavars.
```

- The for loop builds separate Python lists of the categorical and scale variables in the active dataset.

- The SetMacroValue function in the spss module takes a name and a value (string or numeric) and creates a macro of that name that expands to the specified value (a numeric value provided as an argument is converted to a string). The macro is then available to any command syntax following the BEGIN PROGRAM-END PROGRAM block that created the macro. In the present example, this mechanism is used to create macros containing the lists of categorical and scale variables. For example, spss.SetMacroValue("!catvars", categoricalVars) creates a macro named !catvars that expands to the list of categorical variables in the active dataset.

- Tests are performed to determine if the list of categorical variables or the list of scale variables is empty before attempting to create associated macros. For example, if there are no categorical variables in the dataset, then `len(catlist)` will be 0 and interpreted as false for the purpose of evaluating an `if` statement.

- At the completion of the `BEGIN PROGRAM` block, the macro `!catvars` contains the list of categorical variables and `!scavars` contains the list of scale variables. If there are no categorical variables, then `!catvars` will not exist. Similarly, if there are no scale variables, then `!scavars` will not exist.

- The `FREQUENCIES` and `DESCRIPTIVES` commands that follow the program block reference the macros created in the block.

You can also pass information from command syntax to program blocks through the use of datafile attributes. For more information, see the topic "Retrieving Datafile Attributes" in Chapter 14 on p. 223.

Nested Program Blocks

From within Python, you can submit command syntax containing a `BEGIN PROGRAM` block, thus allowing you to nest program blocks. Nested program blocks are not restricted to being Python program blocks, but you can submit a nested block only from Python. For example, you can nest an R program block in a Python program block, but you cannot nest a Python program block in an R program block. You can nest program blocks within nested program blocks, up to five levels of nesting.

One approach for nesting program blocks is to include the nested block in a separate command syntax file and submit an `INSERT` command to read in the block.

Example

```
BEGIN PROGRAM.
import spss
spss.Submit("INSERT FILE='/myprograms/nested_block.sps'.")
END PROGRAM.
```

The file */myprograms/nested_block.sps* would contain a `BEGIN PROGRAM` block, as in:

```
BEGIN PROGRAM PYTHON.
import spss
<Python code>
END PROGRAM.
```

The above approach, however, is not supported if you need to embed the code that nests a program block in a Python module that can be imported. For example, you would like to import a Python module that uses the `INSERT` command to insert a file containing a program block. This is not supported. If you wish to encapsulate nested program blocks in a Python module that can be imported, then embed the nesting code in a user-defined function as shown in the following example.

Example

```
BEGIN PROGRAM.
import spss, myfuncs
myfuncs.demo()
END PROGRAM.
```

- `myfuncs` is a user-defined Python module containing the function (`demo`) that will submit the nested program block.

  A Python module is simply a text file containing Python definitions and statements. You can create a module with a Python IDE, or with any text editor, by saving a file with an extension of *.py*. The name of the file, without the *.py* extension, is then the name of the module.

- The `import` statement includes `myfuncs` so that it is loaded along with the `spss` module. To be sure that Python can find your module, you may want to save it to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

- The code `myfuncs.demo()` calls the function `demo` in the `myfuncs` module.

Following is a sample of the contents of `myfuncs`.

```
import spss
def demo():
   spss.Submit("""
BEGIN PROGRAM PYTHON.
<Python code>
END PROGRAM.""")
```

- The sample `myfuncs` module includes an `import spss` statement. This is necessary since a function in the module makes use of a function from the `spss` module—specifically, the `Submit` function.

- The nested program block is contained within a Python triple-quoted string. Triple-quoted strings allow you to specify a block of commands on multiple lines, resembling the way you might normally write command syntax. For more information, see the topic "Creating Blocks of Command Syntax within Program Blocks" in Chapter 13 on p. 199.

- Notice that `spss.Submit` is indented but the `BEGIN PROGRAM` block is not. Python statements, such as `spss.Submit`, that form the body of a user-defined Python function must be indented. The `BEGIN PROGRAM` block is passed as a string argument to the `Submit` function and is processed by IBM® SPSS® Statistics as a block of Python statements. Python statements are not indented unless they are part of a group of statements, as in a function or class definition, a conditional expression, or a looping structure.

Variable Scope

Python variables specified in a nested program block are local to that block unless they are specified as global variables. In addition, Python variables specified in a program block that invokes a nested block can be read, but not modified, in the nested block. Consider the following simple program block:

```
BEGIN PROGRAM.
import spss
var1 = 0
spss.Submit("INSERT FILE='/myprograms/nested_block.sps'.")
print "Value of var1 from root block after calling nested block: ", var1
try:
  print "Value of var2 from root block: ", var2
except:
  print "Can't read var2 from root block"
END PROGRAM.
```

And the associated nested block (contained in the file */myprograms/nested_block.sps*):

```
BEGIN PROGRAM.
print "Value of var1 from nested block: ", var1
var2 = 1
var1 = 1
END PROGRAM.
```

The result of running the first program block is:

```
Value of var1 from nested block:  0
Value of var1 from root block after calling nested block:  0
Value of var2 from root block:  Can't read var2 from root block
```

- The first line of the result shows that the nested block can read the value of a variable, *var1*, set in the calling block.

- The second line of the result shows that a nested block cannot modify the value of a variable set in a calling block. In other words, *var1* in the nested block is local to that block and has no relation to the variable *var1* in the calling block. If a nested block really needs to modify a variable in the calling block, that variable must be declared global at the start of the nested block.

- The third line of the result shows that a calling block cannot read the value of a variable set in a nested block.

Handling Errors

Errors detected during execution generate exceptions in Python. Aside from exceptions caught by the Python interpreter, the `spss` module catches three types of errors and raises an associated exception: an error in executing a syntax command submitted via the `Submit` function, an error in calling a function in the `spss` module (such as using a string argument where an integer is required), and an error in executing a function in the `spss` module (such as providing an index beyond the range of variables in the active dataset).

   Whenever there is a possibility of generating an error, it's best to include the associated code in a Python `try` clause, followed by an `except` or `finally` clause that initiates the appropriate action.

Chapter 12

Chapter 12

Example

Suppose you need to find all *.sav* files, in a directory, that contain a particular variable. You search for filenames that end in *.sav* and attempt to obtain the list of variables in each. There's no guarantee, though, that a file with a name ending in *.sav* is actually a data file in IBM® SPSS® Statistics format, so your attempt to obtain variable information may fail. Here's a code sample that handles this, assuming that you already have the list of files that end with *.sav*:

```
for fname in savfilelist:
   try:
      spss.Submit("get file='" + dirname + "/" + fname + "'.")
      <test if variable is in file and print file name if it is>
   except:
      pass
```

- The first statement in the `try` clause submits a `GET` command to attempt to open a file from the list of those that end with *.sav*.
- If the file can be opened, control passes to the remainder of the statements in the `try` clause to test if the file contains the variable and print the filename if it does.
- If the file cannot be opened, an exception is raised and control passes to the `except` clause. Since the file isn't a data file in SPSS Statistics format, there's no action to take, so the `except` clause just contains a `pass` statement.

In addition to generating exceptions for particular scenarios, the `spss` module provides functions to obtain information about the errors that gave rise to the exceptions. The function `GetLastErrorLevel` returns the error code for the most recent error, and `GetLastErrorMessage` returns text associated with the error code.

Working with Multiple Versions of IBM SPSS Statistics

Beginning with version 15.0 of IBM® SPSS® Statistics, multiple versions of the IBM® SPSS® Statistics - Integration Plug-In for Python can be used on the same machine, each associated with a major version of SPSS Statistics such as 18.0 or 19. For information on working with multiple versions, see the topic on the Integration Plug-In for Python in the SPSS Statistics Help system.

Creating a Graphical User Interface

The Custom Dialog Builder, introduced in release 17.0, allows you to create a user interface for your Python code, whether it's wrapped in an extension command (a custom command implemented in Python or R) or simply contained in a `BEGIN PROGRAM-END PROGRAM` block. You can also create user interfaces with one of a variety of toolkits available with the Python programming language—such as the `Tkinter` module (provided with Python), or wxPython, which is a popular, downloadable toolkit. Unless you need to drive the user interface from an external Python process or interact with the user at some intermediate point during execution, use the Custom Dialog Builder. For an example of this approach, see "Creating and Deploying Custom Dialogs for Extension Commands" on p. 395.

The remainder of this section contains examples of user interface components built with the wxPython toolkit, which is freely available from *http://www.wxpython.org/.* The examples are intended to display the ease with which you can create some of the more common user interface components that might be useful in Python programs that interact with IBM® SPSS® Statistics. Although the examples demonstrate user interface components within Python programs, the same toolkits can be used in Python scripts.

Example: Simple Message Box

- In this example, we'll create a dialog box that prompts for a Yes or No response. This is done using the `MessageDialog` class from the `wx` module.

```
*python_simple_message_box.sps.
BEGIN PROGRAM.
import wx
app = wx.PySimpleApp()
dlg = wx.MessageDialog(None, "Ok to reformat hard disk?",
                       caption="Important Question",
                       style=wx.YES_NO | wx.NO_DEFAULT | wx.ICON_QUESTION)
ret = dlg.ShowModal()
if ret == wx.ID_YES:
    # put Yes action code here
    print "You said yes"
else:
    # put No action code here
    print "You said No"

dlg.Destroy()
app.Destroy()
END PROGRAM.
```

Figure 12-4
Simple message box
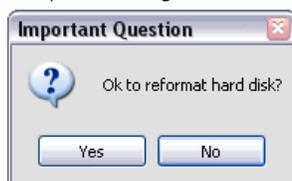
- Once you've installed wxPython, you use it by including an `import` statement for the `wx` module, as in `import wx`. You then create an instance of a wxPython application object, which is responsible for initializing the underlying GUI toolkit and managing the events that comprise the interaction with the user. For the simple example shown here, the `PySimpleApp` class is sufficient.

- The first argument to the `MessageDialog` class specifies a parent window or *None* if the dialog box is top-level, as in this example. The second argument specifies the message to be displayed. The optional argument *caption* specifies the text to display in the title bar of the dialog box. The optional argument *style* specifies the icons and buttons to be shown: `wx.YES_NO` specifies the Yes and No buttons, `wx.NO_DEFAULT` specifies that the default button is No, and `wx.ICON_QUESTION` specifies the question mark icon.

- The `ShowModal` method of the `MessageDialog` instance is used to display the dialog box and returns the button clicked by the user—`wx.ID_YES` or `wx.ID_NO`.

- You call the `Destroy` method when you're done with an instance of a wxPython class. In this example, you call the `Destroy` method for the instance of the `PySimpleApp` class and the instance of the `MessageDialog` class.

Example: Simple File Chooser

In this example, we'll create a dialog box that allows a user to select a file, and we'll include a file type filter for SPSS Statistics *.sav* files in the dialog box. This is done using the `FileDialog` class from the `wx` module.

```
*python_simple_file_chooser.sps.
BEGIN PROGRAM.
import wx, os, spss
app = wx.PySimpleApp()
fileWildcard = "sav files (*.sav)|*.sav|" \
               "All files (*.*)|*.*"

dlg = wx.FileDialog(None,
                    message="Choose a data file",
                    defaultDir=os.getcwd(),
                    defaultFile="",
                    wildcard=fileWildcard,
                    style=wx.OPEN)

if dlg.ShowModal() == wx.ID_OK:
    filespec = dlg.GetPath()
else:
    filespec = None

dlg.Destroy()
app.Destroy()

if filespec:
    spss.Submit("GET FILE='" + str(filespec) + "'.")

END PROGRAM.
```

Figure 12-5
Simple file chooser dialog box
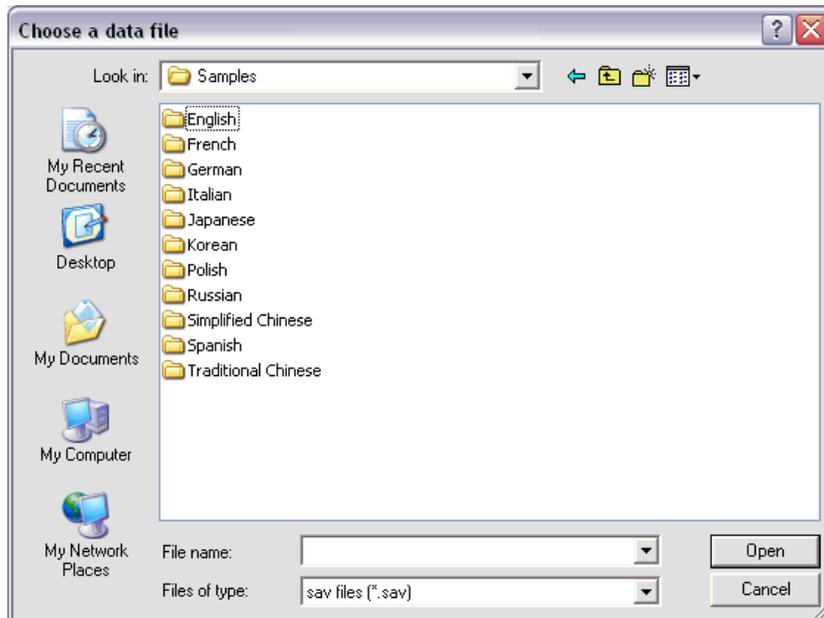
- This example makes use of the `getcwd` function from the `os` module (provided with Python), so the `import` statement includes it as well as the `wx` module for wxPython and the `spss` module.

- The first argument to the `FileDialog` class specifies a parent window or *None* if the dialog box is top-level, as in this example. The optional argument *message* specifies the text to display in the title bar of the dialog box. The optional argument *defaultDir* specifies the default directory, which is set to the current working directory, using the `getcwd` function from the `os` module. The optional argument *defaultFile* specifies a file to be selected when the dialog box opens. An empty string, as used here, specifies that nothing is selected when the dialog box opens. The optional argument *wildcard* specifies the file type filters available to limit the list of files displayed. The argument specifies both the wildcard setting and the label associated with it in the Files of type drop-down list. In this example, the filter `*.sav` is labeled as sav files (*.sav), and the filter `*.*` is labeled as All files (*.*). The optional argument *style* specifies the style of the dialog box. `wx.OPEN` specifies the style used for a File > Open dialog box.

- The `ShowModal` method of the `FileDialog` instance is used to display the dialog box and returns the button clicked by the user—`wx.ID_OK` or `wx.ID_CANCEL`.

- The `GetPath` method of the `FileDialog` instance returns the full path of the selected file.

- If the user clicked OK and a non-empty file path was retrieved from the dialog box, then submit a GET command to SPSS Statistics to open the file.

Example: Simple Multi-Variable Chooser

In this example, we'll create a dialog box for selecting multiple items and populate it with the scale variables from a selected dataset. This is done using the `MultiChoiceDialog` class from the `wx` module.

```
*python_simple_multivariable_chooser.sps.
BEGIN PROGRAM.
import wx, spss, spssaux

spssaux.OpenDataFile("/examples/data/Employee data.sav")
vardict = spssaux.VariableDict(variableLevel=['scale'])
choicelist = vardict.variables
if choicelist:
    app = wx.PySimpleApp()
    dlg = wx.MultiChoiceDialog(None,
            "Select one or more variables\nfor analysis",
            "Descriptive Statistics",
            choices=choicelist)
    if dlg.ShowModal() == wx.ID_OK:
        vars = dlg.GetSelections()
    else:
        vars = None

    dlg.Destroy()
    app.Destroy()

    if vars:
        varlist = [choicelist[i] for i in vars]
        spss.Submit("DESCRIPTIVES " + " ".join(varlist))
END PROGRAM.
```

Figure 12-6
Simple multi-variable chooser dialog box
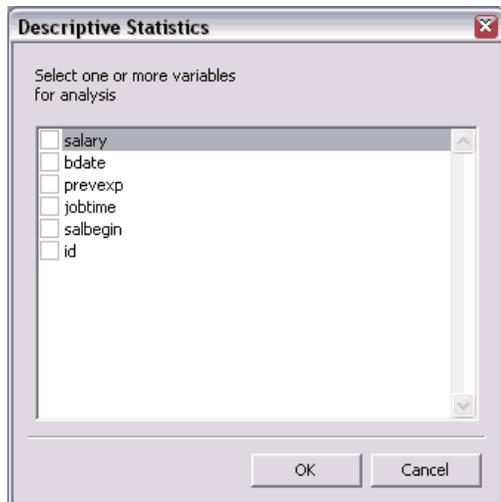
- This example makes use of the `spssaux` module—a supplementary module that is installed with the IBM® SPSS® Statistics - Integration Plug-In for Python—so the `import` statement includes it in addition to the `wx` module for wxPython and the `spss` module.

- The `OpenDataFile` function from the `spssaux` module opens an external SPSS Statistics data file. The argument is the file path specified as a string.

- `VariableDict` is a class in the `spssaux` module that provides an object-oriented approach to obtaining information about the variables in the active dataset. The class allows you to specify a subset of variables whose information is then accessible through the methods and properties of the class. You can specify variables by name, type (string or numeric), or measurement level, as done here for scale variables. For more information, see the topic "Getting Started with the VariableDict Class" in Chapter 14 on p. 225.

- The `variables` property of a `VariableDict` instance provides a list of the names of the variables described by the instance. In this case, the instance describes the scale variables in *Employee data.sav*.

- The first argument to the `MultiChoiceDialog` class specifies a parent window or *None* if the dialog box is top-level, as in this example. The second argument specifies the message text to display in the dialog box. Note that the Python escape sequence for a linefeed, `"\n"`, is used. The third argument specifies the text to display in the title bar of the dialog box. The optional argument *choices* specifies the selectable items in the dialog box—in this case, the set of scale variables in *Employee data.sav*.

- The `ShowModal` method of the `MultiChoiceDialog` instance is used to display the dialog box and returns the button clicked by the user—`wx.ID_OK` or `wx.ID_CANCEL`.

- If the user clicked OK, then get the selected items using the `GetSelections` method of the `MultiChoiceDialog` instance. The method returns the indices of the selected items, starting with the index 0 for the first item in the list.

- *varlist* is a Python list of names of the selected variables and is constructed from the index list returned from `GetSelections`. If you are not familiar with the method used here to create a list, see the section "List Comprehensions" in the Python tutorial, available at *http://docs.python.org/tut/tut.html*.

- The `DESCRIPTIVES` procedure is run for the selected variables using the `Submit` function from the `spss` module. Syntax commands must be specified as strings, so the Python string method `join` is used to construct a string of names from the Python list *varlist*. For more information, see the topic "Dynamically Creating Command Syntax" on p. 182.

Supplementary Python Modules for Use with IBM SPSS Statistics

The `spss` module, included with the IBM® SPSS® Statistics - Integration Plug-In for Python, provides the base functionality for writing Python programs. A number of supplementary Python modules that build on the functionality provided by the `spss` module are available for download from Developer Central at *http://www.spss.com/devcentral/*.

Along with many of the modules, you'll find command syntax (*.sps*) files that provide examples of using the module functions in `BEGIN PROGRAM-END PROGRAM` blocks. The modules are provided in the form of source (*.py*) files, so they can be customized, studied as a learning resource, or used as a foundation for creating your own modules. Instructions for downloading and using the modules are provided on Developer Central.

You can also wrap these, or any other, Python modules in IBM® SPSS® Statistics command syntax by creating an extension command. This allows you to share external functions with users of SPSS Statistics command syntax. For more information, see the topic "Extension Commands" in Chapter 31 on p. 381.

Note: For SPSS Statistics version 16.0 and above, the supplementary modules *spssaux*, *spssdata*, *namedtuple*, and *extension* are installed with the Integration Plug-In for Python.

Getting Help

Help with using the features of the IBM® SPSS® Statistics - Integration Plug-In for Python is available from a number of resources:

- Complete documentation for all of the functions available with the Integration Plug-In for Python is available in the IBM® SPSS® Statistics Help system, under Integration Plug-In for Python Help. The documentation is also available as two PDF's, accessed from Help>Programmability>Python Plug-in and Help>Programmability>Scripting, once the Integration Plug-In for Python is installed. The former describes the interface exposed by the `spss` module, and the latter describes the interface exposed by the `SpssClient` module.

- Once the associated module has been imported, an online description of a particular function, class, method, or module is available using the Python `help` function. For example, to obtain a description of the `Submit` function in the `spss` module, use `help(spss.Submit)` after `import spss`. To display information for all of the objects in a module, use `help(module name)`, as in `help(spss)`. When the `help` function is used within a `BEGIN PROGRAM-END PROGRAM` block, the description is displayed in a log item in the Viewer if a Viewer is available. *Note*: Help for the `SpssClient` module is not available from the Python `help` function.

- The `spss` module and the supplementary modules are provided as source code. Once you're familiar with the Python programming language, you may find that consulting the source code is the best way to locate the information you need, such as which functions or classes are included with a module or what arguments are needed for a given function.

- Detailed command syntax reference information for `BEGIN PROGRAM-END PROGRAM` can be found in the SPSS Statistics Help system.

- For help in getting started with the Python programming language, see the Python tutorial, available at *http://docs.python.org/tut/tut.html*.

- You can also post questions about using Python with SPSS Statistics to the Python Forum on Developer Central.

Best Practices

This section provides advice for dealing with some common issues and introduces a number of features that will help you with writing Python code within IBM® SPSS® Statistics.

Creating Blocks of Command Syntax within Program Blocks

Often, it is desirable to specify blocks of syntax commands on multiple lines within a program block, which more closely resembles the way you might normally write command syntax. This is best accomplished using the Python triple-quoted string convention, where line breaks are allowed and retained as long as they occur within a string enclosed in a set of triple single or double quotes.

Example

```
*python_triple_quoted_string.sps.
BEGIN PROGRAM.
import spss
spss.Submit(r"""
GET FILE='/examples/data/Employee data.sav'.
SORT CASES BY gender.
SPLIT FILE
  LAYERED BY gender.
DESCRIPTIVES
  VARIABLES=salary salbegin jobtime prevexp
  /STATISTICS=MEAN STDDEV MIN MAX.
SPLIT FILE OFF.
""")
END PROGRAM.
```

- The triple double quotes enclose a block of command syntax that is submitted for processing, retaining the line breaks. You can use either triple single quotes or triple double quotes, but you must use the same type (single or double) on both sides of the command syntax block.

- Notice that the triple-quoted expression is prefixed with the letter r. The r prefix to a string specifies Python's raw mode. This allows you to use the single backslash (\) notation for file paths on Windows. That said, it is a good practice to use forward slashes (/) in file paths on Windows, since you may at times forget to use raw mode, and IBM® SPSS® Statistics accepts a forward slash for any backslash in a file specification. For more information, see the topic "Using Raw Strings in Python" on p. 202.

- In the unusual case that the command syntax block contains a triple quote, be sure that it's not the same type as the type you are using to enclose the block; otherwise, Python will treat it as the end of the block.

Wrapping blocks of command syntax in triple quotes within a BEGIN PROGRAM-END PROGRAM block allows you to easily convert a command syntax job to a Python job. For more information, see the topic "Migrating Command Syntax Jobs to Python" in Chapter 21 on p. 325.

Dynamically Specifying Command Syntax Using String Substitution

Most often, you embed command syntax within program blocks so that you can dynamically specify pieces of the syntax, such as variable names. This is best done using string substitution in Python. For example, say you want to create a split file on a particular variable whose name is determined dynamically. Omitting the code for determining the particular variable, a code sample to accomplish this might look like:

```
spss.Submit(r"""
SORT CASES BY %s.
SPLIT FILE
 LAYERED BY %s.
""" %(splitVar,splitVar))
```

Within a string (in this case, a triple-quoted string), `%s` marks the points at which a string value is to be inserted. The particular value to insert is taken from the `%` expression that follows the string; in this case, `%(splitVar,splitVar)`. The value of the first item in the `%` expression replaces the first occurrence of `%s`, the value of the second item replaces the second occurrence of `%s`, and so on. Let's say that the variable *splitVar* has the value `"gender"`. The command string submitted to IBM® SPSS® Statistics would be:

```
SORT CASES BY gender.
SPLIT FILE
 LAYERED BY gender.
```

Note: Python will convert the values supplied in the `%()` expression to the specified format type (the `s` in `%s` specifies a string) if possible and will raise an exception otherwise.

The above approach can become cumbersome once you have to substitute more than a few values into a string expression, since you have to keep track of which occurrence of `%s` goes with which value in the `%` expression. Using a Python dictionary affords an alternative to providing a sequential list of substitution values.

Example

Let's say you have many datasets, each consisting of employee data for a particular department of a large company. Each dataset contains a variable for current salary, a variable for starting salary, and a variable for the number of months since hire. For each dataset, you'd like to compute the average annual percentage increase in salary and sort by that value to identify employees who may be undercompensated. The problem is that the names of the variables you need are not constant across the datasets, while the variable labels are constant. Current salary is always labeled *Current Salary*, starting salary is always labeled *Beginning Salary*, and months since hire is always labeled *Months since Hire*. For simplicity, the following program block performs the calculation for a single file; however, everything other than the file retrieval command is completely general.

```
*python_string_substitution.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/employee data.sav'.")
for i in range(spss.GetVariableCount()):
    label = spss.GetVariableLabel(i).lower()
    if label=='current salary':
        cursal=spss.GetVariableName(i)
    elif label=='beginning salary':
        begsal=spss.GetVariableName(i)
    elif label == 'months since hire':
        mos=spss.GetVariableName(i)
spss.Submit(r"""
SELECT IF %(mos)s > 12.
COMPUTE AVG_PCT_CHANGE =
   100*(%(cur)s - %(beg)s)/(%(beg)s * TRUNC(%(mos)s/12)).
SORT CASES BY AVG_PCT_CHANGE (A).
""" %{'cur':cursal,'beg':begsal,'mos':mos})
END PROGRAM.
```

- First, loop through the variables in the active dataset, setting the Python variable *cursal* to the name of the variable for current salary; *begsal*, to the name of the variable for beginning salary; and *mos*, to the name of the variable for months since hire.

- The Submit function contains a triple-quoted string that resolves to the command syntax needed to perform the calculation. The expression

  ```
  %{'cur':cursal,'beg':begsal,'mos':mos}
  ```

  following the triple quotes defines a Python dictionary that is used to specify the string substitution. A Python dictionary consists of a set of keys, each of which has an associated value that can be accessed simply by specifying the key. In the current example, the dictionary has the keys *cur*, *beg*, and *mos* associated with the values of the variables *cursal*, *begsal*, and *mos*, respectively. Instead of using %s to mark insertion points, you use %(key)s. For example, you insert %(beg)s wherever you want the value associated with the key *beg*—in other words, wherever you want the value of *begsal*.

For the dataset used in this example, *cursal* has the value 'salary', *begsal* has the value 'salbegin', and *mos* has the value 'jobtime'. After the string substitution, the triple-quoted expression resolves to the following block of command syntax:

```
SELECT IF jobtime > 12.
COMPUTE AVG_PCT_CHANGE =
   100*(salary - salbegin)/(salbegin * TRUNC(jobtime/12)).
SORT CASES BY AVG_PCT_CHANGE (A).
```

Of course, if any of the variables *cursal*, *begsal*, or *mos* is undefined at the time of the string substitution, then an exception will occur. It is good practice to add robustness to your programs to try to ensure that unhandled exceptions do not occur. For instance, in the present example, you could wrap the spss.Submit function in a try/except block. For more information, see the topic "Using Exception Handling in Python" on p. 207.

You can simplify the statement for defining the dictionary for string substitution by using the locals function. It produces a dictionary whose keys are the names of the local variables and whose associated values are the current values of those variables. For example,

```
splitVar = 'gender'
spss.Submit(r"""
SORT CASES BY %(splitVar)s.
SPLIT FILE
 LAYERED BY %(splitVar)s.
""" %locals())
```

splitVar is a local variable; thus, the dictionary created by the `locals` function contains the key *splitVar* with the value `'gender'`. The string `'gender'` is then substituted for every occurrence of `%(splitVar)s` in the triple-quoted string.

String substitution is not limited to triple-quoted strings. For example, the code sample

```
spss.Submit("SORT CASES BY %s." %(sortkey))
```

runs a `SORT CASES` command using a single variable whose name is the value of the Python variable *sortkey*.

Using Raw Strings in Python

Python reserves certain combinations of characters beginning with a backslash (\) as escape sequences. For example, `"\n"` is the escape sequence for a linefeed and `"\t"` is the escape sequence for a horizontal tab. This is potentially problematic when specifying strings, such as file paths on Windows or regular expressions, that contain these sequences. For example, the Windows path `"c:\temp\myfile.sav"` would be interpreted by Python as `"c:"`, followed by a tab, followed by `"emp\myfile.sav"`, which is probably not what you intended.

The problem of backslashes is best solved by using raw strings in Python. When you preface a string with an `r` or `R`, Python treats all backslashes in the string as the backslash character and not as the start of an escape sequence. The only caveat is that the last character in the string cannot be a backslash. For example, `filestring = r"c:\temp\myfile.sav"` sets the variable *filestring* to the string `"c:\temp\myfile.sav"`. Because a raw string was specified, the sequence `"\t"` is treated as a backslash character followed by the letter `t`.

You can preface any string, including triple-quoted strings, with `r` or `R` to indicate that it's a raw string. That is a good practice to employ, since then you don't have to worry about any escape sequences that might unintentionally exist in a triple-quoted string containing a block of command syntax. IBM® SPSS® Statistics also accepts a forward slash (/) for any backslash in a file specification. This provides an alternative to using raw strings for file specifications on Windows.

It is also a good idea to use raw strings for regular expressions. Regular expressions define patterns of characters and enable complex string searches. For example, using a regular expression, you could search for all variables in the active dataset whose names end in a digit. For more information, see the topic "Using Regular Expressions to Select Variables" in Chapter 14 on p. 230.

Displaying Command Syntax Generated by Program Blocks

For debugging purposes, it is convenient to see the completed syntax passed to IBM® SPSS® Statistics by any calls to the `Submit` function in the `spss` module. This is enabled through command syntax with `SET PRINTBACK ON MPRINT ON`.

Example

```
SET PRINTBACK ON MPRINT ON.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
varName = spss.GetVariableName(spss.GetVariableCount()-1)
spss.Submit("FREQUENCIES /VARIABLES=" + varName + ".")
END PROGRAM.
```

The generated command syntax is displayed in a log item in the SPSS Statistics Viewer, if the Viewer is available, and shows the completed FREQUENCIES command as well as the GET command. For example, on Windows, assuming that you have copied the *examples* folder to the *C* drive, the result is:

```
300 M>  GET FILE='c:/examples/data/Employee data.sav'.
302 M>  FREQUENCIES /VARIABLES=minority.
```

Creating User-Defined Functions in Python

Undoubtedly, you will eventually want to create generalized code that is specified at run time by a set of parameters. If you simply want to generalize a block of command syntax so that the submitted syntax is specified by parameters at run time, then you can include your syntax in a BEGIN PROGRAM–END PROGRAM block and use string substitution to specify the parameters. For more information, see the topic "Dynamically Specifying Command Syntax Using String Substitution" on p. 200. If you want to create a general-purpose function that can be called like a subroutine, then you'll want to create a user-defined Python function. In fact, you may want to construct a library of your standard utility routines and always import it. The basic steps are:

- Encapsulate your code in a user-defined function. For a good introduction to user-defined functions in Python, see the section "Defining Functions" in the Python tutorial, available at *http://docs.python.org/tut/tut.html*.

- Include your function in a Python module on the Python search path. To be sure that Python can find your new module, you may want to save it to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

  A Python module is simply a text file containing Python definitions and statements. You can create a module with a Python IDE, or with any text editor, by saving a file with an extension of *.py*. The name of the file, without the *.py* extension, is then the name of the module. You can have many functions in a single module.

- Call your function from within a BEGIN PROGRAM–END PROGRAM block, passing specific parameter values to your function. The block should contain an import statement for the module containing the function (unless you've imported the module in a previous block).

Example

A common scenario is to run a particular block of command syntax only if a specific variable exists in the dataset. As an example, the following function checks for the existence of a specified variable in the active dataset or in an optionally specified file. It splits the dataset by the variable if the variable exists.

```
def SplitIfVarExists(varname, filespec=None):
    """Get the file, if specified, and check for the existence of
    the specified variable. Split the dataset by the variable if it exists.
    """

    if filespec:
        try:
            spss.Submit("GET FILE = '%s'." %(filespec))
        except:
            raise ValueError("Cannot open file: " + filespec)

    for i in range(spss.GetVariableCount()):
        name=spss.GetVariableName(i)
        if name.lower()==varname.lower():
            spss.Submit(r"""
            SORT CASES BY %s.
            SPLIT FILE
              LAYERED BY %s.
            """ %(name,name))
            break
```

- The `def` statement signals the beginning of a function named `SplitIfVarExists`. The colon at the end of the `def` statement is required.

- The function takes two parameters: *varname* specifies the variable to check, and *filespec* specifies an optional file to check for the existence of the variable. If *filespec* is omitted, the active dataset is used.

- The function combines Python code with command syntax, which is specified dynamically and submitted to IBM® SPSS® Statistics for processing. The values needed to specify the command syntax come from the function parameters and are inserted into the command string using string substitution. For more information, see the topic "Dynamically Specifying Command Syntax Using String Substitution" on p. 200.

You include the function in a module named `samplelib` and now want to use the function. For example, you are processing datasets containing employee records and want to split them by gender—if a gender variable exists—to obtain separate statistics for the two gender groups. We will assume that if a gender variable exists, it has the name *gender*, although it may be spelled in upper case or mixed case.

```
*python_split_if_var_exists.sps.
BEGIN PROGRAM.
import samplelib
samplelib.SplitIfVarExists('Gender','/examples/data/Employee data.sav')
END PROGRAM.
```

The BEGIN PROGRAM block starts with a statement to import the `samplelib` module, which contains the definition for the `SplitIfVarExists` function. The function is called with a variable name and a file specification.

Note: To run this program block, copy the module file *samplelib.py* from the */examples/python* folder, in the accompanying examples, to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

Creating a File Handle to the IBM SPSS Statistics Install Directory

Depending on how you work with IBM® SPSS® Statistics, it may be convenient to have easy access to files stored in the SPSS Statistics installation directory. This is best done by defining a file handle to the installation directory, using a function from the `spssaux` module.

Example

```
*python_handle_to_installdir.sps.
BEGIN PROGRAM.
import spss, spssaux
spssaux.GetSPSSInstallDir("SPSSDIR")
spss.Submit(r"GET FILE='SPSSDIR/Samples/Employee data.sav'.")
END PROGRAM.
```

- The program block imports and uses the `spssaux` module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python.

- The function `GetSPSSInstallDir`, from the `spssaux` module, takes a name as a parameter and creates a file handle of that name pointing to the location of the SPSS Statistics installation directory.

- The file handle is then available for use in any file specification that follows. Note that the command string for the `GET` command is a raw string; that is, it is prefaced by an `r`. It is a good practice to use raw strings for command strings that include file specifications so that you don't have to worry about unintentional escape sequences in Python. For more information, see the topic "Using Raw Strings in Python" on p. 202.

Choosing the Best Programming Technology

With the introduction of the IBM® SPSS® Statistics - Integration Plug-In for Python, you have a variety of programming technologies (in addition to command syntax) available for use with IBM® SPSS® Statistics—the macro language, Basic scripts, Python scripts, and Python programs. This section provides some advice on choosing the best technology for your task.

To start with, the ability to use Python programs to dynamically create and control command syntax renders SPSS Statistics macros mostly obsolete. Anything that can be done with a macro can be done with a Python user-defined function. For an example of an existing macro recoded in Python, see "Migrating Macros to Python" on p. 327. However, macros are still important for passing information from a `BEGIN PROGRAM` block so that it is available to command syntax outside of the block. For more information, see the section "Passing Values from a Program Block to Command Syntax" in "Mixing Command Syntax and Program Blocks" on p. 187.

Like Basic scripts, Python programs and Python scripts provide solutions for programming tasks that cannot readily be done with command syntax. In that sense, they are not intended as a replacement for the command syntax language. Using a Python program or a Python script is, however, almost always the preferred choice over using a Basic script. For one, Python is a much richer programming language and is supported by a vast open-source user community that actively extends the basic language with utilities such as IDEs, GUI toolkits, and packages for scientific computing. In addition, Python programs included in a command syntax job always run synchronously with the command syntax.

Consider using Python programs for these tasks you may have previously done with Basic scripts:

- Accessing the SPSS Statistics data dictionary

- Dynamically generating command syntax, such as when the particular variables in a dataset are not known in advance

- Manipulating files and directories

- Retrieving case data to accomplish a data-oriented task outside of command syntax

- Encapsulating a set of tasks in a program that accepts parameters and can be invoked from command syntax
- Using a custom dialog box to get input from the user and running user-selected tasks on a selected data file

Consider using Python scripts for these tasks you may have previously done with Basic scripts:

- Manipulating output that appears in the Viewer
- Automatically performing a set of actions when a particular kind of object is created in the Viewer (referred to as autoscripting)
- Driving SPSS Statistics dialog boxes when operating in distributed mode

Use Basic scripts and the OLE automation interfaces for:

- Integrating Viewer output into applications that support OLE automation, such as Microsoft PowerPoint
- Controlling SPSS Statistics from an application that supports Visual Basic, such as Microsoft Office or Visual Basic itself

In addition, consider using the IBM® SPSS® Statistics - Integration Plug-In for Microsoft® .NET to create .NET applications that can invoke and control the SPSS Statistics back end. And consider using the IBM® SPSS® Statistics - Integration Plug-In for R to create custom algorithms in R or to take advantage of the vast statistical libraries available with R. The .NET and R plug-ins are available from Developer Central.

Python Programs vs. Python Scripts

Python programs and Python scripts provide two distinct and mostly non-overlapping means for programming in Python within SPSS Statistics. Python programs operate on the SPSS Statistics processor and are designed for controlling the flow of a command syntax job, reading from and writing to datasets, creating new datasets, and creating custom procedures that generate their own pivot table output. Python scripts operate on user interface and output objects and are designed for customizing pivot tables, exporting items such as charts and tables, invoking SPSS Statistics dialog boxes, and managing connections to instances of SPSS Statistics Server. When a given task can be completed with either a Python program or a Python script, the Python program will always provide better performance and is preferred.

When working with Python programs and Python scripts, keep the following limitations in mind:

- Python programs cannot be run as autoscripts, so if you want to write an autoscript in Python, use a Python script.
- Python programs are not intended to be run from Utilities > Run Script within SPSS Statistics.

For detailed information on running Python programs and Python scripts as well as scenarios where one can invoke the other, see "Scripting Facility" and "Scripting with the Python Programming Language" in the SPSS Statistics Help system.

Using Exception Handling in Python

Errors that occur during execution are called **exceptions** in Python. Python includes constructs that allow you to handle exceptions so that you can decide whether execution should proceed or terminate. You can also raise your own exceptions, causing execution to terminate when a test expression indicates that the job is unlikely to complete in a meaningful way. And you can define your own exception classes, making it easy to package extra information with the exception and to test for exceptions by type. Exception handling is standard practice in Python and should be freely used when appropriate. For information on defining your own exception classes, see the Python tutorial, available at *http://docs.python.org/tut/tut.html*.

Raising an Exception to Terminate Execution

There are certainly cases where it is useful to create an exception in order to terminate execution. Some common examples include:

- A required argument is omitted in a function call.

- A required file, such as an auxiliary Python module, cannot be imported.

- A value passed to a function is of the wrong type, such as numeric instead of string.

Python allows you to terminate execution and to provide an informative error message indicating why execution is being terminated. We will illustrate this by testing whether a required argument is provided for a very simple user-defined function.

```
def ArgRequired(arg=None):
    if arg is None:
        raise ValueError, "You must specify a value."
    else:
        print "You entered:",arg
```

- The Python user-defined function `ArgRequired` has one argument with a default value of *None*.

- The `if` statement tests the value of *arg*. A value of *None* means that no value was provided. In this case, a `ValueError` exception is created with the `raise` statement and execution is terminated. The output includes the type of exception raised and any string provided on the `raise` statement. For this exception, the output includes the line:

    ```
    ValueError:  You must specify a value.
    ```

Handling an Exception without Terminating Execution

Sometimes exceptions reflect conditions that don't preclude the completion of a job. This can be the case when you are processing data that may contain invalid values or are attempting to open files that are either corrupt or have an invalid format. You would like to simply skip over the invalid data or file and continue to the next case or file. Python allows you to do this with the `try` and `except` statements.

As an example, let's suppose that you need to process all *.sav* files in a particular directory. You build a list of them and loop through the list, attempting to open each one. There's no guarantee, however, that a file with a name ending in *.sav* is actually a data file in IBM® SPSS® Statistics

format, so your attempt to open any given file may fail, generating an exception. Following is a code sample that handles this:

```
for fname in savfilelist:
   try:
      spss.Submit("get file='" + dirname + "/" + fname + "'.")
      <do something with the file>
   except:
      pass
```

■ The first statement in the `try` clause submits a `GET` command to attempt to open a file in the list of those that end with *.sav*.

■ If the file can be opened, control passes to the remainder of the statements in the `try` clause that do the necessary processing.

■ If the file can't be opened, an exception is raised and control passes to the `except` clause. Since the file isn't a data file in SPSS Statistics format, there's no action to take. Thus, the `except` clause contains only a `pass` statement. Execution of the loop continues to the next file in the list.

User-Defined Functions That Return Error Codes

Functions in the `spss` module raise exceptions for errors encountered during execution and make the associated error codes available. Perhaps you are dynamically building command syntax to be passed to the `Submit` function, and because there are cases that can't be controlled for, the command syntax fails during execution. And perhaps this happens within the context of a large production job, where you would simply like to flag the problem and continue with the job. Let's further suppose that you have a Python user-defined function that builds the command syntax and calls the `Submit` function. Following is an outline of how to handle the error, extract the error code, and provide it as part of the returned value from the user-defined function.

```
def BuildSyntax(args):

   <Build the command syntax and store it to cmd.
    Store information about this run to id.>
   try:
      spss.Submit(cmd)
   except:
      pass
   return (id,spss.GetLastErrorLevel())
```

■ The `Submit` function is part of a `try` clause. If execution of the command syntax fails, control passes to the `except` clause.

■ In the event of an exception, you should exit the function, returning information that can be logged. The `except` clause is used only to prevent the exception from terminating execution; thus, it contains only a `pass` statement.

■ The function returns a two-tuple, consisting of the value of *id* and the maximum SPSS Statistics error level for the submitted commands. Using a tuple allows you to return the error code separately from any other values that the function normally returns.

The call to `BuildSyntax` might look something like the following:

```
id_info, errcode=BuildSyntax(args)
if errcode > 2:
   <log an error>
```

■ On return, *id_info* will contain the value of *id* and *errcode* will contain the value returned by `spss.GetLastErrorLevel()`.

Differences from Error Handling in Basic

For users familiar with programming in Basic or Visual Basic, it's worth pointing out that Python doesn't have the equivalent of `On Error Resume Next`. You can certainly resume execution after an error by handling it with a `try/except` block, as in:

```
try:
   <statement>
except:
   pass
```

But this has to be done for each statement where an error might occur.

Debugging Python Programs

Two modes of operation are available for running Python programs: enclosing your code in `BEGIN PROGRAM-END PROGRAM` blocks as part of a command syntax job or running it from a Python IDE (Integrated Development Environment). Both modes have features that facilitate debugging.

Using a Python IDE

When you develop your code in a Python IDE, you can test one or many lines of code in the IDE interactive window and see immediate results, which is particularly useful if you are new to Python and are still trying to learn the language. And the Python `print` statement allows you to inspect the value of a variable or the result of an expression.

Most Python IDEs also provide debuggers that allow you to set breakpoints, step through code line by line, and inspect variable values and object properties. Python debuggers are powerful tools and have a nontrivial learning curve. If you're new to Python and don't have a lot of experience working with debuggers, you can do pretty well with `print` statements in the interactive window of an IDE, but for serious use, it is well worth mastering a debugger.

To get started with the Python IDE approach, see "Running Your Code from a Python IDE" on p. 176. You can use the IDLE IDE, which is provided with Python, or you can use one of several third-party Python IDEs, a number of which are free. For a link to information and reviews on available Python IDEs, see the topic "Getting Started with Python" at *http://www.python.org/about/gettingstarted/*.

Benefits of Running Code from Program Blocks

Once you've installed the IBM® SPSS® Statistics - Integration Plug-In for Python, you can start developing Python code within `BEGIN PROGRAM-END PROGRAM` blocks in a command syntax job. Nothing else is required.

One of the benefits of running your code from a `BEGIN PROGRAM-END PROGRAM` block is that output is directed to the Viewer if it is available. Although IBM® SPSS® Statistics output is also available when you are working with a Python IDE, the output in that case is displayed in text form, and charts are not included.

From a program block, you can display the value of a Python variable or the result of a Python expression by including a Python `print` statement in the block. The `print` statement is executed when you run command syntax that includes the program block, and the result is displayed in a log item in the SPSS Statistics Viewer.

Another feature of running Python code from a program block is that Python variables persist from one program block to another. This allows you to inspect variable values as they existed at the end of a program block, as shown in the following:

```
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
ordlist=[]
for i in range(spss.GetVariableCount()):
   if spss.GetVariableMeasurementLevel(i) in ['ordinal']:
      ordlist.append(spss.GetVariableName(i))
cmd="DESCRIPTIVES VARIABLES=%s." %(ordlist)
spss.Submit(cmd)
END PROGRAM.
```

The program block is supposed to create a list of ordinal variables in *Employee data.sav* but will generate an error in its current form, which suggests that there is a problem with the submitted `DESCRIPTIVES` command. If you didn't spot the problem right away, you would probably be inclined to check the value of *cmd*, the string that specifies the `DESCRIPTIVES` command. To do this, you could add a `print cmd` statement after the assignment of *cmd*, or you could simply create an entirely new program block to check the value of *cmd*. The latter approach doesn't require that you rerun your code. It also has the advantage of keeping out of your source code `print` statements that are used only for debugging the source code. The additional program block might be:

```
BEGIN PROGRAM.
print cmd
END PROGRAM.
```

Running this program block after the original block results in the output:

```
DESCRIPTIVES VARIABLES=['educ', 'jobcat', 'minority'].
```

It is displayed in a log item in the Viewer. You now see the problem is that you provided a Python list for the SPSS Statistics variable list, when what you really wanted was a string containing the list items, as in:

```
DESCRIPTIVES VARIABLES=educ jobcat minority.
```

The problem is solved by using the Python string method `join`, which creates a string from a list by concatenating the elements of the list, using a specified string as the separator between elements. In this case, we want each element to be separated by a single space. The correct specification for *cmd* is:

```
cmd="DESCRIPTIVES VARIABLES=%s." %(" ".join(ordlist))
```

In addition to the above remarks, keep the following general considerations in mind:

- Unit test Python user-defined functions and the Python code included in `BEGIN PROGRAM-END PROGRAM` blocks, and try to keep functions and program blocks small so they can be more easily tested.

- Note that many errors that would be caught at compile time in a more traditional, less dynamic language, will be caught at run time in Python—for example, an undefined variable.

Working with Dictionary Information

The spss module provides a number of functions for retrieving dictionary information from the active dataset. It includes functions to retrieve:

- The number of variables in the active dataset
- The weight variable, if any
- Variable names
- Variable labels
- Display formats of variables
- Measurement levels of variables
- The variable type (numeric or string)
- The names of any split variables
- Missing values
- Value labels
- Custom variable attributes
- Datafile attributes
- Multiple response sets

Functions that retrieve information for a specified variable use the position of the variable in the dataset as the identifier, starting with 0 for the first variable in file order. This is referred to as the **index value** of the variable.

Example

The function to retrieve the name of a particular variable is GetVariableName. It requires a single argument, which is the index value of the variable to retrieve. This simple example creates a dataset with two variables and uses GetVariableName to retrieve their names.

```
DATA LIST FREE /var1 var2.
BEGIN DATA
1 2 3 4
END DATA.
BEGIN PROGRAM.
import spss
print "The name of the first variable in file order is (var1): " \
      + spss.GetVariableName(0)
print "The name of the second variable in file order is (var2): " \
      + spss.GetVariableName(1)
END PROGRAM.
```

Example

Often, you'll want to search through all of the variables in the active dataset to find those with a particular set of properties. The function GetVariableCount returns the number of variables in the active dataset, allowing you to loop through all of the variables, as shown in the following example:

```
DATA LIST FREE /var1 var2 var3 var4.
BEGIN DATA
14 25 37 54
END DATA.
BEGIN PROGRAM.
import spss
for i in range(spss.GetVariableCount()):
   print spss.GetVariableName(i)
END PROGRAM.
```

- The Python function range creates a list of integers from 0 to one less than its argument. The sample dataset used in this example has four variables, so the list is [0,1,2,3]. The for loop then iterates over these four values.

- The function GetVariableCount doesn't take any arguments, but Python still requires you to include a pair of parentheses on the function call, as in: GetVariableCount().

In addition to specific functions for retrieving dictionary information, the complete set of dictionary information for the active dataset is available from an in-memory XML representation of the dictionary created by the CreateXPathDictionary function. For an example of this approach, see "Identifying Variables without Value Labels" on p. 220.

Summarizing Variables by Measurement Level

When doing exploratory analysis on a dataset, it can be useful to run FREQUENCIES for the categorical variables and DESCRIPTIVES for the scale variables. This process can be automated by using the GetVariableMeasurementLevel function from the spss module to build separate lists of the categorical and scale variables. You can then submit a FREQUENCIES command for the list of categorical variables and a DESCRIPTIVES command for the list of scale variables, as shown in the following example:

```
*python_summarize_by_level.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
catlist=[]
scalist=[]
for i in range(spss.GetVariableCount()):
   varName=spss.GetVariableName(i)
   if spss.GetVariableMeasurementLevel(i) in ['nominal', 'ordinal']:
      catlist.append(varName)
   else:
      scalist.append(varName)
if len(catlist):
   categoricalVars = " ".join(catlist)
   spss.Submit("FREQUENCIES " + categoricalVars + ".")
if len(scalist):
   scaleVars = " ".join(scalist)
   spss.Submit("DESCRIPTIVES " + scaleVars + ".")
END PROGRAM.
```

- Two lists, *catlist* and *scalist*, are created to hold the names of any categorical and scale variables, respectively. They are initialized to empty lists.

- spss.GetVariableName(i) returns the name of the variable with the index value *i*.

- spss.GetVariableMeasurementLevel(i) returns the measurement level of the variable with the index value *i*. It returns one of four strings: 'nominal', 'ordinal', 'scale', or 'unknown'. If the current variable is either nominal or ordinal, it is added to the list of categorical variables; otherwise, it is added to the list of scale variables. The Python append method is used to add elements to the lists.

- Tests are performed to determine whether there are categorical or scale variables before running a FREQUENCIES or DESCRIPTIVES command. For example, if there are no categorical variables in the dataset, len(catlist) will be zero and interpreted as false for the purpose of evaluating an if statement.

- " ".join(catlist) uses the Python string method join to create a string from the elements of *catlist*, with each element separated by a single space, and likewise for " ".join(scalist).

- The dataset used in this example contains categorical and scale variables, so both a FREQUENCIES and a DESCRIPTIVES command will be submitted to IBM® SPSS® Statistics. The command strings passed to the Submit function are:

  'FREQUENCIES gender educ jobcat minority.'

  'DESCRIPTIVES id bdate salary salbegin jobtime prevexp.'

Listing Variables of a Specified Format

The GetVariableFormat function, from the spss module, returns a string containing the display format for a specified variable—for example, F4, ADATE10, DOLLAR8. Perhaps you need to find all variables of a particular format type, such as all variables with an ADATE format. This is best done with a Python user-defined function that takes the alphabetic part of the format as a parameter and returns a list of variables of that format type.

```
def VarsWithFormat(format):
    """Return a list of variables in the active dataset whose
    display format has the specified string as the alphabetic part
    of its format, e.g. "TIME".
    """
    varList=[]
    format=format.upper()
    for i in range(spss.GetVariableCount()):
        vfmt=spss.GetVariableFormat(i)
        if vfmt.rstrip("0123456789.")==format:
            varList.append(spss.GetVariableName(i))
    return varList
```

- VarsWithFormat is a Python user-defined function that requires a single argument, *format*.

- *varList* is created to hold the names of any variables in the active dataset whose display format has the specified string as its alphabetic part. It is initialized to the empty list.

■ The value returned from GetVariableFormat is in upper case, so the value of *format* is converted to upper case before doing any comparisons.

■ The value returned from GetVariableFormat consists of the alphabetic part of the format, the defined width, and optionally, the number of decimal positions for numeric formats. The alphabetic part of the format is extracted by stripping any numeric characters and periods (.), using the Python string method rstrip.

Example

As a concrete example, print a list of variables with a time format.

```
*python_list_time_vars.sps.
DATA LIST FREE
  /numvar (F4) timevar1 (TIME5) stringvar (A2) timevar2 (TIME12.2).
BEGIN DATA
1 10:05 a 11:15:33.27
END DATA.

BEGIN PROGRAM.
import samplelib
print samplelib.VarsWithFormat("TIME")
END PROGRAM.
```

■ The DATA LIST command creates four variables, two of which have a time format, and BEGIN DATA creates one sample case.

■ The BEGIN PROGRAM block starts with a statement to import the samplelib module, which contains the definition for the VarsWithFormat function.

Note: To run this program block, copy the module file *samplelib.py* from the */examples/python* folder, in the accompanying examples, to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

The result is:

```
['timevar1', 'timevar2']
```

Checking If a Variable Exists

A common scenario is to run a particular block of command syntax only if a specific variable exists in the dataset. For example, you are processing many datasets containing employee records and want to split them by gender—if a gender variable exists—to obtain separate statistics for the two gender groups. We will assume that if a gender variable exists, it has the name *gender*,

although it may be spelled in upper case or mixed case. The following example illustrates the approach using a sample dataset:

```
*python_var_exists.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
for i in range(spss.GetVariableCount()):
   name=spss.GetVariableName(i)
   if name.lower()=="gender":
      spss.Submit(r"""
      SORT CASES BY %s.
      SPLIT FILE
        LAYERED BY %s.
      """ %(name,name))
      break
END PROGRAM.
```

- `spss.GetVariableName(i)` returns the name of the variable with the index value *i*.

- Python is case sensitive, so to ensure that you don't overlook a gender variable because of case issues, equality tests should be done using all upper case or all lower case, as shown here. The Python string method `lower` converts the associated string to lower case.

- A triple-quoted string is used to pass a block of command syntax to IBM® SPSS® Statistics using the `Submit` function. The name of the gender variable is inserted into the command block using string substitution. For more information, see the topic "Dynamically Specifying Command Syntax Using String Substitution" in Chapter 13 on p. 200.

- The `break` statement terminates the loop if a gender variable is found.

To complicate matters, suppose some of your datasets have a gender variable with an abbreviated name, such as *gen* or *gndr*, but the associated variable label always contains the word *gender*. You would then want to test the variable label instead of the variable name (we'll assume that only a gender variable would have *gender* as part of its label). This is easily done by using the `GetVariableLabel` function and replacing

```
name.lower()=="gender"
```

in the `if` statement with

```
"gender" in spss.GetVariableLabel(i).lower()
```

Since `spss.GetVariableLabel(i)` returns a string, you can invoke a Python string method directly on its returned value, as shown above with the `lower` method.

Creating Separate Lists of Numeric and String Variables

The GetVariableType function, from the spss module, returns an integer value of 0 for numeric variables or an integer equal to the defined length for string variables. You can use this function to create separate lists of numeric variables and string variables in the active dataset, as shown in the following example:

```
*python_list_by_type.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
numericVars=[]
stringVars=[]
for i in range(spss.GetVariableCount()):
   if spss.GetVariableType(i) == 0:
      numericVars.append(spss.GetVariableName(i))
   else:
      stringVars.append(spss.GetVariableName(i))
print "String variables:"
print "\n".join(stringVars)
print "\nNumeric variables:"
print "\n".join(numericVars)
END PROGRAM.
```

- The lists *numericVars* and *stringVars* are created to hold the names of the numeric variables and string variables, respectively. They are initialized to empty lists.

- spss.GetVariableType(i) returns an integer representing the variable type for the variable with the index value *i*. If the returned value is 0, then the variable is numeric, so add it to the list of numeric variables; otherwise, add it to the list of string variables.

- The code "\n".join(stringVars) uses the Python string method join to combine the items in *stringVars* into a string with each element separated by "\n", which is the Python escape sequence for a line break. The result is that each element is displayed on a separate line by the print statement.

Retrieving Definitions of User-Missing Values

The GetVarMissingValues function, from the spss module, returns the user-missing values for a specified variable.

```
*python_user_missing_defs.sps.
data  list list (,)/v1 to v4(4f) v5(a4).
begin data.
0,0,0,0,a
end data.

missing values v2(0,9) v3(0 thru 1.5) v4 (LO thru 0, 999) v5(' ').

begin program.
import spss
low, high = spss.GetSPSSLowHigh()
for i in range(spss.GetVariableCount()):
  missList = spss.GetVarMissingValues(i)
  if missList[0] == 0 and missList[1] == None:
    res = 'no missing values'
  else:
    res = missList
    res = [x==low and "LO" or x==high and "HIGH" or x for x in res]
  print spss.GetVariableName(i), res
end program.
```

Result

```
v1 no missing values
v2 [0, 0.0, 9.0, None]
v3 [1, 0.0, 1.5, None]
v4 [2, 'LO', 0.0, 999.0]
v5 [0, '    ', None, None]
```

■ The `GetSPSSLowHigh` function, from the `spss` module, is used to get the actual values IBM® SPSS® Statistics uses for `LO` and `HI`, which are then stored to the Python variables *low* and *high*.

■ The `GetVarMissingValues` method returns a tuple of four elements, where the first element specifies the missing value type: 0 for discrete values, 1 for a range of values, and 2 for a range of values and a single discrete value. The remaining three elements in the result specify the missing values.

■ For variables with no missing values, the result is `[0,None,None,None]`. Testing that the first element of the result is `0` and the second is `None` is sufficient to determine the absence of missing values.

■ For variables with discrete missing values, the second, third, and fourth elements of the result specify the missing values. The result will contain one or more `None` values when there are less than three missing values, as for the variable *v2* in the current example.

■ For variables with a range of missing values, the second and third elements of the result specify the lower and upper limits of the range, respectively. In the current example, the range `0` to `1.5` is specified as missing for the variable *v3*. The result from `GetVarMissingValues` is `[1,0.0,1.5,None]`.

■ For variables with a range of missing values and a single discrete missing value, the second and third elements of the result specify the range and the fourth element specifies the discrete value. In the current example, the range `LO` to `0` is specified as missing for the variable *v4*, along with the discrete value `999`. When a missing value range is specified with `LO` or `HI`, the result contains the value SPSS Statistics uses for `LO` or `HI`. The list comprehension `[x==low and "LO" or x==high and "HIGH" or x for x in res]` replaces any values of `LO` and `HI` in the result with the strings `"LO"` and `"HI"`. In the present case, the displayed result is `[2, 'LO', 0.0, 999.0]`.

Note: If you are not familiar with list comprehensions, see the section "List Comprehensions" in the Python tutorial, available at *http://docs.python.org/tut/tut.html*.

■ For string variables, the missing value type is always 0 since only discrete missing values are allowed. Returned values are right-padded to the defined width of the string variable, as shown for the variable *v5* in the current example. In the case of a long string variable (a string variable with a maximum width greater than eight bytes), the returned value is right-padded to a width of 8, which is the maximum width of a missing value for a long string variable.

The Spssdata class in the spssdata module (a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python) provides a number of convenient functions, built on GetVarMissingValues, for dealing with missing values when reading data. For more information, see the topic "Reading Case Data with the Spssdata Class" in Chapter 15 on p. 246.

Identifying Variables without Value Labels

The task of retrieving value label information can be done in a variety of ways. For small datasets, it is most easily done by using the VariableDict class from the spssaux module (see a reworking of the example in this section on p. 229). Alternatively, you can use the valueLabels property of the Variable class as long as you don't need to retrieve the information in the context of a procedure. For more information, see the topic "Example: Displaying Value Labels as Cases in a New Dataset" in Chapter 16 on p. 274. The approach in this section uses the CreateXPathDictionary function (from the spss module) to create an in-memory XML representation of the dictionary for the active dataset, from which you can extract dictionary information. Information can be retrieved with a variety of tools, including the EvaluateXPath function from the spss module. This approach is best suited for datasets with large dictionaries and can be used in any circumstance.

As an example, we'll obtain a list of the variables that do not have value labels. The example utilizes the xml.sax module, a standard module distributed with Python that simplifies the task of working with XML and provides an alternative to the EvaluateXPath function. The first step is to define a Python class to select the XML elements and associated attributes of interest. Not surprisingly, the discussion that follows assumes familiarity with classes in Python.

```
class valueLabelHandler(ContentHandler):
    """Create two sets: one listing all variable names and
    the other listing variables with value labels"""
    def __init__(self):
        self.varset = set()
        self.vallabelset = set()
    def startElement(self, name, attr):
        if name == u"variable":
            self.varset.add(attr.getValue(u"name"))
        elif name == u"valueLabelVariable":
            self.vallabelset.add(attr.getValue(u"name"))
```

■ The job of selecting XML elements and attributes is accomplished with a content handler class. You define a content handler by inheriting from the base class ContentHandler that is provided with the xml.sax module. We'll use the name *valueLabelHandler* for our version of a content handler.

- The `__init__` method defines two attributes, *varset* and *vallabelset*, that will be used to store the set of all variables in the dataset and the set of all variables with value labels. The attributes *varset* and *vallabelset* are defined as Python sets and, as such, they support all of the usual set operations, such as intersections, unions, and differences. In fact, the set of variables without value labels is just the difference of the two sets *varset* and *vallabelset*.

- The `startElement` method of the content handler processes every element in the variable dictionary. In the present example, it selects the name of each variable in the dictionary as well as the name of any variable that has associated value label information and updates the two sets *varset* and *vallabelset*.

  Specifying the elements and attributes of interest requires familiarity with the schema for the XML representation of the IBM® SPSS® Statistics dictionary. For example, you need to know that variable names can be obtained from the *name* attribute of the *variable* element, and variables with value labels can be identified simply by retrieving the *name* attribute from each *valueLabelVariable* element. Documentation for the dictionary schema is available in the Help system.

- The strings specifying the element and attribute names are prefaced with a `u`, which makes them Unicode strings. This ensures compatibility with the XML representation of the SPSS Statistics dictionary, which is in Unicode.

Once you have defined a content handler, you define a Python function to parse the XML, utilizing the content handler to retrieve and store the desired information.

```
def FindVarsWithoutValueLabels():
    handler = valueLabelHandler()
    tag = "D"+ str(random.uniform(0,1))
    spss.CreateXPathDictionary(tag)

    # Retrieve and parse the variable dictionary
    xml.sax.parseString(spss.GetXmlUtf16(tag),handler)
    spss.DeleteXPathHandle(tag)

    # Print a list of variables in varset that aren't in vallabelset
    nolabelset = handler.varset.difference(handler.vallabelset)
    if nolabelset:
        print "The following variables have no value labels:"
        print "\n".join([v for v in nolabelset])
    else:
        print "All variables in this dataset have at least one value label."
```

- `handler = valueLabelHandler()` creates an instance of the `valueLabelHandler` class and stores a reference to it in the Python variable *handler*.

- `spss.CreateXPathDictionary(tag)` creates an XML representation of the dictionary for the active dataset. The argument *tag* defines an identifier used to specify this dictionary in subsequent operations. The dictionary resides in an in-memory workspace—referred to as the XML workspace—which can contain procedure output and dictionaries, each with its own identifier. To avoid possible conflicts with identifiers already in use, the identifier is constructed using the string representation of a random number.

- The `parseString` function does the work of parsing the XML, making use of the content handler to select the desired information. The first argument is the XML to be parsed, which is provided here by the `GetXmlUtf16` function from the `spss` module. It takes the identifier for the desired item in the XML workspace and retrieves the item. The second argument is the handler to use—in this case, the content handler defined by the `valueLabelHandler`

class. At the completion of the `parseString` function, the desired information is contained in the attributes *varset* and *vallabelset* in the handler instance.

■ `spss.DeleteXPathHandle(tag)` deletes the XML dictionary item from the XML workspace.

■ As mentioned above, the set of variables without value labels is simply the difference between the sets *varset* and *vallabelset*. This is computed using the `difference` method for Python sets and the result is stored to *nolabelset*.

In order to make all of this work, you include both the function and the class in a Python module along with the following set of `import` statements for the necessary modules:

```
from xml.sax.handler import ContentHandler
import xml.sax
import random, codecs, locale
import spss
```

Example

As a concrete example, determine the set of variables in *Employee data.sav* that do not have value labels.

```
*python_vars_no_value_labels_xmlsax.sps.
BEGIN PROGRAM.
import spss, FindVarsUtility
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
FindVarsUtility.FindVarsWithoutValueLabels()
END PROGRAM.
```

The `BEGIN PROGRAM` block starts with a statement to import the `FindVarsUtility` module, which contains the definition for the `FindVarsWithoutValueLabels` function as well as the definition for the `valueLabelHandler` class.

Note: To run this program block, copy the module file *FindVarsUtility.py*, located in the */examples/python* folder of the accompanying examples, to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1. If you are interested in making use of the `xml.sax` module, the `FindVarsUtility` module may provide a helpful starting point.

Identifying Variables with Custom Attributes

The `GetVarAttributeNames` and `GetVarAttributes` functions, from the `spss` module, allow you to retrieve information about any custom variable attributes for the active dataset.

Example

A number of variables in the sample dataset *employee_data_attrs.sav* have a variable attribute named `'DemographicVars'`. Create a list of these variables.

```
*python_var_attr.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/employee_data_attrs.sav'.")
varList=[]
attribute='DemographicVars'
for i in range(spss.GetVariableCount()):
   if (attribute in spss.GetVarAttributeNames(i)):
      varList.append(spss.GetVariableName(i))
if varList:
   print "Variables with attribute " + attribute + ":"
   print '\n'.join(varList)
else:
   print "No variables have the attribute " + attribute
END PROGRAM.
```

- The `GetVarAttributeNames` function returns a tuple containing the names of any custom variable attributes for the specified variable.

- The Python variable *varList* contains the list of variables that have the specified attribute.

Retrieving Datafile Attributes

The `GetDataFileAttributeNames` and `GetDataFileAttributes` functions, from the `spss` module, allow you to retrieve information about any datafile attributes for the active dataset.

Example

The sample dataset *employee_data_attrs.sav* has a number of datafile attributes. Determine if the dataset has a datafile attribute named `'LastRevised'`. If the attribute exists, then retrieve its value.

```
*python_file_attr.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/employee_data_attrs.sav'.")
for name in spss.GetDataFileAttributeNames():
   if (name == 'LastRevised'):
      print "Dataset last revised on:", spss.GetDataFileAttributes(name)[0]
END PROGRAM.
```

- The `GetDataFileAttributeNames` function returns a tuple of names of the datafile attributes, if any, for the active dataset.

- The `GetDataFileAttributes` function returns a tuple of the values (datafile attributes can consist of an array of values) for the specified datafile attribute. In the present example, the attribute `'LastRevised'` consists of a single value, which is the 0th element of the result.

Passing Information from Command Syntax to Python

Datafile attributes are stored in a dataset's dictionary and apply to the dataset as a whole, rather than to particular variables. Their global nature makes them suitable for storing information to be passed from command syntax (residing outside of program blocks) to program blocks that follow, as shown in this example:

```
*python_pass_value_to_python.sps.
GET FILE='/examples/data/Employee data.sav'.
DATAFILE ATTRIBUTE ATTRIBUTE=pythonArg('cheese').
BEGIN PROGRAM.
import spss
product = spss.GetDataFileAttributes('pythonArg')[0]
print "Value passed to Python:",product
END PROGRAM.
```

- Start by loading a dataset, which may or may not be the dataset that you ultimately want to use for an analysis. Then add a datafile attribute whose value is the value you want to make available to Python. If you have multiple values to pass, you can use multiple attributes or an attribute array. The attribute(s) are then accessible from program blocks that follow the DATAFILE ATTRIBUTE command(s). In the current example, we've created a datafile attribute named *pythonArg* with a value of 'cheese'.

- The program block following the DATAFILE ATTRIBUTE command uses the GetDataFileAttributes function to retrieve the value of *pythonArg*. The value is stored to the Python variable *product*.

Retrieving Multiple Response Sets

The GetMultiResponseSetNames and GetMultiResponseSet functions, from the spss module, allow you to retrieve information about any multiple response sets for the active dataset.

Example

The sample dataset *telco_extra_mrsets.sav* has a number of multiple response sets. Store the multiple response sets in a Python dictionary and display the elementary variables associated with each set.

```
*python_mrset.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/telco_extra_mrsets.sav'.")
dict = {}
for name in spss.GetMultiResponseSetNames():
  mrset = spss.GetMultiResponseSet(name)
  dict[name]={'label':mrset[0],'coding':mrset[1],'counted':mrset[2],
              'type':mrset[3],'vars':mrset[4]}
for name, set in dict.iteritems():
  print "\nElementary Variables for " + name
  print "\n".join(set['vars'])
END PROGRAM.
```

- The GetMultiResponseSetNames function returns a list of names of the multiple response sets, if any, for the active dataset.

- The `GetMultiResponseSet` function returns the details of the specified multiple response set. The result is a tuple of five elements. The first element is the label, if any, for the set. The second element specifies the variable coding—'Categories' or 'Dichotomies'. The third element specifies the counted value and applies only to multiple dichotomy sets. The fourth element specifies the data type—'Numeric' or 'String'. The fifth element is a list of the elementary variables that define the set.
- The Python variable *dict* is a Python dictionary whose keys are the names of the multiple response sets. The value associated with each key is also a Python dictionary and consists of the details of the multiple response set.

Using Object-Oriented Methods for Retrieving Dictionary Information

The `spssaux` module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python, provides object-oriented methods that simplify the task of retrieving variable dictionary information.

Getting Started with the VariableDict Class

The object-oriented methods for retrieving dictionary information are encapsulated in the `VariableDict` class in the `spssaux` module. In order to use these methods, you first create an instance of the `VariableDict` class and store it to a variable, as in:

```
varDict = spssaux.VariableDict()
```

When the argument to `VariableDict` is empty, as shown above, the instance will contain information for all variables in the active dataset. Of course, you have to include the statement `import spssaux` so that Python can load the functions and classes in the `spssaux` module. Note that if you delete, rename, or reorder variables in the active dataset, you should obtain a refreshed instance of the `VariableDict` class.

You can also call `VariableDict` with a list of variable names or a list of index values for a set of variables. The resulting instance will then contain information for just that subset of variables. To illustrate this, consider the variables in *Employee data.sav* and an instance of `VariableDict` that contains the variables *id*, *salary*, and *jobcat*. To create this instance from a list of variable names, use:

```
varDict = spssaux.VariableDict(['id','salary','jobcat'])
```

The same instance can be created from a list of variable index values, as in:

```
varDict = spssaux.VariableDict([0,5,4])
```

Remember that an index value of 0 corresponds to the first variable in file order, so the variable *id* has an index of 0, the variable *salary* has an index of 5, and the variable *jobcat* has an index of 4.

The number of variables in the current instance of the class is available from the `numvars` property, as in:

```
varDict.numvars
```

A Python list of variables in the current instance of the class is available from the `variablesf` method, as in:

```
varDict.variablesf()
```

You may want to consider creating multiple instances of the `VariableDict` class, each assigned to a different variable and each containing a particular subset of variables that you need to work with.

Note: You can select variables for an instance of `VariableDict` by variable type (`'numeric'` or `'string'`), by variable measurement level (`'nominal'`, `'ordinal'`, `'scale'`, or `'unknown'`), or by using a regular expression; and you can specify any combination of these criteria. You can also specify these same types of criteria for the `variablesf` method in order to list a subset of the variables in an existing instance. For more information on using regular expressions, see "Using Regular Expressions to Select Variables" on p. 230. For more information on selecting variables by variable type or variable level, include the statement `help(spssaux.VariableDict)` in a program block, after having imported the `spssaux` module.

Retrieving Variable Information

Once you have created an instance of the `VariableDict` class, you have a variety of ways of retrieving variable dictionary information.

Looping through the variables in an instance of VariableDict. You can loop through the variables, extracting information one variable at a time, by iterating over the instance of `VariableDict`. For example,

```
varDict = spssaux.VariableDict()
for var in varDict:
   print var, var.VariableName, "\t", var.VariableLabel
```

- The Python variable *varDict* holds an instance of the `VariableDict` class for all of the variables in the active dataset.
- On each iteration of the loop, the Python variable *var* is an object representing a different variable in *varDict* and provides access to that variable's dictionary information through properties of the object. For example, `var.VariableName` returns the variable name for the variable represented by the current value of *var*, and including `var` by itself returns the index value of the current variable.

Note: A list of all available properties and methods for the `VariableDict` class can be obtained by including the statement `help(spssaux.VariableDict)` in a program block, assuming that you have already imported the `spssaux` module.

Accessing information by variable name. You can retrieve information for any variable in the current instance of `VariableDict` simply by specifying the variable name. For example, to retrieve the measurement level for a variable named *jobcat*, use:

```
varDict['jobcat'].VariableLevel
```

Accessing information by a variable's index within an instance. You can access information for a particular variable using its index within an instance. When you call `VariableDict` with an explicit variable list, the index within the instance is simply the position of the variable in that list, starting from 0. For example, consider the following instance based on *Employee data.sav* as the active dataset:

```
varDict = spssaux.VariableDict(['id','salary','jobcat'])
```

The index 0 in the instance refers to *id*, 1 refers to *salary*, and 2 refers to *jobcat*. The code to retrieve, for example, the variable name for the variable with index 1 in the instance is:

```
varDict[1].VariableName
```

The result, of course, is `'salary'`. Notice that *salary* has an index value of 5 in the associated dataset but an index of 1 in the instance. This is an important point; in general, the variable's index value in the dataset isn't equal to its index in the instance.

It may be convenient to obtain the variable's index value in the dataset from its index in the instance. As an example, get the index value in the dataset of the variable with index 2 in `varDict`. The code is:

```
varDict[2]
```

The result is 4, since the variable with index 2 in the instance is *jobcat* and it has an index value of 4 in the dataset.

Accessing information by a variable's index value in the dataset. You also have the option of addressing variable properties by the index value in the dataset. This is done using the index value as an argument to a method call. For example, to get the name of the variable with the index value of 4 in the dataset, use:

```
varDict.VariableName(4)
```

For the dataset and instance used above, the result is `'jobcat'`.

Setting Variable Properties

The `VariableDict` class allows you to set a number of properties for existing variables in the active dataset. You can set the variable label, the measurement level, the output format, value labels, missing values, and variable attributes. For example, to update the variable label of *jobtime* to 'Months on the job' in *Employee data.sav*, use:

```
varDict = spssaux.VariableDict()
varDict['jobtime'].VariableLabel='Months on the job'
```

For more information, include the statement `help(spssaux.Variable)` in a program block.

Defining a List of Variables between Two Variables

Sometimes you cannot use references such as var1 TO xyz5; you have to actually list all of the variables of interest. This task is most easily done using the range method from the VariableDict class. As a concrete example, print the list of scale variables between *bdate* and *jobtime* in *Employee data.sav*.

```
*python_vars_between_vars.sps.
BEGIN PROGRAM.
import spssaux
spssaux.OpenDataFile('/examples/data/Employee data.sav')
vdict=spssaux.VariableDict()
print vdict.range(start="bdate",end="jobtime",variableLevel=["scale"])
END PROGRAM.
```

■ The OpenDataFile function from the spssaux module is used to open *Employee data.sav*. The argument to the function is the file specification in quotes. Although not used here, OpenDataFile also allows you to associate a dataset name with the opened file. For more information, include the statement help(spssaux.OpenDataFile) in a program block, after having imported the spssaux module.

■ The range method from the VariableDict class returns a list of variable names (from the current instance of class) between the variables specified by the arguments *start* and *end*. In the current example, the instance of VariableDict contains all of the variables in the active dataset, in file order. When the *variableLevel* argument is used, only those variables with one of the specified measurement levels will be included in the list. The variables specified as *start* and *end* (*bdate* and *jobtime* in this example) are considered for inclusion in the list.

For more information on the range method, include the statement help(spssaux.VariableDict.range) in a program block.

Specifying Variable Lists with TO and ALL

Sometimes you'll want to specify variable lists with the TO and ALL keywords, like you can with variable lists in IBM® SPSS® Statistics command syntax. This is particularly useful if you're writing an extension command (a user-defined SPSS Statistics command implemented in Python or R) and want to provide users with the convenience of TO and ALL. Handling TO and ALL is accomplished with the expand method from the VariableDict class.

Example: Using ALL

```
*python_ALL_keyword.sps.
BEGIN PROGRAM.
import spssaux
spssaux.OpenDataFile('/examples/data/Employee data.sav')
vdict=spssaux.VariableDict(variableLevel=['scale'])
print vdict.expand("ALL")
END PROGRAM.
```

- An instance of the `VariableDict` class is created for the scale variables in *Employee data.sav* and saved to the Python variable *vdict*.

- `vdict.expand("ALL")` returns a list of all of the variables in the `VariableDict` instance—in this case, all of the scale variables in *Employee data.sav*. The result is:

  ```
  ['salary', 'bdate', 'prevexp', 'jobtime', 'salbegin', 'id']
  ```

Example: Using TO

```
*python_TO_keyword.sps.
BEGIN PROGRAM.
import spssaux
spssaux.OpenDataFile('/examples/data/Employee data.sav')
vdict=spssaux.VariableDict()
print vdict.expand(["educ","TO","prevexp"])
END PROGRAM.
```

- An instance of the `VariableDict` class is created for the all of the variables in *Employee data.sav* and saved to the Python variable *vdict*.

- `vdict.expand(["educ","TO","prevexp"])` returns a list of all of the variables in the `VariableDict` instance between *educ* and *prevexp* inclusive. The result is:

  ```
  ['educ', 'jobcat', 'salary', 'salbegin', 'jobtime', 'prevexp']
  ```

You can also specify the range of variables in a character string, as in `vdict.expand("educ TO prevexp")`, and you can include variables in addition to the endpoints of the range, as in `vdict.expand(["gender","educ","TO","prevexp"])`. Finally, variable names specified for the `expand` method are not case sensitive.

For more information on the `expand` method, include the statement `help(spssaux.VariableDict.expand)` in a program block.

Identifying Variables without Value Labels

The `VariableDict` class allows you to retrieve value label information through the `ValueLabels` property. The following example shows how to obtain a list of variables that do not have value labels:

```
*python_vars_no_value_labels.sps.
BEGIN PROGRAM.
import spss, spssaux
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
varDict = spssaux.VariableDict()
varList = [var.VariableName for var in varDict
           if not var.ValueLabels]
print "List of variables without value labels:"
print "\n".join(varList)
END PROGRAM.
```

- `var.ValueLabels` returns a Python dictionary containing value label information for the variable represented by *var*. If there are no value labels for the variable, the dictionary will be empty and `var.ValueLabels` will be interpreted as false for the purpose of evaluating an `if` statement.

■ The Python variable *varList* contains the list of variables that do not have value labels. *Note*: If you are not familiar with the method used here to create a list, see the section "List Comprehensions" in the Python tutorial, available at *http://docs.python.org/tut/tut.html*.

■ If you have PRINTBACK and MPRINT on, you'll notice a number of OMS commands in the Viewer log when you run this program block. The ValueLabels property utilizes OMS to get value labels from the active dataset's dictionary.

The method used above for finding variables without value labels can be quite expensive when processing all of the variables in a large dictionary. In such cases, consider using the valueLabels property of the Variable class as long as you don't need to retrieve the information in the context of a procedure. For more information, see the topic "Example: Displaying Value Labels as Cases in a New Dataset" in Chapter 16 on p. 274.

Using Regular Expressions to Select Variables

Regular expressions define patterns of characters and enable complex string searches. For example, using a regular expression, you could select all variables in the active dataset whose names end in a digit. The VariableDict class allows you to use regular expressions to select the subset of variables for an instance of the class or to obtain a selected list of variables in an existing instance.

Example

The sample dataset *demo.sav* contains a number of variables whose names begin with 'own', such as *owntv* and *ownvcr*. We'll use a regular expression to create an instance of VariableDict that contains only variables with names beginning with 'own'.

```
*python_re_1.sps.
BEGIN PROGRAM.
import spss, spssaux
spss.Submit("GET FILE='/examples/data/demo.sav'.")
varDict = spssaux.VariableDict(pattern=r'own')
print "\n".join(varDict.variablesf())
END PROGRAM.
```

■ The argument *pattern* is used to specify a regular expression when creating an instance of the VariableDict class. A variable in the active dataset is included in the instance only if the regular expression provides a match to its name. When testing a regular expression against a name, comparison starts with the beginning of the name. In the current example, the regular expression is simply the string 'own' and will provide a match to any variable whose name begins with 'own'. Patterns for regular expressions are always case insensitive.

■ Notice that the string for the regular expression is prefaced with r, indicating that it will be treated as a raw string. It is a good idea to use raw strings for regular expressions to avoid unintentional problems with backslashes. For more information, see the topic "Using Raw Strings in Python" in Chapter 13 on p. 202.

■ The variablesf method of VariableDict creates a Python list of all variables in the current instance.

Example

In the following example, we create a sample dataset containing some variables with names that end in a digit and create an instance of `VariableDict` containing all variables in the dataset. We then show how to obtain the list of variables in the instance whose names end in a digit.

```
*python_re_2.sps.
DATA LIST FREE
  /id  gender age incat region score1 score2 score3.
BEGIN DATA
1 0 35 3 10 85 76 63
END DATA.
BEGIN PROGRAM.
import spssaux
varDict = spssaux.VariableDict()
print "\n".join(varDict.variablesf(pattern=r'.*\d$'))
END PROGRAM.
```

- The argument *pattern* can be used with the `variablesf` method of `VariableDict` to create a list of variables in the instance whose names match the associated regular expression. In this case, the regular expression is the string `'.*\d$'`.

- If you are not familiar with the syntax of regular expressions, a good introduction can be found in the section "Regular expression operations" in the Python Library Reference, available at *http://docs.python.org/lib/module-re.html*. Briefly, the character combination `'.*'` will match an arbitrary number of characters (other than a line break), and `'\d'` will match a single digit at the end of a string. The combination `'.*\d$'` will then match any string that ends in a digit. For an example that uses a more complex regular expression, see "Using Regular Expressions" on p. 331.

Working with Case Data in the Active Dataset

The IBM® SPSS® Statistics - Integration Plug-In for Python provides the ability to read case data from the active dataset, create new variables in the active dataset, and append new cases to the active dataset. This is accomplished using methods from the `Cursor` class, available with the `spss` module. To concurrently access multiple open datasets, use the `Dataset` class. For more information, see the topic "Creating and Accessing Multiple Datasets" in Chapter 16 on p. 261.

Using the Cursor Class

The `Cursor` class provides three usage modes: read mode allows you to read cases from the active dataset, write mode allows you to add new variables (and their case values) to the active dataset, and append mode allows you to append new cases to the active dataset. To use the `Cursor` class, you first create an instance of the class and store it to a Python variable, as in:

```
dataCursor = spss.Cursor(accessType='w')
```

The optional argument *accessType* specifies the usage mode: read (`'r'`), write (`'w'`), or append (`'a'`). The default is read mode. Each usage mode supports its own set of methods.

Note: For users of a 14.0.x version of the plug-in who are upgrading to version 15.0 or higher, read mode is equivalent to the `Cursor` class provided with 14.0.x versions. No changes to your 14.0.x code for the `Cursor` class are required to run the code with version 15.0 or higher.

Reading Case Data with the Cursor Class

To read case data, you create an instance of the `Cursor` class in read mode, as in:

```
dataCursor = spss.Cursor(accessType='r')
```

Read mode is the default mode, so specifying `accessType='r'` is optional. For example, the above is equivalent to:

```
dataCursor = spss.Cursor()
```

Invoking `Cursor` with just the `accessType` argument, or no arguments, indicates that case data should be retrieved for all variables in the active dataset.

You can also call `Cursor` with a list of index values for a set of specific variables to retrieve. Index values represent position in the active dataset, starting with 0 for the first variable in file order. To illustrate this, consider the variables in *Employee data.sav* and imagine that you want to retrieve case data for only the variables *id* and *salary*, with index values of 0 and 5, respectively. The code to do this is:

```
dataCursor = spss.Cursor([0,5])
```

Note: When working with a subset of the variables in the active dataset, you may consider using the spssdata module, which allows you to specify a list of variable names, rather than indexes. For more information, see the topic "Using the spssdata Module" on p. 245.

Example: Retrieving All Cases

Once you've created an instance of the Cursor class, you can retrieve data by invoking methods on the instance. The method for retrieving all cases is fetchall, as shown in the following example:

```
*python_get_all_cases.sps.
DATA LIST FREE /var1 (F) var2 (A2).
BEGIN DATA
11 ab
21 cd
31 ef
END DATA.
BEGIN PROGRAM.
import spss
dataCursor=spss.Cursor()
data=dataCursor.fetchall()
dataCursor.close()
print "Case data:", data
END PROGRAM.
```

- The fetchall method doesn't take any arguments, but Python still requires a pair of parentheses when calling the method.

- The Python variable *data* contains the data for all cases and all variables in the active dataset.

- dataCursor.close() closes the Cursor object. Once you've retrieved the needed data, you should close the Cursor object, since you can't use the spss.Submit function while a data cursor is open.

Note: When reading from datasets with splits, fetchall returns the remaining cases in the current split. For more information on working with splits, see the example "Handling Data with Splits" in this section.

Result

```
Case data: ((11.0, 'ab'), (21.0, 'cd'), (31.0, 'ef'))
```

- The case data is returned as a list of Python **tuples**. Each tuple represents the data for one case, and the tuples are arranged in the same order as the cases in the dataset. For example, the tuple containing the data for the first case in the dataset is (11.0, 'ab'), the first tuple in the list. If you're not familiar with the concept of a Python tuple, it's a lot like a Python list—it consists of a sequence of addressable elements. The main difference is that you can't change an element of a tuple like you can for a list. You can of course replace the tuple, effectively changing it.

- Each element in one of these tuples contains the data value for a specific variable. When you invoke the Cursor class with spss.Cursor(), as in this example, the elements correspond to the variables in file order.

Note: When working with retrieved case data, you may also consider using the `spssdata` module, which allows you to address elements of tuples (containing case data) by the name of the associated variable instead of its index value. For more information, see the topic "Using the spssdata Module" on p. 245.

- By default, missing values are converted to the Python data type *None*, which is used to signify the absence of a value. For more information on missing values, see the example on "Missing Data" that follows.

Note: Be careful when using the `fetchall` method for large datasets, since Python holds the retrieved data in memory. In such cases, when you have finished processing the data, consider deleting the variable used to store it. For example, if the data are stored in the variable *data*, you can delete the variable with `del data`.

Example: Retrieving Cases Sequentially

You can retrieve cases one at a time in sequential order using the `fetchone` method.

```
*python_get_cases_sequentially.sps.
DATA LIST FREE /var1 (F) var2 (A2).
BEGIN DATA
11 ab
21 cd
END DATA.
BEGIN PROGRAM.
import spss
dataCursor=spss.Cursor()
print "First case:", dataCursor.fetchone()
print "Second case:", dataCursor.fetchone()
print "End of file reached:", dataCursor.fetchone()
dataCursor.close()
END PROGRAM.
```

Each call to `fetchone` retrieves the values of the specified variables (in this example, all variables) for the next case in the active dataset. The `fetchone` method doesn't take any arguments.

Result

```
First case: (11.0, 'ab')
Second case: (21.0, 'cd')
End of file reached: None
```

Calling `fetchone` after the last case has been read returns the Python data type *None*.

Example: Retrieving Data for a Selected Variable

As an example of retrieving data for a subset of variables, we'll take the case of a single variable.

```
*python_get_one_variable.sps.
DATA LIST FREE /var1 (F) var2 (A2) var3 (F).
BEGIN DATA
11 ab 13
21 cd 23
31 ef 33
END DATA.
BEGIN PROGRAM.
import spss
dataCursor=spss.Cursor([2])
data=dataCursor.fetchall()
dataCursor.close()
print "Case data for one variable:", data
END PROGRAM.
```

The code `spss.Cursor([2])` specifies that data will be returned for the single variable with index value 2 in the active dataset. For the current example, this corresponds to the variable *var3*.

Note: When working with a subset of the variables in the active dataset, you may consider using the `spssdata` module, which allows you to specify a list of variable names, rather than indexes. For more information, see the topic "Using the spssdata Module" on p. 245.

Result

```
Case data for one variable: ((13.0,), (23.0,), (33.0,))
```

The data for each case is represented by a tuple containing a single element. Python denotes such a tuple by following the value with a comma, as shown here.

Example: Missing Data

In this example, we create a dataset that includes both system-missing and user-missing values.

```
*python_get_missing_data.sps.
DATA LIST LIST (',') /numVar (f) stringVar (a4).
BEGIN DATA
1,a
,b
3,,
9,d
END DATA.
MISSING VALUES numVar (9) stringVar (' ').
BEGIN PROGRAM.
import spss
dataCursor=spss.Cursor()
data=dataCursor.fetchall()
dataCursor.close()
print "Case data with missing values:\n", data
END PROGRAM.
```

Result

```
Case data with missing values:
((1.0, 'a   '), (None, 'b   '), (3.0, None), (None, 'd   '))
```

When the data are read into Python, system-missing values are converted to the Python data type *None*, which is used to signify the absence of a value. By default, user-missing values are also converted to *None*. You can use the `SetUserMissingInclude` method to specify that

user-missing values be treated as valid, as shown in the following reworking of the previous example.

```
DATA LIST LIST (',') /numVar (f) stringVar (a4).
BEGIN DATA
1,a
,b
3,,
9,d
END DATA.
MISSING VALUES numVar (9) stringVar (' ').
BEGIN PROGRAM.
import spss
dataCursor=spss.Cursor()
dataCursor.SetUserMissingInclude(True)
data=dataCursor.fetchall()
dataCursor.close()
print "Case data with user-missing values treated as valid:\n", data
END PROGRAM.
```

Result

```
Case data with user-missing values treated as valid:
((1.0, 'a   '), (None, 'b   '), (3.0, '    '), (9.0, 'd   '))
```

Example: Handling Data with Splits

When reading datasets in which split-file processing is in effect, you'll need to be aware of the behavior at a split boundary. Detecting split changes is necessary when you're creating custom pivot tables from data with splits and want separate results displayed for each split group. The `IsEndSplit` method, from the `Cursor` class, allows you to detect split changes when reading from datasets that have splits.

```
*python_detect_split_change.sps.
DATA LIST FREE /salary (F) jobcat (F).
BEGIN DATA
21450 1
45000 1
30000 2
30750 2
103750 3
72500 3
57000 3
END DATA.

SPLIT FILE BY jobcat.

BEGIN PROGRAM.
import spss
cur=spss.Cursor()
for i in range(spss.GetCaseCount()):
    cur.fetchone()
    if cur.IsEndSplit():
        print "A new split begins at case", i+1
        # Fetch the first case of the new split group
        cur.fetchone()
cur.close()
END PROGRAM.
```

- `cur.IsEndSplit()` returns a Boolean value—*true* if a split boundary has been crossed and *false* otherwise. For the sample dataset used in this example, split boundaries are crossed when reading the third and fifth cases.

- The value returned from the `fetchone` method is *None* at a split boundary. In the current example, this means that *None* is returned when attempting to read the third and fifth cases. Once a split has been detected, you call `fetchone` again to retrieve the first case of the next split group, as shown in this example.

- Although not shown in this example, `IsEndSplit` also returns *true* when the end of the dataset has been reached. This scenario would occur if you replace the `for` loop with a `while True` loop that continues reading until the end of the dataset is detected. Although a split boundary and the end of the dataset both result in a return value of *true* from `IsEndSplit`, the end of the dataset is identified by a return value of *None* from a subsequent call to `fetchone`.

Handling IBM SPSS Statistics Datetime Values

Dates and times in IBM® SPSS® Statistics are represented internally as seconds. By default, data retrieved from SPSS Statistics for a datetime variable is returned as a floating point number representing some number of seconds and fractional seconds. SPSS Statistics knows how to correctly interpret this number when performing datetime calculations and displaying datetime values, but without special instructions, Python doesn't. To illustrate this point, consider the following sample data and code to retrieve the data:

```
DATA LIST FREE /bdate (ADATE10).
BEGIN DATA
02/13/2006
END DATA.
BEGIN PROGRAM.
import spss
data=spss.Cursor()
row=data.fetchone()
print row[0]
data.close()
END PROGRAM.
```

The result from Python is `13359168000.0`, which is a perfectly valid representation of the date 02/13/2006 if you happen to know that SPSS Statistics stores dates internally as the number of seconds since October 14, 1582. You can, however, specify to convert a datetime value from SPSS Statistics into a Python datetime object with the *cvtDates* argument to the `spss.Cursor` function. The retrieved value will then render in a recognizable date format within Python and can be manipulated with functions from the Python datetime module (a built-in module distributed with Python).

The following is a reworking of the previous example, making use of the *cvtDates* argument.

```
*python_convert_datetime_values.sps.
DATA LIST FREE /bdate (ADATE10).
BEGIN DATA
02/13/2006
END DATA.
BEGIN PROGRAM.
import spss
data=spss.Cursor(cvtDates='ALL')
row=data.fetchone()
print row[0]
data.close()
END PROGRAM.
```

- The argument *cvtDates* is a sequence of index values of the variables to convert or the name "ALL" as in this example. If a variable specified in *cvtDates* does not have a date format, it is not converted.

- The result from Python is `2006-02-13 00:00:00`, which is the display of a Python datetime object.

Working in Unicode Mode

For IBM® SPSS® Statistics 16.0 and higher, the SPSS Statistics processor from which you retrieve data can operate in code page mode (the default) or Unicode mode. In code page mode, strings are returned to Python in the character encoding of the current locale, whereas in Unicode mode, strings are returned as Python Unicode objects (more specifically, they are converted by SPSS Statistics from UTF-8 to UTF-16). This applies to variable dictionary information and string data. Objects in the XML workspace are always in Unicode.

Special care must be taken when working in Unicode mode with Python programs. Specifically, Python string literals used within Python programs in command syntax files need to be explicitly expressed as UTF-16 strings. This is best done by using the `u()` function from the `spssaux` module. The function has the following behavior:

- If SPSS Statistics is in Unicode mode, the function returns its argument in Unicode.

- If SPSS Statistics is not in Unicode mode or the argument is not a string, the argument is returned unchanged.

Note: If the `u()` function or its equivalent is not used, the literal will be encoded in UTF-8 when SPSS Statistics is in Unicode mode. Therefore, if the string literals in a command syntax file only consist of plain roman characters (7-bit ASCII), the `u()` function is not needed.

The following example demonstrates some of this behavior and the usage of the `u()` function.

```
set unicode on locale=english.
BEGIN PROGRAM.
import spss, spssaux
from spssaux import u
literal = "âbc"
try:
   print "literal without conversion:", literal
except:
   print "can't print literal"
try:
   print "literal converted to utf-16:", u(literal)
except:
   print "can't print literal"
END PROGRAM.
```

Following are the results:

```
literal without conversion: can't print literal
literal converted to utf-16: âbc
```

Creating New Variables with the Cursor Class

To add new variables along with their case values to the active dataset, you create an instance of the Cursor class in write mode, as in:

```
dataCursor = spss.Cursor(accessType='w')
```

Populating case values for new variables involves reading and updating cases, so write mode also supports the functionality available in read mode. As with a read cursor, you can create a write cursor with a list of index values for a set of specific variables (perhaps used to determine case values for the new variables). For example, to create a write cursor that also allows you to retrieve case data for the variables with index values 1 and 3 in the active dataset, use:

```
dataCursor = spss.Cursor([1,3],accessType='w')
```

Write mode also supports multiple data passes, allowing you to add new variables on any data pass. For more information, see the example on "Adding Group Percentile Values to a Dataset" on p. 243.

Example

In this example, we create a new string variable and a new numeric variable and populate their case values for the first and third cases in the active dataset.

```
*python_add_vars.sps.
DATA LIST FREE /case (A5).
BEGIN DATA
case1
case2
case3
END DATA.
BEGIN PROGRAM.
import spss
cur=spss.Cursor(accessType='w')
# Specify new variables
cur.SetVarNameAndType(['numvar','strvar'],[0,1])
cur.SetVarLabel('numvar','Sample numeric variable')
cur.SetVarLabel('strvar','Sample string variable')
cur.CommitDictionary()
# Set values for the first case in the active dataset
cur.fetchone()
cur.SetValueNumeric('numvar',1)
cur.SetValueChar('strvar','a')
cur.CommitCase()
# Set values for the third case in the active dataset
cur.fetchmany(2)
cur.SetValueNumeric('numvar',3)
cur.SetValueChar('strvar','c')
cur.CommitCase()
cur.close()
END PROGRAM.
```

- New variables are created using the `SetVarNameAndType` method from the `Cursor` class. The first argument is a list or tuple of strings that specifies the name of each new variable. The second argument is a list or tuple of integers specifying the variable type of each variable. Numeric variables are specified by a value of 0 for the variable type. String variables are specified with a type equal to the defined length of the string (a maximum of 32767). In this example, we create a numeric variable named *numvar* and a string variable of length 1 named *strvar*.

- After calling `SetVarNameAndType`, you have the option of specifying variable properties (in addition to the variable type), such as the measurement level, variable label, and missing values. In this example, variable labels are specified using the `SetVarLabel` method.

- Specifications for new variables must be committed to the cursor's dictionary before case values can be set. This is accomplished by calling the `CommitDictionary` method, which takes no arguments. The active dataset's dictionary is updated when the cursor is closed.

- To set case values, you first position the record pointer to the desired case using the `fetchone` or `fetchmany` method. `fetchone` advances the record pointer by one case, and `fetchmany` advances it by a specified number of cases. In this example, we set case values for the first and third cases.

  Note: To set the value for the first case in the dataset, you must call `fetchone` as shown in this example.

- Case values are set using the `SetValueNumeric` method for numeric variables and the `SetValueChar` method for string variables. For both methods, the first argument is the variable name and the second argument is the value for the current case. A numeric variable whose value is not specified is set to the system-missing value, whereas a string variable whose value is not specified will have a blank value. For numeric variables, you can use the value *None* to specify a system-missing value. For string variables, you can use `str(None)` to specify a blank string.

  Values of numeric variables with a date or datetime format should be specified as Python `time.struct_time` or `datetime.datetime` objects, which are then converted to the appropriate IBM® SPSS® Statistics value. Values of variables with `TIME` and `DTIME` formats should be specified as the number of seconds in the time interval.

- The `CommitCase` method must be called to commit the values for each modified case. Changes to the active dataset take effect when the cursor is closed.

Note: You cannot add new variables to an empty dataset using the `Cursor` class. If you need to create a dataset from scratch and you are a user of SPSS Statistics 15.0, use the mode `'n'` of the `Spssdata` class. For users of SPSS Statistics 16.0 and higher, it is recommended to use the `Dataset` class to create a new dataset. For more information, see the topic "Example: Creating and Saving Datasets" in Chapter 16 on p. 268.

Appending New Cases with the Cursor Class

To append new cases to the active dataset, you create an instance of the `Cursor` class in append mode, as in:

```
dataCursor = spss.Cursor(accessType='a')
```

Example

In this example, two new cases are appended to the active dataset.

```
*python_append_cases.sps.
DATA LIST FREE /case (F) value (A1).
BEGIN DATA
1 a
END DATA.
BEGIN PROGRAM.
import spss
cur=spss.Cursor(accessType='a')
cur.SetValueNumeric('case',2)
cur.SetValueChar('value','b')
cur.CommitCase()
cur.SetValueNumeric('case',3)
cur.SetValueChar('value','c')
cur.CommitCase()
cur.EndChanges()
cur.close()
END PROGRAM.
```

- Case values are set using the `SetValueNumeric` method for numeric variables and the `SetValueChar` method for string variables. For both methods, the first argument is the variable name, as a string, and the second argument is the value for the current case. A numeric variable whose value is not specified is set to the system-missing value, whereas a string variable whose value is not specified will have a blank value. For numeric variables, you can use the value *None* to specify a system-missing value. For string variables, you can use `str(None)` to specify a blank string.

- The `CommitCase` method must be called to commit the values for each new case. Changes to the active dataset take effect when the cursor is closed. When working in append mode, the cursor is ready to accept values for a new case (using `SetValueNumeric` and `SetValueChar`) once `CommitCase` has been called for the previous case.

- The `EndChanges` method signals the end of appending cases and must be called before the cursor is closed or the new cases will be lost.

Note: Append mode does not support reading case data or creating new variables. A dataset must contain at least one variable in order to append cases to it, but it need not contain any cases. If you need to create a dataset from scratch and you are a user of IBM® SPSS® Statistics 15.0, use the mode `'n'` of the `Spssdata` class. For users of SPSS Statistics 16.0 and higher, it is recommended to use the `Dataset` class to create a new dataset. For more information, see the topic "Example: Creating and Saving Datasets" in Chapter 16 on p. 268.

Example: Counting Distinct Values Across Variables

In this example, we count the distinct values across all variables for each case in the active dataset and store the results to a new variable. User-missing and system-missing values are ignored in the count of distinct values.

```
*python_distinct_values_across_variables.sps.
DATA LIST LIST (',') /var1 (F) var2 (F)  var3 (F) var4 (F).
BEGIN DATA
1,2,3,4
0,1,1,1
2,3, ,2
1,1,3,4
END DATA.
MISSING VALUES var1 (0).
BEGIN PROGRAM.
import spss
cur = spss.Cursor(accessType='w')
cur.SetVarNameAndType(['distinct'],[0])
cur.CommitDictionary()
for i in range(spss.GetCaseCount()):
    row = cur.fetchone()
    vals = set(row)
    vals.discard(None)
    cur.SetValueNumeric('distinct', len(vals))
    cur.CommitCase()
cur.close()
END PROGRAM.
```

- Since we need to read from the active dataset as well as write to it, we use an instance of the `Cursor` class in write mode.

- The `SetVarNameAndType` method is used to create the new variable *distinct* that will hold the number of distinct values for each case. The `CommitDictionary` method is called to commit the new variable before reading the data.

- The `fetchone` method is used to read each case sequentially. It also has the effect of advancing the record pointer by one case, allowing you to set the value of *distinct* for each case.

- The Python `set` function creates a set object containing the distinct elements in *row*. The `discard` method of the set object removes the value *None*, representing any user-missing or system-missing values.

Example: Adding Group Percentile Values to a Dataset

In this example, we calculate the quartiles for the cases associated with each value of a grouping variable—in particular, the quartiles for *salary* grouped by *jobcat* for the *Employee data.sav* dataset—and add the results as new variables. This involves two passes of the data. The first pass reads the data and calculates the group quartiles. The second pass adds the quartile values as new variables to the active dataset.

Note: This can also be done with the IBM® SPSS® Statistics `Rank` procedure.

```
*python_add_group_percentiles.sps.
BEGIN PROGRAM.
import spss, math
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")

# Create a cursor that will only read the values of jobcat and salary
cur=spss.Cursor(var=[4,5],accessType='w')
cur.AllocNewVarsBuffer(24)

# Accumulate frequencies of salaries for each employment category
salaries={}; counts={}
for i in range(spss.GetCaseCount()):
   row=cur.fetchone()
   jobcat=row[0]
   salary=row[1]
   salaries[jobcat]=salaries.get(jobcat,{})
   salaries[jobcat][salary]=salaries[jobcat].get(salary,0) + 1
   counts[jobcat]=counts.get(jobcat,0) + 1

# Calculate the cutpoint salary value for each percentile for each
# employment category
percentiles={}
for jobcat in salaries:
   cutpoints = [int(math.ceil(counts[jobcat]*f)) for f in [.25, .50, .75]]
   tempcount=0; pctindex=0
   percentiles[jobcat]=[]
   salarylist=sorted(salaries[jobcat].keys())
   for salary in salarylist:
      tempcount+=salaries[jobcat][salary]
      if tempcount>=cutpoints[pctindex]:
         percentiles[jobcat].append(salary)
         pctindex+=1
         if pctindex == 3:
            break

# Create and populate new variables for the percentiles
cur.reset()
cur.SetVarNameAndType(['salary_25','salary_50','salary_75'],[0,0,0])
cur.CommitDictionary()
for i in range(spss.GetCaseCount()):
   row=cur.fetchone()
   jobcat=row[0]
   cur.SetValueNumeric('salary_25',percentiles[jobcat][0])
   cur.SetValueNumeric('salary_50',percentiles[jobcat][1])
   cur.SetValueNumeric('salary_75',percentiles[jobcat][2])
   cur.CommitCase()
cur.close()
end program.
```
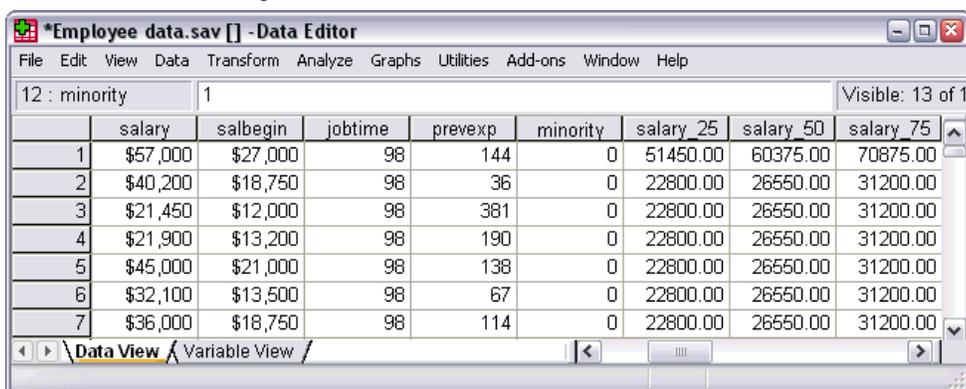
- The code makes use of the `ceil` function from the `math` module, so the `import` statement includes the `math` module.

- `spss.Cursor(var=[4,5],accessType='w')` creates a write cursor. `var=[4,5]` specifies that only values of the variables with indexes 4 (*jobcat*) and 5 (*salary*) are retrieved when reading cases with this cursor.

- In the case of multiple data passes where you need to add variables on a data pass other than the first (as in this example), you must call the `AllocNewVarsBuffer` method to allocate the buffer size for the new variables. Each numeric variable requires eight bytes, so 24 bytes are needed for the three new variables in this example. When used, `AllocNewVarsBuffer` must be called before reading any data with `fetchone`, `fetchmany`, or `fetchall` and before calling `CommitDictionary`.

- The first data pass accumulates the frequencies of each salary value for each employment category. The Python dictionary *salaries* has a key for each employment category found in the case data. The value associated with each key is itself a dictionary whose keys are the salaries and whose values are the associated frequencies for that employment category. The

code `salaries[jobcat].get(salary,0)` looks in the dictionary associated with the current employment category (*jobcat*) for a key equal to the current value of *salary*. If the key exists, its value is returned; otherwise, 0 is returned.

- The Python dictionary *percentiles* has a key for each employment category found in the case data. The value associated with each key is a list of the quartiles for that employment category. For simplicity, when a quartile boundary falls exactly on a particular case number, the associated case value (rather than an interpolation) is used as the quartile. For example, for an employment category with 84 cases, the first quartile falls exactly on case 21.

- The `reset` method is used to reset the cursor's record pointer in preparation for a second data pass. When executing multiple data passes, the `reset` method must be called prior to defining new variables on subsequent passes.

- A second data pass is used to add the variables *salary_25*, *salary_50*, and *salary_75*, containing the quartile values, to the active dataset. For each case, the values of these variables are those for the employment category associated with the case.

Figure 15-1
Percentiles added to original data file as new variables

Using the spssdata Module

The `spssdata` module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python, builds on the functionality in the `Cursor` class to provide a number of features that simplify the task of working with case data.

- You can specify a set of variables to retrieve using variable names instead of index values, and you can use `VariableDict` objects created with the `spssaux` module to specify variable subsets.

- Once data have been retrieved, you can access case data by variable name.

- When reading case data, you can automatically skip over cases that have user- or system-missing values for any of the retrieved variables.

The `Spssdata` class provides four usage modes: read mode allows you to read cases from the active dataset, write mode allows you to add new variables (and their case values) to the active dataset, append mode allows you to append new cases to the active dataset, and new mode allows you to create an entirely new dataset (for users of IBM® SPSS® Statistics 16.0 or higher, it is

recommended to use the `Dataset` class to create a new dataset). To use the `Spssdata` class, you first create an instance of the class and store it to a Python variable, as in:

```
data = spssdata.Spssdata(accessType='w')
```

The optional argument *accessType* specifies the usage mode: read (`'r'`), write (`'w'`), append (`'a'`), or new (`'n'`). The default is read mode.

Notes

- For users of a 14.0.x version of the plug-in who are upgrading to version 15.0 or higher, read mode for the `Spssdata` class (for version 15.0 or higher) is equivalent to the `Spssdata` class provided with 14.0.x versions. No changes to your 14.0.x code for the `Spssdata` class are required to run the code with version 15.0 or higher.

- You can obtain general help for the `Spssdata` class by including the statement `help(spssdata.Spssdata)` in a program block, assuming you've already imported the `spssdata` module.

Reading Case Data with the Spssdata Class

To read case data with the `Spssdata` class, you create an instance of the class in read mode, as in:

```
data = spss.Spssdata(accessType='r')
```

Read mode is the default mode, so specifying `accessType='r'` is optional. For example, the above is equivalent to:

```
data = spss.Spssdata()
```

Invoking `Spssdata` without any arguments, as shown here, specifies that case data for all variables in the active dataset will be retrieved.

You can also call `Spssdata` with a set of variable names or variable index values, expressed as a Python list, a Python tuple, or a string. To illustrate this, consider the variables in *Employee data.sav* and an instance of `Spssdata` used to retrieve only the variables *salary* and *educ*. To create this instance from a set of variable names expressed as a tuple, use:

```
data = spssdata.Spssdata(indexes=('salary','educ'))
```

You can create the same instance from a set of variable index values using

```
data = spssdata.Spssdata(indexes=(5,3))
```

since the *salary* variable has an index value of 5 in the dataset, and the *educ* variable has an index value of 3. Remember that an index value of 0 corresponds to the first variable in file order.

You also have the option of calling `Spssdata` with a variable dictionary that's an instance of the `VariableDict` class from the `spssaux` module. Let's say you have such a dictionary stored to the variable *varDict*. You can create an instance of `Spssdata` for the variables in *varDict* with:

```
data = spssdata.Spssdata(indexes=(varDict,))
```

Example: Retrieving Data

Once you have created an instance of the `Spssdata` class, you can retrieve data one case at a time by iterating over the instance of `Spssdata`, as shown in this example:

```
*python_using_Spssdata_class.sps.
DATA LIST FREE /sku (A8) qty (F5.0).
BEGIN DATA
10056789 123
10044509 278
10046887 212
END DATA.
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata()
for row in data:
   print row.sku, row.qty
data.CClose()
END PROGRAM.
```

■ The `Spssdata` class has a built-in iterator that sequentially retrieves cases from the active dataset. Once you've created an instance of the class, you can loop through the case data simply by iterating over the instance. In the current example, the instance is stored in the Python variable *data* and the iteration is done with a `for` loop. The `Spssdata` class also supports the `fetchall` method from the `Cursor` class so that you can retrieve all cases with one call if that is more convenient, as in `data.fetchall()`.

Note: Be careful when using the `fetchall` method for large datasets, since Python holds the retrieved data in memory. In such cases, when you have finished processing the data, consider deleting the variable used to store it. For example, if the data are stored in the variable *allcases*, you can delete the variable with `del allcases`.

■ On each iteration of the loop, the variable *row* contains the data for a single case. You can access variable values within the case by variable name or variable index. In the current example, `row.sku` is the value of the variable *sku*, and `row.qty` is the value of the variable *qty* for the current case. Alternatively, using index values, `row[0]` gives the value of *sku* and `row[1]` gives the value of *qty*.

■ When you're finished with an instance of the `Spssdata` class, call the `CClose` method.

Result

```
10056789 123.0
10044509 278.0
10046887 212.0
```

Example: Skipping Over Cases with Missing Values

The `Spssdata` class provides the option of skipping over cases that have user- or system-missing values for any of the retrieved variables, as shown in this example. If you need to retrieve all cases but also check for the presence of missing values in the retrieved values, you can use the `hasmissing` and `ismissing` methods described in the next example.

```
*python_skip_missing.sps.
DATA LIST LIST (',') /numVar (f) stringVar (a4).
BEGIN DATA
0,a
1,b
,c
3,,
END DATA.
MISSING VALUES stringVar (' ') numVar(0).
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(omitmissing=True)
for row in data:
   print row.numVar, row.stringVar
data.CClose()
END PROGRAM.
```

■ The sample data in this example contain three cases with either user- or system-missing values. Cases 1 and 4 contain a user-missing value and case 3 contains a system-missing value.

■ The optional parameter `omitmissing`, to the `Spssdata` class, determines whether cases with missing values are read (the default) or skipped. Setting `omitmissing` to `True` specifies that cases with either user- or system-missing values are skipped when the data are read.

Result

```
1.0 b
```

Example: Identifying Cases and Variables with Missing Values

Sometimes you may need to read all of the data but take specific action when cases with missing values are read. The `Spssdata` class provides the `hasmissing` and `ismissing` methods for detecting missing values. The `hasmissing` method checks if any variable value in the current case is missing (user- or system-missing), and `ismissing` checks if a specified value is missing for a particular variable.

```
*python_check_missing.sps.
DATA LIST LIST (',') /numVar (f) stringVar (a4).
BEGIN DATA
0,a
1,b
,c
3,,
END DATA.
MISSING VALUES stringVar (' ') numVar(0).

BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(convertUserMissing=False)
# Compile and store missing value information for the variables
# in the current cursor.
data.makemvchecker()
# Loop through the cases in the active dataset.
for i,row in enumerate(data):
    # Check if the current case (row) has missing values.
    if data.hasmissing(row):
        print "Case: " + str(i+1)
        # Loop through the variables in the current cursor.
        for name in data.varnames():
            varvalue = row[data.getvarindex(name)]
            if varvalue==None:
                print "\tThe value for variable " + str(name) + \
                    " is system-missing."
            elif data.ismissing(name,varvalue):
                print "\tThe value for variable " + str(name) + \
                    " is user-missing."
data.CClose()
END PROGRAM.
```

- The sample data in this example contain three cases with either user- or system-missing values. Cases 1 and 4 contain a user-missing value and case 3 contains a system-missing value.

- `convertUserMissing=False` specifies that user-missing values are treated as valid data—that is, they are not converted to the Python data type *None*.

- The `makemvchecker` method from the `Spssdata` class gathers missing value information for all of the variables in the current cursor for use in checking for user- and system-missing values. This method must be called before calling either the `hasmissing` or `ismissing` methods from the `Spssdata` class. The results of the `makemvchecker` method are stored to a property of the current `Spssdata` instance and used when needed.

- For each case (row), `data.hasmissing(row)` returns *true* if the case contains a missing value.

- The `varnames` method from the `Spssdata` class returns a list of the variables whose values are being retrieved for the current cursor.

- The `getvarindex` method from the `Spssdata` class returns the index number (in the current cursor) of the specified variable.

- The `ismissing` method returns *true* if the specified value is missing for the specified variable. Since `if varvalue==None` will identify system-missing values, user-missing values, in this case, are identified by a return value of *true* from `ismissing`.

Result

```
Case: 1
    The value for variable numVar is user-missing.
Case: 3
    The value for variable numVar is system-missing.
Case: 4
    The value for variable stringVar is user-missing.
```

Example: Handling Data with Splits

When reading from datasets with splits, you may want to know when a split boundary has been crossed. Detecting split changes is necessary when you're creating custom pivot tables from data with splits and want separate results to be displayed for each split group. In this example, we simply count the number of cases in each split group.

```
*python_Spssdata_split_change.sps.
DATA LIST LIST (',') /salary (F) jobcat (F).
BEGIN DATA
21450,1
45000,1
30750,2
103750,3
57000,3
72500,3
END DATA.

SORT CASES BY jobcat.
SPLIT FILE BY jobcat.

BEGIN PROGRAM.
import spss, spssdata
data=spssdata.Spssdata()
counts=[]
first = True
for row in data:
   if data.IsStartSplit():
      if first:
         first = False
      else:
         counts.append(splitcount)
      splitcount=1
   else:
      splitcount+=1
data.CClose()
counts.append(splitcount)
print counts
END PROGRAM.
```

- The built-in iterator for the Spssdata class iterates over all of the cases in the active dataset, whether splits are present or not.

- Use the IsStartSplit method from the Spssdata class to detect a split change. It returns a Boolean value—*true* if the current case is the first case of a new split group and *false* otherwise.

- In the current example, the Python variable *counts* is a list of the case counts for each split group. It is updated with the count from the previous split once the first case of the next split is detected.

Handling IBM SPSS Statistics Datetime Values

Dates and times in IBM® SPSS® Statistics are represented internally as the number of seconds from October 14, 1582, which means that data retrieved from SPSS Statistics for a datetime variable is returned as this number of seconds. SPSS Statistics knows how to correctly interpret this number when performing datetime calculations and displaying datetime values, but without special instructions, Python doesn't. Fortunately, the `Spssdata` class will do the necessary transformations for you and convert a datetime value into a Python datetime object, which will render in a recognizable date format and can be manipulated with functions from the Python datetime module (a built-in module distributed with Python).

To convert values from a datetime variable to a Python datetime object, you specify the variable name in the argument *cvtDates* to the `Spssdata` class (in addition to specifying it in *indexes*), as shown in this example:

```
*python_convert_datetime_values_spssdata.sps.
DATA LIST FREE /bdate (ADATE10).
BEGIN DATA
02/13/2006
END DATA.
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(indexes=('bdate',), cvtDates=('bdate',))
row=data.fetchone()
print row[0]
data.CClose()
END PROGRAM.
```

- The argument *cvtDates* can be a list, a tuple, an instance of the `VariableDict` class from the `spssaux` module, or the name "ALL." A tuple containing a single element is denoted by following the value with a comma, as shown here. If a variable specified in *cvtDates* does not have a date format, it is not converted.

- The `Spssdata` class supports the `fetchone` method from the `Cursor` class, which is used here to retrieve the single case in the active dataset. For reference, it also supports the `fetchall` method from the `Cursor` class.

- The result from Python is `2006-02-13 00:00:00`, which is the display of a Python datetime object.

Creating New Variables with the Spssdata Class

To add new variables to the active dataset using the `Spssdata` class, you create an instance of the class in write mode, as in:

```
data = spss.Spssdata(accessType='w')
```

Like the `Cursor` class, write mode for the `Spssdata` class supports the functionality available in read mode. For example, you can create a write cursor that also allows you to retrieve case data for a subset of variables—perhaps those variables used to determine case values for the new variables, as in:

```
data = spss.Spssdata(indexes=('salary','educ'),accessType='w')
```

For more information, see the topic "Reading Case Data with the Spssdata Class" on p. 246.

Write mode also supports multiple data passes, allowing you to add new variables on any data pass. For more information, see the example on "Adding Group Percentile Values to a Dataset with the Spssdata Class" on p. 257.

Example

```
*python_Spssdata_add_vars.sps.
DATA LIST FREE /var1 (F) var2 (A2)  var3 (F).
BEGIN DATA
11 ab 13
21 cd 23
31 ef 33
END DATA.
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(accessType='w')
data.append(spssdata.vdef('var4',
            vlabel='Sample numeric variable',vfmt=["F",2,0]))
data.append(spssdata.vdef('strvar',
            vlabel='Sample string variable',vtype=8))
data.commitdict()
for i,row in enumerate(data):
   data.casevalues([4+10*(i+1),'row' + str(i+1)])
data.CClose()
END PROGRAM.
```

- The `append` method from the `Spssdata` class is used to add the specifications for a new variable. The argument to `append` is a tuple of attributes that specifies the variable's properties, such as the variable name, the variable type, the variable label, and so forth. You use the `vdef` function in the `spssdata` module to generate a suitable tuple. `vdef` requires the variable name, specified as a string, and an optional set of arguments that specify the variable properties. The available arguments are: *vtype*, *vlabel*, *vmeasurelevel*, *vfmt*, *valuelabels*, *missingvalues*, and *attrib*.

  String variables are specified with a value of *vtype* equal to the defined length of the string (maximum of 32767), as in `vtype=8` for a string of length 8. If *vtype* is omitted, *vfmt* is used to determine the variable type. If both *vtype* and *vfmt* are omitted, the variable is assumed to be numeric. Numeric variables can be explicitly specified with a value of 0 for *vtype*.

  For more information on using the `vdef` function to specify variable properties, include the statement `help(spssdata.vdef)` in a program block once you've imported the `spssdata` module. Examples of specifying missing values, value labels, and variable attributes are provided in the sections that follow.

- Once specifications for the new variables have been added with the `append` method, the `commitdict` method is called to create the new variables.

- The `casevalues` method is used to assign the values of new variables for the current case. The argument is a sequence of values, one for each new variable, in the order appended. If the sequence is shorter than the number of variables, the omitted variables will have the system-missing value.

Note: You can also use the `setvalue` method to set the value of a specified variable for the current case. For more information, include the statement `help(spssdata.Spssdata.setvalue)` in a program block.

■ The `CClose` method closes the cursor. Changes to the active dataset don't take effect until the cursor is closed.

Note: You cannot add new variables to an empty dataset using write mode from the `Spssdata` class. If you need to create a dataset from scratch and you are a user of IBM® SPSS® Statistics 15.0, use the mode `'n'` of the `Spssdata` class. For users of SPSS Statistics 16.0 and higher, it is recommended to use the `Dataset` class to create a new dataset. For more information, see the topic "Example: Creating and Saving Datasets" in Chapter 16 on p. 268.

Specifying Missing Values for New Variables

User missing values for new variables are specified with the *missingvalues* argument to the `vdef` function.

```
*python_Spssdata_define_missing.sps.
DATA LIST FREE /var1 (F).
BEGIN DATA
1
END DATA.
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(accessType='w')
data.append(spssdata.vdef('var2',missingvalues=[0]))
data.append(spssdata.vdef('var3',
            missingvalues=[spssdata.spsslow,"THRU",0]))
data.append(spssdata.vdef('var4',
            missingvalues=[9,"THRU",spssdata.spsshigh,0]))
data.append(spssdata.vdef('var5',vtype=2,missingvalues=[' ','NA']))
data.commitdict()
data.CClose()
END PROGRAM.
```

■ Three numeric variables (*var2*, *var3*, and *var4*) and one string variable (*var5*) are created. String variables are specified by a value of *vtype* greater than zero and equal to the defined width of the string (*vtype* can be omitted for numeric variables).

■ To specify a discrete set of missing values, provide the values as a list or tuple, as shown for the variables *var2* and *var5* in this example. You can specify up to three discrete missing values.

■ To specify a range of missing values (for a numeric variable), set the first element of the list to the low end of the range, the second element to the string `'THRU'` (use upper case), and the third element to the high end of the range, as shown for the variable *var3*. The global variables *spsslow* and *spsshigh* in the `spssdata` module contain the values SPSS Statistics uses for `LO (LOWEST)` and `HI (HIGHEST)`, respectively.

■ To include a single discrete value along with a range of missing values, use the first three elements of the missing value list to specify the range (as done for *var3*) and the fourth element to specify the discrete value, as shown for the variable *var4*.

Optionally, you can provide the missing value specification in the same form as that returned by the `GetVarMissingValues` function from the `spss` module—a tuple of four elements where the first element specifies the missing value type (0 for discrete values, 1 for a range, and 2 for a range and a single discrete value) and the remaining three elements specify the missing values.

The following code illustrates this approach for the same variables and missing values used in the previous example:

```
DATA LIST FREE /var1 (F).
BEGIN DATA
1
END DATA.
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(accessType='w')
data.append(spssdata.vdef('var2',missingvalues=[0,0,None,None]))
data.append(spssdata.vdef('var3',
           missingvalues=[1,spssdata.spsslow,0,None]))
data.append(spssdata.vdef('var4',
           missingvalues=[2,9,spssdata.spsshigh,0]))
data.append(spssdata.vdef('var5',
           vtype=2,missingvalues=[0,' ','NA',None]))
data.commitdict()
data.CClose()
END PROGRAM.
```

The Python data type *None* is used to specify unused elements of the 4-tuple. For more information on the `GetVarMissingValues` function, see "Retrieving Definitions of User-Missing Values" on p. 218.

Defining Value Labels for New Variables

Value labels are specified with the *valuelabels* argument to the `vdef` function.

```
*python_Spssdata_define_vallabels.sps.
DATA LIST FREE /var1 (F).
BEGIN DATA
1
END DATA.
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(accessType='w')
data.append(spssdata.vdef('var2',valuelabels={0:"No",1:"Yes"}))
data.append(spssdata.vdef('var3',
           vtype=1,valuelabels={"f":"female","m":"male"}))
data.commitdict()
data.CClose()
END PROGRAM.
```

- The argument *valuelabels* is specified as a Python dictionary. Each key in the dictionary is a value with an assigned label, and the value associated with the key is the label.

- Values for string variables—`"f"` and `"m"` in this example—must be quoted. String variables are specified by a value of *vtype* greater than zero and equal to the defined length of the string.

Defining Variable Attributes for New Variables

Variable attributes are specified with the *attrib* argument to the `vdef` function.

```
*python_Spssdata_define_varattributes.sps.
DATA LIST FREE /var1 (F).
BEGIN DATA
1
END DATA.
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(accessType='w')
data.append(spssdata.vdef('minority',
          attrib={"demographicVars":"1","binary":"Yes"}))
data.commitdict()
data.CClose()
END PROGRAM.
```

- The argument *attrib* is specified as a Python dictionary. Each key in the dictionary is the name of a new variable attribute, and the value associated with the key is the attribute value, specified as a string.

- The new variable *minority* is created, having the attributes *demographicVars* and *binary*. The value of *demographicVars* is "1" and the value of *binary* is "Yes".

Setting Values for Date Format Variables

In SPSS Statistics, dates are stored internally as the number of seconds from October 14, 1582. When setting values for new date format variables, you'll need to provide the value as the appropriate number of seconds. The spssdata module provides the yrmodasec function for converting from a date (represented as a four-digit year, month, and day) to the equivalent number of seconds.

```
*python_set_date_var.sps.
DATA LIST FREE /case (F).
BEGIN DATA
1
END DATA.
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(accessType='w')
data.append(spssdata.vdef('date',vfmt=["ADATE",10]))
data.commitdict()
data.fetchone()
data.casevalues([spssdata.yrmodasec([2006,10,20])])
data.CClose()
END PROGRAM.
```

- The vdef function from the Spssdata class is used to specify the properties for a new date format variable called *date*. The format is specified as ADATE (American date) with a width of 10 characters.

- The append method adds the specifications for this new variable and commitdict creates the variable.

- The fetchone method, available with the Spssdata class, sets the record pointer to the first case.

- The casevalues method is used to set the value of *date* for the first case, using the value returned by the yrmodasec method. The argument to yrmodasec is a three-element sequence consisting of the four-digit year, the month, and the day.

Note: The `SetValueNumeric` method in the `Cursor` class provides automatic conversion from a Python datetime object to the equivalent value in SPSS Statistics. For more information, see the topic "Creating New Variables with the Cursor Class" on p. 240.

Appending New Cases with the Spssdata Class

To append new cases to the active dataset with the `Spssdata` class, you create an instance of the class in append mode, as in:

```
data = spss.Spssdata(accessType='a')
```

Example

```
*python_Spssdata_add_cases.sps.
DATA LIST FREE /case (F) value (A1).
BEGIN DATA
1 a
END DATA.
BEGIN PROGRAM.
import spssdata
data=spssdata.Spssdata(accessType='a')
data.appendvalue('case',2)
data.appendvalue('value','b')
data.CommitCase()
data.appendvalue('case',3)
data.appendvalue('value','c')
data.CommitCase()
data.CClose()
END PROGRAM.
```

- Case values are set using the `appendvalue` method from the `Spssdata` class. The first argument is the variable name, as a string, and the second argument is the value for the current case. A numeric variable whose value is not specified is set to the system-missing value, whereas a string variable whose value is not specified will have a blank value. You can also use the variable index instead of the variable name. Variable index values represent position in the active dataset, starting with 0 for the first variable in file order.

- The `CommitCase` method must be called to commit the values for each new case. Changes to the active dataset take effect when the cursor is closed. When working in append mode, the cursor is ready to accept values for a new case (using `appendvalue`) once `CommitCase` has been called for the previous case.

- When working in append mode with the `Spssdata` class, the `CClose` method must be used to close the cursor.

Note: Append mode does not support reading case data or creating new variables. A dataset must contain at least one variable in order to append cases to it, but it need not contain any cases. If you need to create a dataset from scratch and you are a user of IBM® SPSS® Statistics 15.0, use the mode `'n'` of the `Spssdata` class. For users of SPSS Statistics 16.0 and higher, it is recommended to use the `Dataset` class to create a new dataset. For more information, see the topic "Example: Creating and Saving Datasets" in Chapter 16 on p. 268.

Example: Adding Group Percentile Values to a Dataset with the Spssdata Class

This example is a reworking of the code for "Adding Group Percentile Values to a Dataset" on p. 243, but using the `Spssdata` class. The example calculates the quartiles for the cases associated with each value of a grouping variable—in particular, the quartiles for *salary* grouped by *jobcat* for the *Employee data.sav* dataset—and adds the results as new variables. This involves two passes of the data. The first pass reads the data and calculates the group quartiles. The second pass adds the quartile values as new variables to the active dataset.

```
*python_Spssdata_add_group_percentiles.sps.
BEGIN PROGRAM.
import spss, spssdata, math
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")

# Create a cursor that will only read the values of jobcat and salary
data=spssdata.Spssdata(indexes=['jobcat','salary'],accessType='w')

# Accumulate frequencies of salaries for each employment category
salaries={}; counts={}
for row in data:
   salaries[row.jobcat]=salaries.get(row.jobcat,{})
   salaries[row.jobcat][row.salary]= \
      salaries[row.jobcat].get(row.salary,0) + 1
   counts[row.jobcat]=counts.get(row.jobcat,0) + 1

# Calculate the cutpoint salary value for each percentile for each
# employment category
percentiles={}
for jobcat in salaries:
   cutpoints = [int(math.ceil(counts[jobcat]*f)) for f in [.25, .50, .75]]
   tempcount=0; pctindex=0
   percentiles[jobcat]=[]
   salarylist=sorted(salaries[jobcat].keys())
   for salary in salarylist:
      tempcount+=salaries[jobcat][salary]
      if tempcount>=cutpoints[pctindex]:
         percentiles[jobcat].append(salary)
         pctindex+=1
         if pctindex == 3:
            break

# Create and populate new variables for the percentiles
data.restart()
data.append(spssdata.vdef('salary_25'))
data.append(spssdata.vdef('salary_50'))
data.append(spssdata.vdef('salary_75'))
data.commitdict()
for row in data:
   data.setvalue('salary_25',percentiles[row.jobcat][0])
   data.setvalue('salary_50',percentiles[row.jobcat][1])
   data.setvalue('salary_75',percentiles[row.jobcat][2])
   data.CommitCase()
data.CClose()
end program.
```

- `spssdata.Spssdata(indexes=['jobcat','salary'],accessType='w')` creates a write cursor that also allows you to retrieve case data for the two variables *jobcat* and *salary*.

- Aside from the changes introduced by using the `Spssdata` class, the algorithm is unchanged from the version that uses the `Cursor` class. For more information, see the topic "Example: Adding Group Percentile Values to a Dataset" on p. 243.

- Once the quartile values are determined, the `restart` method from the `Spssdata` class is called to reset the write cursor in preparation for another data pass. `restart` needs to be called before creating new variables on subsequent data passes.

- Specifications for the three new variables *salary_25*, *salary_50*, and *salary_75* are set with the `append` method from the `Spssdata` class. The `commitdict` method is called to create the new variables. For more information, see the topic "Creating New Variables with the Spssdata Class" on p. 251.

- Case values are set using the `setvalue` method from the `Spssdata` class. The first argument to `setvalue` is the variable name and the second argument is the associated value for the current case. For each case, the values of *salary_25*, *salary_50*, and *salary_75* are those for the employment category associated with the case. When `setvalue` is used, you must call the `CommitCase` method to commit the changes for each case.

Note

In the case of multiple data passes where you need to add variables on a data pass other than the first (as in this example), you may need to allocate the buffer size (in bytes) for the new variables, using the optional argument *maxaddbuffer* to the `Spssdata` class. By default, *maxaddbuffer* is set to 80 bytes, which is sufficiently large to accommodate 10 numeric variables, and thus large enough to handle the three new numeric variables created in this example. In the case where you are only adding variables on the first data pass, the buffer allocation is done automatically for you. The following rules specify the buffer sizes needed for numeric and string variables:

- Each numeric variable requires eight bytes.

- Each string variable requires a size that is an integer multiple of eight bytes, and large enough to store the defined length of the string (one byte per character). For example, you would allocate eight bytes for strings of length 1–8 and 16 bytes for strings of length 9–16.

Example: Generating Simulated Data

It is often necessary (or convenient) to generate data files in order to test the variability of results, bootstrap statistics, or work on code development before the actual data file is available. The following Python user-defined function creates a new dataset containing a set of simulated performance ratings given by each member of a work group to every other member of the group.

Note: For users of IBM® SPSS® Statistics 16.0 and higher, it is recommended to use the `Dataset` class to create a new dataset. For more information, see the topic "Example: Creating and Saving Datasets" in Chapter 16 on p. 268.

```
def GenerateData(ngroups,nmembers,maxrating):
    """Generate simulated performance rating data given by each member
    of a work group to each other member of the group.
    ngroups is the number of groups.
    nmembers is the number of members in each group.
    maxrating is the maximum performance rating.
    """
    cur = spssdata.Spssdata(accessType='n')
    cur.append(spssdata.vdef("group",vfmt=["F",2,0]))
    cur.append(spssdata.vdef("rater",vfmt=["F",2,0]))
    cur.append(spssdata.vdef("ratee",vfmt=["F",2,0]))
    cur.append(spssdata.vdef("rating",vfmt=["F",2,0]))
    cur.commitdict()
    for group in range(1,ngroups+1):
        for rater in range(1,nmembers+1):
            for ratee in range(1,rater) + range(rater+1,nmembers+1):
                cur.appendvalue("group",group)
                cur.appendvalue("rater",rater)
                cur.appendvalue("ratee",ratee)
                cur.appendvalue("rating",
                               round(random.uniform(0,maxrating) + 0.5))
                cur.CommitCase()
    cur.CClose()
```

- `GenerateData` is a Python user-defined function that requires three arguments that define the generated dataset.

- To create a new dataset, you use the new mode (`accessType='n'`) of the `Spssdata` class.

- Specifications for the variables in the new dataset are set with the `append` method from the `Spssdata` class. The `commitdict` method is called to create the new variables. For more information, see the topic "Creating New Variables with the Spssdata Class" on p. 251.

- Case values are set using the `appendvalue` method from the `Spssdata` class. For more information, see the topic "Appending New Cases with the Spssdata Class" on p. 256.

- Each new case contains the rating given to one group member (the *ratee*) by another group member (the *rater*), as well as identifiers for the group, the group member providing the rating, and the group member being rated. Ratings are integers from 1 to *maxrating* with each integer having an equal probability. The rating formula makes use of the `uniform` function from the `random` module, a standard module provided with Python. The Python module that contains the `GenerateData` function includes a statement to import the `random` module. Of course, any appropriate distribution formula could be used instead of the uniform distribution used here.

- The `CommitCase` method must be called to commit the values for each new case. Changes to the active dataset take effect when the cursor is closed. The cursor is ready to accept values for a new case (using `appendvalue`) once `CommitCase` has been called for the previous case.

- When creating a new dataset with the `Spssdata` class, the `CClose` method must be used to close the cursor.

Example

As an example, generate a sample dataset for 10 groups with 6 members each and a maximum score of 7.
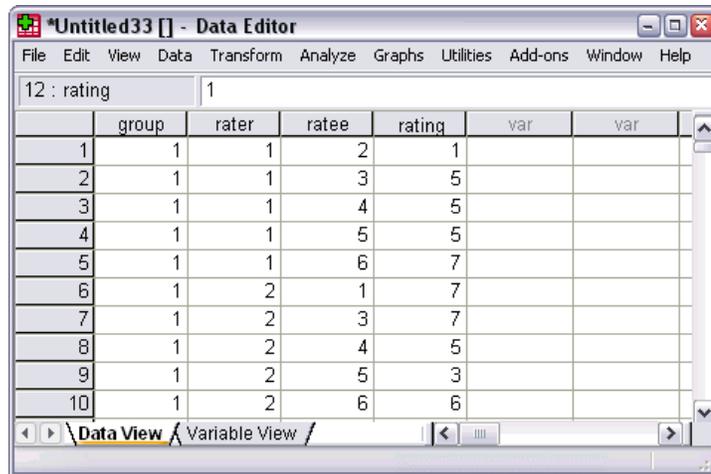
```
*python_generate_data.sps.
BEGIN PROGRAM.
import samplelib
samplelib.GenerateData(10,6,7)
END PROGRAM.
```

The BEGIN PROGRAM block starts with a statement to import the sesamplelib module, which contains the definition for the GenerateData function.

Note: To run this program block, copy the module file *samplelib.py* from the */examples/python* folder, in the accompanying examples, to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

Figure 15-2
Resulting dataset

	group	rater	ratee	rating	var	var
1	1	1	2	1		
2	1	1	3	5		
3	1	1	4	5		
4	1	1	5	5		
5	1	1	6	7		
6	1	2	1	7		
7	1	2	3	7		
8	1	2	4	5		
9	1	2	5	3		
10	1	2	6	6		

Untitled33 [] - Data Editor

File Edit View Data Transform Analyze Graphs Utilities Add-ons Window Help

12 : rating   1

Data View / Variable View

Creating and Accessing Multiple Datasets

The IBM® SPSS® Statistics - Integration Plug-In for Python provides the ability to create new datasets and concurrently access multiple datasets. This is accomplished using the `Dataset` class, available with the `spss` module. You can create one or more new datasets from existing datasets, combining the data from the existing datasets in any way you choose, and you can concurrently read from or modify case values and variable properties of multiple datasets without having to explicitly activate each one.

   `Dataset` objects are available within data steps. Data steps set up the environment that allows you to create new datasets and access multiple datasets. Data steps cannot be initiated if there are pending transformations. If you need to access case data while maintaining pending transformations, use the `Cursor` class. For more information, see the topic "Working with Case Data in the Active Dataset" in Chapter 15 on p. 233.

Getting Started with the Dataset Class

To create a `Dataset` object, you must first initiate a data step. This is best done with the `DataStep` class (introduced in version 17.0), which will execute any pending transformations prior to starting the data step. The `DataStep` class is designed to be used with the Python `with` statement, as in the following:

```
from __future__ import with_statement
import spss
with spss.DataStep():
   <actions to perform within the data step>
```

■   The code `from __future__ import with_statement` makes the Python `with` statement available and must be the first statement in the program.

■   `with spss.DataStep():` initiates a block of code associated with a data step. The data step is implicitly started after executing any pending transformations. All code associated with the data step should reside in the block. When the block completes, the data step is implicitly ended, even if an exception prevents the block from completing normally.

Once a data step has been initiated, you create a `Dataset` object for an open dataset by specifying the name of the dataset, as in:

```
dataset = spss.Dataset(name="DataSet1")
```

The Python variable *dataset* is an instance of the `Dataset` class and provides access to the case data and variables in the dataset named *DataSet1*.

- Specifying `"*"` for the *name* argument or omitting the argument creates a `Dataset` object for the active dataset.

- Specifying the Python data type *None* or the empty string `''` for the *name* argument creates a new dataset and an associated `Dataset` object that will allow you to populate the dataset. The name of the new dataset is automatically generated by IBM® SPSS® Statistics and can be retrieved from the `name` property of the `Dataset` object. For more information, see the topic "Example: Creating and Saving Datasets" on p. 268.

Note: An instance of the `Dataset` class cannot be used outside of the data step in which it was created.

Accessing, Adding, or Deleting Variables

Access to the variables in a dataset, including the ability to add new variables or delete existing ones, is provided by the `VariableList` object associated with the `Dataset` object. You obtain the `VariableList` object from the `varlist` property of the `Dataset` object, as in:

```
variableList = dataset.varlist
```

Note: A `VariableList` object is not a Python list but has some of the functionality of a Python list, such as the ability to append and insert elements, the ability to iterate over the elements, and the support of the Python `len` function. An instance of the `VariableList` class cannot be used outside of the data step in which it was created.

Getting or Setting Variable Properties

From the `VariableList` object you can access any existing variable, allowing you to retrieve or modify any property of the variable, such as the measurement level or custom attributes. To access a variable from the `VariableList` object, you can specify the variable name, as in:

```
variable = variableList['salary']
```

Python is case sensitive, so the variable name must match the case as specified when the variable was defined in IBM® SPSS® Statistics. You can also specify the variable by its index value, as in

```
variable = variableList[5]
```

which accesses the variable with index 5 in the dataset. Index values represent position in the dataset, starting with 0 for the first variable in file order. The Python variable *variable* is an instance of the `Variable` class. Properties of the `Variable` class allow you to get or set properties of the associated variable. For example,

```
varLevel = variable.measurementLevel
```

gets the measurement level and stores it to the Python variable *varLevel*, whereas

```
variable.measurementLevel = 'ordinal'
```

sets the measurement level. For a complete list of available variable properties, see the topic on the `Variable` class in the SPSS Statistics Help system.

Looping through the variables in an instance of VariableList. You can iterate over an instance of the `VariableList` class, allowing you to loop through the variables in the associated dataset, in file order. For example:

```
for var in dataset.varlist:
   print var.name
```

- On each iteration of the loop, *var* is an instance of the `Variable` class, representing a particular variable in the `VariableList` instance.

The number of variables in a `VariableList` instance, which is also the number of variables in the associated dataset, is available using the Python `len` function, as in:

```
len(variableList)
```

Adding Variables

You can add a new variable to the dataset using either the `append` or `insert` method of the `VariableList` object. The variable is added to the `VariableList` object as well as the associated dataset.

Appending a variable. The `append` method adds a new variable to the end of the variable list. For example, the following appends a numeric variable named *newvar1*:

```
variableList.append(name='newvar1',type=0)
```

- The arguments to the `append` method are the name of the new variable and the variable type—0 for numeric and an integer equal to the defined length (maximum of 32767) for a string variable. The variable type is optional and defaults to numeric.

Inserting a variable. The `insert` method adds a new variable at a specified position in the variable list. For example, the following inserts a string variable of length 10 named *newvar2* at position 3 in the variable list:

```
variableList.insert(name='newvar2',type=10,index=3)
```

- The arguments to the `insert` method are the name of the new variable, the variable type (as described for the `append` method), and the index position at which to insert the variable. The variable type is optional and defaults to numeric. The index position is optional—by default, the variable is appended to the end of the list. When the variable is inserted within the variable list, the index of the variable it replaces is incremented by 1, as are the indexes of the variables that follow it in the list.

Deleting Variables

You can delete a specified variable from the `VariableList` instance, which results in deleting it from the associated dataset. The variable to be deleted can be specified by name or index. For example:

```
del variableList['salary']
```

or

```
del variableList[5]
```

Retrieving, Modifying, Adding, or Deleting Cases

Access to the cases in a dataset, including the ability to add new cases or delete existing ones, is provided by the `CaseList` object associated with the `Dataset` object. You obtain the `CaseList` object from the `cases` property of the `Dataset` object, as in:

```
caseList = dataset.cases
```

Note: A `CaseList` object is not a Python list but has some of the functionality of a Python list, such as the ability to append and insert elements, the ability to iterate over the elements, and the support of the Python `len` function. An instance of the `CaseList` class cannot be used outside of the data step in which it was created.

You can loop through the cases in an instance of the `CaseList` class. For example:

```
for row in dataset.cases:
    print row
```

- On each iteration of the loop, *row* is a case from the associated dataset.
- Case values are returned as a list where each element of the list is the value of the associated variable, in file order.

The number of cases in a `CaseList` instance, which is also the number of cases in the associated dataset, is available using the `len` function, as in:

```
len(caseList)
```

Retrieving Case Values

From the `CaseList` object, you can retrieve a specific case or a range of cases, and you can limit the retrieval to a specified variable or a range of variables within those cases.

- System-missing values are returned as the Python data type *None*.
- By default, values of variables with date or datetime formats are returned as floating point numbers representing the number of seconds from October 14, 1582. You can specify to convert values of those variables to Python datetime objects with the optional *cvtDates* argument to the `Dataset` class. The retrieved datetime value will then render in a recognizable date format within Python and can be manipulated with functions from the Python datetime module (a built-in module distributed with Python).
- Values of variables with `TIME` and `DTIME` formats are returned as floating point numbers representing the number of seconds in the time interval.
- The `CaseList` class does not provide any special handling for datasets with split groups—it simply returns all cases in the dataset. If you need to differentiate the data in separate split groups, consider using the `Cursor` class to read your data, or you may want to use the `spss.GetSplitVariableNames` function to manually process the split groups.

Retrieving a single case. Specific cases are retrieved from the `CaseList` object by specifying the case number, starting with 0, as in:

```
oneCase = caseList[0]
```

Referencing a case number beyond the last one in the dataset raises an exception.

Retrieving a single value within a case. You can get the value for a single variable within a case by specifying the case number and the index of the variable (index values represent position in the dataset, starting with 0 for the first variable in file order). The following gets the value of the variable with index 1 for case number 0.

```
oneValue = caseList[0,1]
```

The result is returned as a list with a single element.

Retrieving a Range of Values

You can use the Python slice notation to specify ranges of cases and ranges of variables within a case. Values for multiple cases are returned as a list of elements, each of which is a list of values for a single case.

Retrieve the values for a range of cases. The following retrieves the values for cases 0, 1, and 2.

```
data = caseList[0:3]
```

Each element of the resulting list is a list containing the variable values for a single case, as in the following for a dataset with two numeric variables:

```
[[51.0, 73.0], [46.0, 123.0], [38.0, 18.0]]
```

Retrieve the values for a range of variables within a specific case. The following retrieves the values for variables with index values 0, 1, and 2 for case number 0.

```
data = caseList[0,0:3]
```

The result is a list containing the case values of the specified variables, as in:

```
[15.0, 37.0, 120.0]
```

Retrieve the values of a specific variable for a range of cases. The following retrieves the values of the variable with index value 1 for case numbers 0, 1, and 2.

```
data = caseList[0:3,1]
```

Each element of the resulting list is a one element list containing the value of the specified variable for a single case, as in:

```
[[37.0], [250.0], [180.0]]
```

Retrieve the values for a range of variables and for a range of cases. The following retrieves the values for the variables with index values 1, 2, and 3 for case numbers 4, 5, and 6.

```
data = caseList[4:7,1:4]
```

Each element of the resulting list is a list containing the values of the specified variables for a single case, as in:

```
[[312.0, 144.0, 349.0], [529.0, 178.0, 441.0], [544.0, 120.0, 455.0]]
```

Negative index values. Case indexing supports the use of negative indices, both for the case number and the variable index. The following gets the value of the second to last variable (in file order) for the last case.

```
value = caseList[-1,-2]
```

Modifying Case Values

From the `CaseList` object, you can modify the values for a specific case or a range of cases, and you can set the value of a particular variable or a range of variables within those cases.

- The specified values can be numeric or string values and must match the variable type of the associated variable. Values of *None* are converted to system-missing for numeric variables and blanks for string variables. Unicode and string values are converted according to the current mode of the IBM® SPSS® Statistics processor (Unicode mode or code page mode).

- Values of numeric variables with a date or datetime format should be specified as Python `time.struct_time` or `datetime.datetime` objects, which are then converted to the appropriate SPSS Statistics value. Values of variables with TIME and DTIME formats should be specified as the number of seconds in the time interval.

Setting values for a single case. Values for a single case are provided as a list or tuple of values. The first element corresponds to the first variable in file order, the second element corresponds to the second variable in file order, and so on. Case numbers start from 0. The following sets the values for the case with index 1—the second case in the dataset—for a dataset with eight numeric variables.

```
caseList[1] = [35,150,100,2110,19,2006,3,4]
```

Setting a single value within a case. You can set the value for a single variable within a case by specifying the case number and the index of the variable (index values represent position in the dataset, starting with 0 for the first variable in file order). The following sets the value of the variable with index 0 for case number 12 (case numbers start from 0).

```
caseList[12,0] = 14
```

Setting a Range of Values

You can use the Python slice notation to specify a range of cases and a range of variables within a case. Values for multiple cases are specified as a list or tuple of elements, each of which is a list or tuple of values for a single case.

Set the values for a range of cases. The following sets the values for cases 0, 1, and 2 for a dataset with four variables, the second of which is a string variable and the rest of which are numeric variables:

```
caseList[0:3] = ([172,'m',27,34500],[67,'f',32,32500],[121,'f',37,23000])
```

Set the values for a range of variables within a specific case. The following sets the values for the variables with index values 5, 6, and 7 for case number 34.

```
caseList[34,5:8] = [65,2,41]
```

Set the values of a specific variable for a range of cases. The following sets the values of the variable with index value 5 for case numbers 0, 1, and 2:

```
caseList[0:3,5] = [65,22,17]
```

Set the values for a range of variables and for a range of cases. The following sets the values for the variables with index values 5 and 6 for case numbers 4, 5, and 6.

```
caseList[4:7,5:7] = ([61,17],[25,32],[45,21])
```

Negative index values. Case indexing supports the use of negative indices, both for the case number and the variable index. The following specifies the value of the second to last variable (in file order) for the last case.

```
caseList[-1,-2] = 8
```

Adding or Deleting Cases

From the `CaseList` object, you can add or delete cases.

Adding Cases

You can add a new case to the dataset using either the `append` or `insert` method of the `CaseList` object. The new case is added to the `CaseList` object as well as the associated dataset.

■ The specified case values can be numeric or string values and must match the variable type of the associated variable. Values of *None* are converted to system-missing for numeric variables and blanks for string variables.

■ Values of numeric variables with a date or datetime format should be specified as Python `time.struct_time` or `datetime.datetime` objects, which are then converted to the appropriate IBM® SPSS® Statistics value. Values of variables with `TIME` and `DTIME` formats should be specified as the number of seconds in the time interval.

Appending a case. The `append` method adds a new case to the end of the dataset. For example, the following appends a case to a dataset consisting of a single numeric variable and a single string variable:

```
caseList.append([2,'b'])
```

■ The argument to the `append` method is a tuple or list specifying the case values. The first element in the tuple or list is the value for the first variable in file order, the second is the value of the second variable in file order, and so on.

Inserting a case. The `insert` method adds a new case at a specified position in the dataset. For example, the following inserts a case at case number 1 (case numbers start from 0) into a dataset consisting of a single numeric variable and a single string variable:

```
caseList.insert([2,'b'],1)
```

■ The first argument to the `insert` method is a tuple or list specifying the case values, as described for the `append` method. The second argument specifies the position at which the case is inserted and is optional—by default, the new case is appended to the end of the dataset.

Deleting Cases

You can delete a specified case from the `CaseList` object, which results in deleting that case from the associated dataset. Cases are specified by case number, starting with 0 for the first case in the dataset. For example:

```
del caseList[0]
```

Example: Creating and Saving Datasets

When creating new datasets that you intend to save, you'll want to keep track of the dataset names since the save operation is done outside of the associated data step. In this example, we split a dataset into separate datasets—one new dataset for each value of a particular variable. The new datasets are then saved to the file system.

```
*python_dataset_save.sps.
DATA LIST FREE /dept (F2) empid (F4) salary (F6).
BEGIN DATA
7   57   57000
5   23   40200
3   62   21450
3   18   21900
5   21   45000
5   29   32100
7   38   36000
3   42   21900
7   11   27900
END DATA.
DATASET NAME saldata.
SORT CASES BY dept.
BEGIN PROGRAM.
from __future__ import with_statement
import spss
with spss.DataStep():
    ds = spss.Dataset()
    # Create a new dataset for each value of the variable 'dept'
    newds = spss.Dataset(name=None)
    newds.varlist.append('dept')
    newds.varlist.append('empid')
    newds.varlist.append('salary')
    dept = ds.cases[0,0][0]
    dsNames = {newds.name:dept}
    for row in ds.cases:
        if (row[0] != dept):
            newds = spss.Dataset(name=None)
            newds.varlist.append('dept')
            newds.varlist.append('empid')
            newds.varlist.append('salary')
            dept = row[0]
            dsNames[newds.name] = dept
        newds.cases.append(row)
# Save the new datasets
for name,dept in dsNames.iteritems():
    strdept = str(dept)
    spss.Submit(r"""
    DATASET ACTIVATE %(name)s.
    SAVE OUTFILE='/mydata/saldata_%(strdept)s.sav'.
    """ %locals())
spss.Submit(r"""
DATASET ACTIVATE saldata.
DATASET CLOSE ALL.
""" %locals())
END PROGRAM.
```

- `with spss.DataStep():` initiates a block of code associated with a data step. The data step is implicitly started after executing any pending transformations. All code associated with the data step should reside in the `with` block as shown here. When the block completes, the data step is implicitly ended.

- `spss.Dataset()` creates an instance of the `Dataset` class for the active dataset, which is then stored to the Python variable *ds*.

- `spss.Dataset(name=None)` creates a new dataset and an associated `Dataset` object, which is then stored to the Python variable *newds*. In this example, a new dataset will be created for each value of the IBM® SPSS® Statistics variable *dept*. New datasets are not set as active, so the active dataset is not changed by this operation.

- The `append` method of the `VariableList` object (obtained from the `varlist` property of the `Dataset` object) is used to add the variables to the new dataset. All of the variables in this example are numeric so the optional variable type argument to the `append` method is omitted.

- The code `ds.cases[0,0]` returns a list with a single element, which is the value of *dept* for the first case in the active dataset. You then extract the value by getting the 0th element of the list.

- When a new dataset is created, the name of the dataset is automatically generated by SPSS Statistics and is available from the *name* property of the `Dataset` object, as in `newds.name`. The names of the new datasets are stored to the Python dictionary *dsNames*. The dictionary keys are the values of the SPSS Statistics variable *dept* and the associated values are the names of the new datasets. In this example, storing the dataset names is necessary since they will be needed after the data step has been terminated and the `Dataset` objects no longer available.

- `ds.cases` is the `CaseList` object associated with the active dataset, so the first `for` loop iterates through the cases in the active dataset. On each iteration of the loop, *row* is a list consisting of the case values of the variables arranged in file order. When a new value of *dept* is encountered from the active dataset, a new dataset is created.

- The `append` method of the `CaseList` object is used to add the current case from the active dataset to the new dataset.

- The operation of saving the new datasets is done with command syntax, which is submitted with the `Submit` function (you can also use the `saveDataFile` function from the `spssaux` module to save the active dataset). The `Submit` function cannot be used within a data step so it is called after the data step has ended; that is, outside of the `with` block. The `SAVE` command works on the active dataset, so the `DATASET ACTIVATE` command is used to activate each new dataset, using the dataset names stored in *dsNames*.

Note: For a simpler example of creating a new dataset, see the topic on the `DataStep` class in the SPSS Statistics Help system.

Example: Merging Existing Datasets into a New Dataset

Using `Dataset` objects you can create a new dataset from multiple open datasets, combining the data from the existing datasets in any way you choose. As an example, the following Python user-defined function merges the case data from two existing datasets by combining cases that have matching values of a specified key variable. The function provides similar functionality to the IBM® SPSS® Statistics `MATCH FILES` command but doesn't require that the input datasets be sorted on the key variable, and only cases with matching values of the key variable in both datasets are retained.

```
def MergeDs(ds1Name, ds2Name, keyvar):
    """Merge the case data from two datasets using a specified key variable.
    A new dataset is created with the merged cases. Only cases with matching
    values of the key variable in both datasets are retained. The order of
    the cases in the resulting dataset is the order of the cases in the first
    specified dataset. Datasets do not need to be sorted on the key variable
    before merging.
    ds1Name and ds2Name are the names of the two datasets.
    keyvar is the name of the key variable.
    """
    with spss.DataStep():
        try:
            ds1 = spss.Dataset(name=ds1Name)
            ds2 = spss.Dataset(name=ds2Name)
            ds1keyind = ds1.varlist[keyvar].index
            ds2keyind = ds2.varlist[keyvar].index
        except:
            raise ValueError("Cannot access dataset " + ds1Name + " or dataset " +
            ds2Name + " or one of the datasets does not contain the specified " +
            "key variable " + keyvar)

        newds = spss.Dataset(name=None)
        for var in ds1.varlist:
            newds.varlist.append(var.name,var.type)
        # Create a list of the index values of the variables in the second dataset,
        # excluding the key variable.
        ds2varind = [i for i in range(len(ds2.varlist)) if i != ds2keyind]
        for i in ds2varind:
            newds.varlist.append(ds2.varlist[i].name,ds2.varlist[i].type)
        # Store the case values of the key variable from the second dataset
        keys2 = [item[0] for item in ds2.cases[0:len(ds2.cases),ds2keyind]]
        # Populate the case values of the new dataset
        for row in ds1.cases:
            try:
                ds2rowindex = keys2.index(row[ds1keyind])
                newcase = row
                for i in ds2varind:
                    newcase.append(ds2.cases[ds2rowindex,i][0])
                newds.cases.append(newcase)
            except:
                pass
```

- The `try` clause attempts to create dataset objects for the two datasets specified by the arguments *ds1Name* and *ds2Name*. Each dataset is also checked for the existence of the key variable specifed by the argument *keyvar*. The index value of the key variable, if it exists, is obtained from the `index` property of the associated `Variable` object, as in `ds1.varlist[keyvar].index`.

  If either dataset cannot be accessed or does not contain the specified key variable, an exception is raised.

- `spss.Dataset(name=None)` creates a new dataset. Variables of the same name and type as those from the dataset specified by *ds1Name* are added to the new dataset using the `append` method of the associated `VariableList` object (obtained from the `varlist` property of the `Dataset` object). The order of the appended variables is the file order of the variables from the dataset *ds1Name*.

- Variables of the same name and type as those from the dataset specified by *ds2Name* are then appended to the new dataset, except for the key variable that has already been added to the new dataset.

■ The case values of the key variable from the dataset *ds2Name* are stored to the Python list variable *keys2* in case order.

■ The `for row` loop iterates through all of the cases in dataset *ds1Name*. Each case is checked to see if the value of the key variable from dataset *ds1Name*—given by `row[ds1keyind]`—can be found in dataset *ds2Name*. If the key value exists in both datasets, a case is appended to the new dataset consisting of the variable values from dataset *ds1Name* followed by those from *ds2Name*, excluding the value of the key variable from *ds2Name*. If the key value from *ds1Name* is not found in *ds2Name* the case is skipped.

Example

The following example merges two very simple datasets.

```
*python_dataset_mergeds.sps.
DATA LIST LIST(,)/id var1.
BEGIN DATA
1, 11
2, 21
5, 51
3, 31
4, 41
END DATA.
DATASET NAME data1.
DATA LIST LIST(,)/id var2.
BEGIN DATA
2, 22
1, 12
6, 62
4, 42
END DATA.
DATASET NAME data2.
BEGIN PROGRAM.
import samplelib
samplelib.MergeDs('data1','data2','id')
END PROGRAM.
```

The `BEGIN PROGRAM` block starts with a statement to import the `samplelib` module, which contains the definition for the `MergeDs` function. The function is called with the names of the two datasets and the name of the key variable.

Note: To run this program block, copy the module file *samplelib.py* from the */examples/python* folder, in the accompanying examples, to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

Example: Modifying Case Values Utilizing a Regular Expression

The ability to modify case values of an open dataset allows you to transform your existing data without creating new variables. As an example, consider a string variable that contains a U.S. telephone number. The string may or may not contain delimiters (such as parentheses, dashes, periods, or spaces) or a 1 preceding the area code. You would like to remove these extraneous characters to obtain a 10-character string consisting of the area code, followed by the 3-digit trunk number, followed by the 4-digit number. For instance, you would like to transform (312) 555-1212 to 3125551212.

```
*python_dataset_modify_cases_re.sps.
DATA LIST/phone (A20).
BEGIN DATA
(312) 555-1212
3125551212
312-555-1212

13125551212
1 312 555 1212
END DATA.
BEGIN PROGRAM.
from __future__ import with_statement
import spss, re
phoneRegex = re.compile(r'[1]?(\d{3})\D*(\d{3})\D*(\d{4})')
with spss.DataStep():
   ds = spss.Dataset()
   for i in range(len(ds.cases)):
      match = phoneRegex.search(ds.cases[i][0])
      if match:
         ds.cases[i,0] = "".join(match.groups()).ljust(ds.varlist[0].type)
END PROGRAM.
```

■ This example makes use of the built-in Python module `re` for working with regular expressions, so the `import` statement includes it.

■ A regular expression will be used to extract the part of the string that contains the telephone number. The expression is `[1]?(\d{3})\D*(\d{3})\D*(\d{4})`. It will match an optional single occurrence of the digit `1`, followed by three digits (the area code), followed by an arbitrary number of nondigit characters, followed by three digits (the trunk), followed by an arbitrary number of nondigit characters, followed by four digits (the number).

Note: If you are not familiar with the syntax of regular expressions, a good introduction can be found in the section "Regular expression operations" in the Python Library Reference, available at *http://docs.python.org/lib/module-re.html*. You can also find information in "Using Regular Expressions" on p. 331.

■ The `compile` function from the `re` module compiles a regular expression. Compiling regular expressions is optional but increases the efficiency of matching when the expression is used several times in a single program. The argument is the regular expression as a string. The result of the `compile` function is a regular expression object, which in this example is stored to the Python variable *phoneRegex*.

Note: The `r` preceding the regular expression specifies a raw string, which ensures that any character sets specifying Python escape sequences are treated as raw characters and not the corresponding escape sequence. The current regular expression does not contain any Python escape sequences, but it is good practice to always use a raw string for a regular expression to avoid unintentional escape sequences.

■ `spss.Dataset()` creates an instance of the `Dataset` class for the active dataset, which is then stored to the Python variable *ds*.

■ The `for` loop iterates through all of the cases in the active dataset, making use of the fact that `len(ds.cases)` returns the number of cases in the dataset.

■ The `search` method of the compiled regular expression object scans a string for a match to the regular expression associated with the object. In this example, the string to scan is the value of the IBM® SPSS® Statistics variable *phone* for the current case, which is given by `ds.cases[i][0]`. The result of the `search` method is stored to the Python variable *match*.

■ The `search` method returns *None* if no position in the string matches the regular expression. This will occur if the value of *phone* for the current case is system-missing or does not contain the form of a U.S. telephone number. In either case, no action is taken.

■ If a match is found, the value of *phone* for the current case—specified by `ds.cases[i,0]`—is replaced with the telephone number without a leading `1` and with all delimiters removed. If no match is found, as for the single case with a missing value of *phone*, the case is simply skipped and is not modified.

The `groups` method of the match object returns a Python tuple containing the strings that match each of the groups defined in the regular expression. In this example, the regular expression contains the three groups `(\d{3})`, `(\d{3})`, and `(\d{4})` that contain the area code, trunk, and number respectively.

`"".join(match.groups())` collapses the tuple returned by the `groups` method into a string, concatenating the elements of the tuple with no separator.

Note: The transformation of the string containing the telephone number can also be done using the `sub` function from the `re` module.

Example: Displaying Value Labels as Cases in a New Dataset

The `valueLabels` property of the `Variable` class allows you to easily retrieve the value labels for any variable in an open dataset. The following Python user-defined function utilizes the `valueLabels` property to create a new dataset whose cases are the value labels from the active dataset.

```
def CreateVallabDs(filespec=None, dsName=None):
    """Create a new dataset containing those variables from the active dataset
    that have value labels. The case values of the new dataset are the value
    labels from the active dataset. If the active dataset does not have value
    labels, a message to that effect is printed in the log and no dataset
    is created.
    filespec is the file specification of an optional file to open as the
    active dataset.
    dsName is the optional name of the new dataset.
    """
    if filespec:
        try:
            spss.Submit("GET FILE = '%s'." %(filespec))
        except:
            raise ValueError("Cannot open file: " + filespec)

    with spss.DataStep():
        ds = spss.Dataset()
        oldname = ds.name
        newds = spss.Dataset(name=None)
        newname = newds.name
        labelsets=[]
        # Retrieve the value labels from the active dataset and create a variable
        # in the new dataset for each variable in the active one that has value
        # labels.
        for var in ds.varlist:
            if len(var.valueLabels):
                labels = var.valueLabels.data.values()
                labelsets.append(labels)
                maxlabelwidth = max([len(item) for item in labels])
                newds.varlist.append(var.name,maxlabelwidth)
        # Populate the cases of the new dataset
        if labelsets:
            maxnumvallabs = max([len(item) for item in labelsets])
            for i in range(maxnumvallabs):
                casevals = []
                for j in range(len(newds)):
                    if i <= len(labelsets[j]) - 1:
                        vallabel = labelsets[j][i]
                        casevals.append(vallabel.ljust(newds.varlist[j].type))
                    else:
                        casevals.append(None)
                newds.cases.append(casevals)
        else:
            # Discard the new dataset if no value labels were found in the active
            # dataset
            newds.close()
            print "Active dataset has no value labels."

    # Set the name of the new dataset to the specified name, if provided
    if labelsets and dsName:
        spss.Submit("""
        DATASET ACTIVATE %(newname)s.
        DATASET NAME %(dsName)s.
        DATASET ACTIVATE %(oldname)s.
        """ %locals())
```

- `CreateVallabDs` is a Python user-defined function with the two optional arguments *filespec* and *dsName*.

- The Python variable *ds* is a `Dataset` object for the active dataset. The name of the active dataset is retrieved with the `name` property and stored to the Python variable *oldname*.

- `spss.Dataset(name=None)` creates a new dataset and an associated `Dataset` object, which is then stored to the Python variable *newds*. The auto-generated name of the new dataset is stored to the Python variable *newname*. It will be needed later in case the user specified a value for the *dsName* argument.

- The first `for` loop iterates through the variables in the active dataset. On each iteration of the loop, *var* is an instance of the `Variable` class, representing a particular variable in the active dataset (variables are accessed in file order).

- `var.valueLabels` is an object representing the value labels for the variable associated with the current value of *var*. The object supports the `len` function, which returns the number of value labels. The `data` property of the object returns a Python dictionary whose keys are the values and whose associated values are the value labels. The Python variable *labels* is then a list consisting of the value labels for the current variable.

- The `append` method of the `VariableList` object (obtained from the `varlist` property of the `Dataset` object) is used to add a string variable to the new dataset. The name of the new variable is the name of the current variable from the active dataset and is retrieved from the `name` property of the current `Variable` object. The length of the new string variable is the length of the longest value label for the associated variable from the active dataset.

- If the active dataset has value labels, processing continues with populating the new dataset with those value labels. The `for i` loop has an iteration for each case in the new dataset. The number of cases is simply the number of value labels for the variable with the most value labels.

- The `for j` loop iterates over the variables in the new dataset. The number of variables in the new dataset is determined from `len(newds)`.

- The Python variable *casevals* is a list containing the case values for a new case. Since some variables may have fewer value labels than others, some of the case values for such variables will be missing. This is handled by specifying *None* for the value, which results in a system-missing value in the dataset.

- The `append` method of the `CaseList` object (obtained from the `cases` property of the `Dataset` object) is used to add a case to the new dataset.

- If the active dataset has no value labels, the new dataset is closed by calling the `close` method of the `Dataset` object. Since the new dataset is not the active one, the effect of the `close` method is to discard the dataset.

- If the user specified a value for the *dsName* argument and the active dataset has value labels, then activate the new dataset using the auto-generated name stored in *newname*, set the name of the new dataset to the specified name, and activate the original dataset. The activation and assignment of the dataset name are done through command syntax, which is submitted with the `Submit` function. Although you can activate a dataset within a data step, you cannot change the name of a dataset within a data step, so command syntax is used. The `Submit` function cannot be used within a data step so it is called after the data step has ended; that is, outside of the `with` block.

Example

As an example, create a new dataset from the value labels in *Employee data.sav*.

```
*python_dataset_create_vallabds.sps.
BEGIN PROGRAM.
import spss, samplelib
samplelib.CreateVallabDs(filespec='/examples/data/Employee data.sav')
end program.
```

The `BEGIN PROGRAM` block starts with a statement to import the `samplelib` module, which contains the definition for the `CreateVallabDs` function. The function is called with a file specification.

Note: To run this program block, copy the module file *samplelib.py* from the */examples/python* folder, in the accompanying examples, to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

Retrieving Output from Syntax Commands

The `spss` module provides the means to retrieve the output produced by syntax commands from an in-memory workspace, allowing you to access command output in a purely programmatic fashion.

Getting Started with the XML Workspace

To retrieve command output, you first route it via the Output Management System (OMS) to an area in memory referred to as the **XML workspace**. There it resides in a structure that conforms to the IBM® SPSS® Statistics Output XML Schema (*xml.spss.com/spss/oms*). Output is retrieved from this workspace with functions that employ XPath expressions.

For users familiar with XPath and desiring the greatest degree of control, the `spss` module provides a function that evaluates an XPath expression against an output item in the workspace and returns the result. For those unfamiliar with XPath, the `spssaux` module—a supplementary module that is installed with the IBM® SPSS® Statistics - Integration Plug-In for Python—includes a function for retrieving output from an XML workspace that constructs the appropriate XPath expression for you based on a few simple inputs. For more information, see the topic "Using the spssaux Module" on p. 284.

The examples in this section utilize explicit XPath expressions. Constructing the correct XPath expression (SPSS Statistics currently supports XPath 1.0) obviously requires knowledge of the XPath language. If you're not familiar with XPath, this isn't the place to start. In a nutshell, XPath is a language for finding information in an XML document, and it requires a fair amount of practice. If you're interested in learning XPath, a good introduction is the XPath tutorial provided by W3Schools at *http://www.w3schools.com/xpath/*.

In addition to familiarity with XPath, constructing the correct XPath expression requires an understanding of the structure of XML output produced by OMS, which includes understanding the XML representation of a pivot table. You can find an introduction, along with example XML, in the "Output XML Schema" topic in the Help system.

Example

In this example, we'll retrieve the mean value of a variable calculated from the Descriptives procedure, making explicit use of the `OMS` command to route the output to the XML workspace and using XPath to locate the desired value in the workspace.

```
*python_get_output_with_xpath.sps.
GET FILE='/examples/data/Employee data.sav'.
*Route output to the XML workspace.
OMS SELECT TABLES
  /IF COMMANDS=['Descriptives'] SUBTYPES=['Descriptive Statistics']
  /DESTINATION FORMAT=OXML XMLWORKSPACE='desc_table'
  /TAG='desc_out'.
DESCRIPTIVES VARIABLES=salary, salbegin, jobtime, prevexp
 /STATISTICS=MEAN.
OMSEND TAG='desc_out'.
*Get output from the XML workspace using XPath.
BEGIN PROGRAM.
import spss
handle='desc_table'
context="/outputTree"
xpath="//pivotTable[@subType='Descriptive Statistics'] \
       /dimension[@axis='row'] \
       /category[@varName='salary'] \
       /dimension[@axis='column'] \
       /category[@text='Mean'] \
       /cell/@text"
result=spss.EvaluateXPath(handle,context,xpath)
print "The mean value of salary is:",result[0]
spss.DeleteXPathHandle(handle)
END PROGRAM.
```

■ The OMS command is used to direct output from a syntax command to the XML workspace. The XMLWORKSPACE keyword on the DESTINATION subcommand, along with FORMAT=OXML, specifies the XML workspace as the output destination. It is a good practice to use the TAG subcommand, as done here, so as not to interfere with any other OMS requests that may be operating. The identifiers used for the COMMANDS and SUBTYPES keywords on the IF subcommand can be found in the OMS Identifiers dialog box, available from the Utilities menu.

   Note: The spssaux module provides a function for routing output to the XML workspace that doesn't involve the explicit use of the OMS command. For more information, see the topic "Using the spssaux Module" on p. 284.

■ The XMLWORKSPACE keyword is used to associate a name with this output in the workspace. In the current example, output from the DESCRIPTIVES command will be identified with the name *desc_table*. You can have many output items in the XML workspace, each with its own unique name.

■ The OMSEND command terminates active OMS commands, causing the output to be written to the specified destination—in this case, the XML workspace.

■ The BEGIN PROGRAM block extracts the mean value of *salary* from the XML workspace and displays it in a log item in the Viewer. It uses the function EvaluateXPath from the spss module. The function takes an explicit XPath expression, evaluates it against a specified output item in the XML workspace, and returns the result as a Python list.

■ The first argument to the EvaluateXPath function specifies the particular item in the XML workspace (there can be many) to which an XPath expression will be applied. This argument is referred to as the handle name for the output item and is simply the name given on the XMLWORKSPACE keyword on the associated OMS command. In this case, the handle name is *desc_table*.

- The second argument to `EvaluateXPath` defines the XPath context for the expression and should be set to `"/outputTree"` for items routed to the XML workspace by the `OMS` command.

- The third argument to `EvaluateXPath` specifies the remainder of the XPath expression (the context is the first part) and must be quoted. Since XPath expressions almost always contain quoted strings, you'll need to use a different quote type from that used to enclose the expression. For users familiar with XSLT for OXML and accustomed to including a namespace prefix, note that XPath expressions for the `EvaluateXPath` function should not contain the `oms:` namespace prefix.

- The XPath expression in this example is specified by the variable *xpath*. It is not the minimal expression needed to select the mean value of *salary* but is used for illustration purposes and serves to highlight the structure of the XML output.

  `//pivotTable[@subType='Descriptive Statistics']` selects the Descriptives Statistics table.

  `/dimension[@axis='row']/category[@varName='salary']` selects the row for *salary*.

  `/dimension[@axis='column']/category[@text='Mean']` selects the *Mean* column within this row, thus specifying a single cell in the pivot table.

  `/cell/@text` selects the textual representation of the cell contents.

- When you have finished with a particular output item, it is a good idea to delete it from the XML workspace. This is done with the `DeleteXPathHandle` function, whose single argument is the name of the handle associated with the item.

If you're familiar with XPath, you might want to convince yourself that the mean value of *salary* can also be selected with the following simpler XPath expression:

`//category[@varName='salary']//category[@text='Mean']/cell/@text`

Note: To the extent possible, construct your XPath expressions using language-independent attributes, such as the variable name rather than the variable label. That will help reduce the translation effort if you need to deploy your code in multiple languages. Also consider factoring out language-dependent identifiers, such as the name of a statistic, into constants. You can obtain the current language with the `SHOW OLANG` command.

You may also consider using **text_eng** attributes in place of **text** attributes in XPath expressions. **text_eng** attributes are English versions of **text** attributes and have the same value regardless of the output language. The `OATTRS` subcommand of the `SET` command specifies whether **text_eng** attributes are included in OXML output.

Retrieving Images Associated with Output

You can retrieve images associated with output routed to the XML workspace. This is particularly useful if you are developing an external application that utilizes the Integration Plug-In for Python to harvest output from SPSS Statistics. In this example, we'll retrieve both a bar chart and a statistic associated with output from the Frequencies procedure and create a simple html page that displays the information.

```
#GetOutputWithXPath.py
import spss, spssaux, tempfile, os.path

#Generate output from the Frequencies procedure for the variables inccat and
#income from demo.sav, and route the output to the XML workspace. In this example,
#the output consists of pivot tables and bar charts.
spss.Submit("""GET FILE='/examples/data/demo.sav'.
OMS
  /SELECT CHARTS TABLES
  /IF COMMANDS=['Frequencies']
  /DESTINATION FORMAT=OXML IMAGES=YES
   CHARTFORMAT=IMAGE IMAGEROOT='myimages' IMAGEFORMAT=JPG XMLWORKSPACE='demo'.

FREQUENCIES VARIABLES=inccat income
  /STATISTICS=MEDIAN
  /BARCHART FREQ
  /ORDER=ANALYSIS.

OMSEND.
""")

#Get the median value of the variable income from the XML workspace.
xpath="//category[@text='Median']//category[@varName='income']/cell/@number"
median=spss.EvaluateXPath('demo','/outputTree',xpath)[0]

#Get the bar chart for the variable inccat and save it to the user's temporary directory.
xpath="//chartTitle[@text='Income category in thousands']/chart/@imageFile"
imagename=spss.EvaluateXPath('demo','/outputTree',xpath)[0]
image = spss.GetImage('demo',imagename)
f = file(os.path.join(tempfile.gettempdir(),imagename),'wb')
f.truncate(image[1])
f.write(image[0])
f.close()

#Generate an html file that displays the retrieved bar chart along with an annotation
#for the median income.
f = file(os.path.join(tempfile.gettempdir(),'demo.html'),'w')
f.write('<html>')
f.write('<head>')
f.write('<title>Sample web page</title>')
f.write('</head>')
f.write('<body>')
f.write('<h2>Sample web page content</h2>')
f.write('<img src="file:///' + os.path.join(tempfile.gettempdir(),imagename) +'"/>')
f.write('<p><strong>***The median income is ' + median + ' thousand</p>')
f.close()
```

- The OMS command routes output from the FREQUENCIES command to the XML workspace. The XMLWORKSPACE keyword specifies that this output will be identified by the name *demo*.

- To route images along with the OXML output, the IMAGES keyword on the DESTINATION subcommand (of the OMS command) must be set to YES, and the CHARTFORMAT, MODELFORMAT, or TREEFORMAT keyword must be set to IMAGE.

- The first call to the EvaluateXPath function retrieves the median value of the variable *income*. In this case, the value returned by EvaluateXPath is a list with a single element, which is then stored to the variable *median*.

- The second call to the EvaluateXPath function is used to retrieve the name of the image associated with the bar chart for the variable *inccat*. The chart is identified by the chart title 'Income category in thousands' and the name of the image is the value of the imageFile attribute of the associated chart element.

- The GetImage function retrieves the image in binary form.

  The first argument to the GetImage function is the name of the handle that identifies the associated output in the XML workspace. The output in this example is associated with the handle name *demo*.

The second argument to `GetImage` is the name associated with the image.

The value returned by the `GetImage` function is a tuple with 3 elements. The first element is the binary image. The second element is the amount of memory required for the image. The third element is a string specifying the image type: "PNG", "JPG", "EMF", "BMP", or "VML".

- The image is written to an external file in the current user's temporary directory. The name of the file is the name of the image retrieved from the XML workspace. In that regard, image names in OXML output have the form of a filename, including the file extension—for example, *myimages_000.jpg*. Note also that the output file is opened in binary mode.

- A simple html file named *demo.html* is created in the current user's temporary directory. It contains a reference to the image file and an annotation for the retrieved value of the median income.

Writing XML Workspace Contents to a File

When writing and debugging XPath expressions, it is often useful to have a sample file that shows the XML structure. This is provided by the function `GetXmlUtf16` in the `spss` module, as well as by an option on the `OMS` command. The following program block recreates the XML workspace for the preceding example and writes the XML associated with the handle *desc_table* to the file */temp/descriptives_table.xml*.

```
*python_write_workspace_item.sps.
GET FILE='/examples/data/Employee data.sav'.
*Route output to the XML workspace.
OMS SELECT TABLES
  /IF COMMANDS=['Descriptives'] SUBTYPES=['Descriptive Statistics']
  /DESTINATION FORMAT=OXML XMLWORKSPACE='desc_table'
  /TAG='desc_out'.
DESCRIPTIVES VARIABLES=salary, salbegin, jobtime, prevexp
 /STATISTICS=MEAN.
OMSEND TAG='desc_out'.
*Write an item from the XML workspace to a file.
BEGIN PROGRAM.
import spss
spss.GetXmlUtf16('desc_table','/temp/descriptives_table.xml')
spss.DeleteXPathHandle('desc_table')
END PROGRAM.
```

The section of */temp/descriptives_table.xml* that specifies the Descriptive Statistics table, including the mean value of *salary*, is:

```
<pivotTable subType="Descriptive Statistics" text="Descriptive Statistics">
  <dimension axis="row" displayLastCategory="true" text="Variables">
    <category label="Current Salary" text="Current Salary"
    varName="salary" variable="true">
      <dimension axis="column" text="Statistics">
        <category text="N">
          <cell number="474" text="474"/>
        </category>
        <category text="Mean">
          <cell decimals="2" format="dollar" number="34419.567510548"
          text="$34,419.57"/>
        </category>
      </dimension>
    </category>
```

> *Note*: The output is written in Unicode (UTF-16), so you need an editor that can handle this in order to display it correctly. Notepad is one such editor.

Using the spssaux Module

The `spssaux` module, a supplementary module that is installed with the IBM® SPSS® Statistics - Integration Plug-In for Python, provides functions that simplify the task of writing to and reading from the XML workspace. You can route output to the XML workspace without the explicit use of the `OMS` command, and you can retrieve values from the workspace without the explicit use of XPath.

The `spssaux` module provides two functions for use with the XML workspace:

- `CreateXMLOutput` takes a command string as input, creates an appropriate `OMS` command to route output to the XML workspace, and submits both the `OMS` command and the original command to IBM® SPSS® Statistics.

- `GetValuesFromXMLWorkspace` retrieves output from an XML workspace by constructing the appropriate XPath expression from the inputs provided.

In addition, the `spssaux` module provides the function `CreateDatasetOutput` to route procedure output to a dataset. The output can then be retrieved using the `Cursor` class from the `spss` module or the `Spssdata` class from the `spssdata` module. This presents an approach for retrieving procedure output without the use of the XML workspace.

Example: Retrieving a Single Cell from a Table

The functions `CreateXMLOutput` and `GetValuesFromXMLWorkspace` are designed to be used together. To illustrate this, we'll redo the example from the previous section that retrieves the mean value of *salary* in *Employee data.sav* from output produced by the Descriptives procedure.

```
*python_get_table_cell.sps.
BEGIN PROGRAM.
import spss,spssaux
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
cmd="DESCRIPTIVES VARIABLES=salary,salbegin,jobtime,prevexp \
            /STATISTICS=MEAN."
handle,failcode=spssaux.CreateXMLOutput(
                        cmd,
                        omsid="Descriptives",
                        visible=True)
# Call to GetValuesFromXMLWorkspace assumes that Output Labels
# are set to "Labels", not "Names".
result=spssaux.GetValuesFromXMLWorkspace(
                handle,
                tableSubtype="Descriptive Statistics",
                rowCategory="Current Salary",
                colCategory="Mean",
                cellAttrib="text")
print "The mean salary is: ", result[0]
spss.DeleteXPathHandle(handle)
END PROGRAM.
```

As an aid to understanding the code, the `CreateXMLOutput` function is set to display Viewer output (`visible=True`), which includes the Descriptive Statistics table shown here.

Figure 17-1
Descriptive Statistics table

	N	Mean
Current Salary	474	$34,419.57
Beginning Salary	474	$17,016.09
Months since Hire	474	81.11
Previous Experience (months)	474	95.86
Valid N (listwise)	474	

- The call to `CreateXMLOutput` includes the following arguments:

   cmd. The command, as a quoted string, to be submitted. Output generated by this command will be routed to the XML workspace.

   omsid. The OMS identifier for the command whose output is to be captured. A list of these identifiers can be found in the OMS Identifiers dialog box, available from the Utilities menu. Note that by using the optional *subtype* argument (not shown here), you can specify a particular table type or a list of table types to route to the XML workspace.

   visible. This argument specifies whether output is directed to the Viewer in addition to being routed to the XML workspace. In the current example, *visible* is set to *true*, so that Viewer output will be generated. However, by default, `CreateXMLOutput` does not create output in the Viewer. A visual representation of the output is useful when you're developing code, since you can use the row and column labels displayed in the output to specify a set of table cells to retrieve.

   Note: You can obtain general help for the `CreateXMLOutput` function, along with a complete list of available arguments, by including the statement `help(spssaux.CreateXMLOutput)` in a program block.

- `CreateXMLOutput` returns two parameters—a handle name for the output item in the XML workspace and the maximum SPSS Statistics error level for the submitted syntax commands (0 if there were no errors).

- The call to `GetValuesFromXMLWorkspace` includes the following arguments:

   handle. This is the handle name of the output item from which you want to retrieve values. When `GetValuesFromXMLWorkspace` is used in conjunction with `CreateXMLOutput`, as is done here, this is the first of the two parameters returned by `CreateXMLOutput`.

   tableSubtype. This is the OMS table subtype identifier that specifies the table from which to retrieve values. In the current example, this is the Descriptive Statistics table. A list of these identifiers can be found in the OMS Identifiers dialog box, available from the Utilities menu.

   rowCategory. This specifies a particular row in an output table. The value used to identify the row depends on the optional *rowAttrib* argument. When *rowAttrib* is omitted, as is done here, *rowCategory* specifies the name of the row as displayed in the Viewer. In the current example, this is *Current Salary*, assuming that Output Labels are set to Labels, not Names.

   colCategory. This specifies a particular column in an output table. The value used to identify the column depends on the optional *colAttrib* argument. When *colAttrib* is omitted, as is done here, *colCategory* specifies the name of the column as displayed in the Viewer. In the current example, this is *Mean*.

cellAttrib. This argument allows you to specify the type of output to retrieve for the selected table cell(s). In the current example, the mean value of *salary* is available as a number in decimal form (cellAttrib="number") or formatted as dollars and cents with a dollar sign (cellAttrib="text"). Specifying the value of *cellAttrib* may require inspection of the output XML. This is available from the GetXmlUtf16 function in the spss module. For more information, see the topic "Writing XML Workspace Contents to a File" on p. 283.

Note: You can obtain general help for the GetValuesFromXMLWorkspace function, along with a complete list of available arguments, by including the statement help(spssaux.GetValuesFromXMLWorkspace) in a program block.

- GetValuesFromXMLWorkspace returns the selected items as a Python list. You can also obtain the XPath expression used to retrieve the items by specifying the optional argument xpathExpr=True. In this case, the function returns a Python **two-tuple** whose first element is the list of retrieved values and whose second element is the XPath expression.

- Some table structures cannot be accessed with the GetValuesFromXMLWorkspace function and require the explicit use of XPath expressions. In such cases, the XPath expression returned by specifying xpathExpr=True (in GetValuesFromXMLWorkspace) may be a helpful starting point.

Note: If you need to deploy your code in multiple languages, consider using language-independent identifiers where possible, such as the variable name for *rowCategory* rather than the variable label used in the current example. When using a variable name for *rowCategory* or *colCategory*, you'll also need to include the *rowAttrib* or *colAttrib* argument and set it to varName. Also consider factoring out language-dependent identifiers, such as the name of a statistic, into constants. You can obtain the current language with the SHOW OLANG command.

Example: Retrieving a Column from a Table

In this example, we will retrieve a column from the Iteration History table for the Quick Cluster procedure and check to see if the maximum number of iterations has been reached.

```
*python_get_table_column.sps.
BEGIN PROGRAM.
import spss, spssaux
spss.Submit("GET FILE='/examples/data/telco_extra.sav'.")
cmd = "QUICK CLUSTER\
  zlnlong zlntoll zlnequi zlncard zlnwire zmultlin zvoice\
  zpager zinterne zcallid zcallwai zforward zconfer zebill\
  /MISSING=PAIRWISE\
  /CRITERIA= CLUSTER(3) MXITER(10) CONVERGE(0)\
  /METHOD=KMEANS(NOUPDATE)\
  /PRINT INITIAL."
mxiter = 10
handle,failcode=spssaux.CreateXMLOutput(
                          cmd,
                          omsid="Quick Cluster",
                          subtype="Iteration History",
                          visible=True)
result=spssaux.GetValuesFromXMLWorkspace(
                handle,
                tableSubtype="Iteration History",
                colCategory="1",
                cellAttrib="text")
if len(result)==mxiter:
    print "Maximum iterations reached for QUICK CLUSTER procedure"
spss.DeleteXPathHandle(handle)
END PROGRAM.
```

As an aid to understanding the code, the `CreateXMLOutput` function is set to display Viewer output (`visible=True`), which includes the Iteration History table shown here.

Figure 17-2
Iteration History table

Iteration	Change in Cluster Centers		
	1	2	3
1	3.298	3.590	3.491
2	1.016	.427	.931
3	.577	.320	.420
4	.240	.180	.195
5	.119	.125	.108
6	.093	.083	.027
7	.069	.094	.032
8	.059	.051	.018
9	.035	.085	.063
10	.025	.359	.333

- The call to `CreateXMLOutput` includes the argument *subtype*. It limits the output routed to the XML workspace to the specified table—in this case, the Iteration History table. The value specified for this parameter should be the OMS table subtype identifier for the desired table. A list of these identifiers can be found in the OMS Identifiers dialog box, available from the Utilities menu.

- By calling `GetValuesFromXMLWorkspace` with the argument *colCategory*, but without the argument *rowCategory*, all rows for the specified column will be returned. Referring to the Iteration History table shown above, the column labeled *1*, under the *Change in Cluster Centers* heading, contains a row for each iteration (as do the other two columns). The variable *result* will then be a list of the values in this column, and the length of this list will be the number of iterations.

Example: Retrieving Output without the XML Workspace

In this example, we'll use the `CreateDatasetOutput` function to route output from a `FREQUENCIES` command to a dataset. We'll then use the output to determine the three most frequent values for a specified variable—in this example, the variable *jobtime* from *Employee data.sav*.

```
*python_output_to_dataset.sps.
BEGIN PROGRAM.
import spss, spssaux, spssdata
spss.Submit(r"""
GET FILE='/examples/data/Employee data.sav'.
DATASET NAME employees.
""")
cmd = "FREQUENCIES jobtime /FORMAT=DFREQ."
datasetName, err = spssaux.CreateDatasetOutput(
                              cmd,
                              omsid='Frequencies',
                              subtype='Frequencies')
spss.Submit("DATASET ACTIVATE " + datasetName + ".")
data = spssdata.Spssdata()
print "Three most frequent values of jobtime:\n"
print"Months\tFrequency"
for i in range(3):
   row=data.fetchone()
   print str(row.Var2) + "\t\t" + str(int(row.Frequency))
data.close()
END PROGRAM.
```

As a guide to understanding the code, a portion of the output dataset is shown here.

Figure 17-3
Resulting dataset from CreateDatasetOutput

- In order to preserve the active dataset, the `CreateDatasetOutput` function requires it to have a dataset name. If the active dataset doesn't have a name, it is assigned one. Here, we've simply assigned the name *employees* to the active dataset.

- The call to `CreateDatasetOutput` includes the following arguments:

  cmd. The command, as a quoted string, to be submitted. Output generated by this command will be routed to a new dataset.

  omsid. The OMS identifier for the command whose output is to be captured. A list of these identifiers can be found in the OMS Identifiers dialog box, available from the Utilities menu.

  subtype. This is the OMS table subtype identifier for the desired table. In the current example, this is the Frequencies table. Like the values for *omsid*, these identifiers are available from the OMS Identifiers dialog box.

Note: You can obtain general help for the `CreateDatasetOutput` function, along with a complete list of available arguments, by including the statement `help(spssaux.CreateDatasetOutput)` in a program block.

- `CreateDatasetOutput` returns two parameters—the name of the dataset containing the output and the maximum SPSS Statistics error level for the submitted syntax commands (0 if there were no errors).

- Once you have called `CreateDatasetOutput`, you need to activate the output dataset before you can retrieve any data from it. In this example, data is retrieved using an instance of the `Spssdata` class from the `spssdata` module, a supplementary module (installed with the Integration Plug-In for Python) that provides a number of features that simplify the task of working with case data. The instance is stored to the Python variable *data*.

- Using `/FORMAT=DFREQ` for the `FREQUENCIES` command produces output where categories are sorted in descending order of frequency. Obtaining the three most frequent values simply requires retrieving the first three cases from the output dataset.

- Cases are retrieved one at a time in sequential order using the `fetchone` method, as in `data.fetchone()`. On each iteration of the `for` loop, *row* contains the data for a single case. Referring to the portion of the output dataset shown in the previous figure, *Var2* contains the values for *jobtime* and *Frequency* contains the frequencies of these values. You access the value for a particular variable within a case by specifying the variable name, as in `row.Var2` or `row.Frequency`.

For more information on working with the `Spssdata` class, see "Using the spssdata Module" on p. 245.

Chapter
18

Creating Procedures

The IBM® SPSS® Statistics - Integration Plug-In for Python enables you to create user-defined Python programs that have almost the same capabilities as IBM® SPSS® Statistics procedures, such as FREQUENCIES or REGRESSION. Since they behave like built-in SPSS Statistics procedures, we'll refer to such Python programs as **procedures**. A procedure can read the data, perform computations on the data, add new variables and/or new cases to the active dataset, and produce pivot table output and text blocks. Procedures are the natural approach in a variety of situations, for instance:

■ You have a statistical analysis that can be done by combining various built-in procedures and/or transformations, but it requires logic to determine which procedures and transformations to run and when to run them. In addition, it may need to use output from one procedure or transformation in another. Since you can submit syntax commands from Python, you can write a procedure that uses Python logic to drive the SPSS Statistics program flow. The program flow might depend on the data as well as a set of input parameters to the procedure.

■ You have a custom algorithm—perhaps a statistical analysis that isn't provided by SPSS Statistics—that you want to apply to SPSS Statistics datasets. You can code the algorithm in Python and include it in a procedure that reads the data from SPSS Statistics and applies the algorithm. You might even use the powerful data transformation abilities of SPSS Statistics to transform the data before reading it into Python—for instance, aggregating the data. The results can be written as new variables or new cases to the active dataset or as pivot table output directed to the Viewer or exported via the Output Management System (OMS).

Getting Started with Procedures

Procedures are simply user-defined Python functions that take advantage of the IBM® SPSS® Statistics - Integration Plug-In for Python features to read the data, write to the active dataset, and produce output. Since they're written in Python, procedures have access to the full computational power of the Python language. As a simple example, consider a procedure that reads the active dataset and creates a pivot table summarizing the number of cases with and without missing values.

© Copyright SPSS Inc. 1989, 2010   291

```
def MissingSummary(filespec):
    """Summarize the cases with and without missing values in
    the active dataset.
    filespec is a string that identifies the file to be read.
    """
    spss.Submit("GET FILE='%s'." %(filespec))
    # Read the data and check for missing values
    data=spssdata.Spssdata()
    data.makemvchecker()
    nvalid = 0; nmissing = 0
    for row in data:
        if data.hasmissing(row):
            nmissing += 1
        else:
            nvalid +=1
    data.close()
    # Create pivot table and text block output
    spss.StartProcedure("myorganization.com.MissingSummary")
    table = spss.BasePivotTable("Case Summary","OMS table subtype")
    table.SetDefaultFormatSpec(spss.FormatSpec.Count)
    table.SimplePivotTable(rowlabels=['Valid','Missing'],
                           collabels=['Count'],
                           cells = [nvalid,nmissing])
    spss.TextBlock("Sample Text Block","A line of sample text in a text block")
    spss.EndProcedure()
```

■ Python functions are defined with the keyword `def`, followed by the name of the function and a list of parameters enclosed in parentheses. In this example, the name of the function is `MissingSummary`, and it requires a single argument specifying the file to be read. The colon at the end of the `def` statement is required.

■ The `Submit` function is used to submit a `GET` command to open the file passed in as *filespec*.

■ The code to read the data and identify cases with missing values makes use of the `Spssdata` class from the `spssdata` module (a supplementary module installed with the Integration Plug-In for Python). The `Spssdata` class builds on the functionality in the `Cursor` class (provided with the `spss` module) to simplify the task of working with case data. For our purposes, the `Spssdata` class contains convenient methods for identifying missing values. For more information, see the topic "Reading Case Data with the Spssdata Class" in Chapter 15 on p. 246.

■ The `close` method closes the cursor used to read the data. You must close any open cursor before creating output with the `StartProcedure` function discussed below.

■ To create output in the form of pivot tables or text blocks, you first call the `StartProcedure` function from the `spss` module. The single argument to the `StartProcedure` function is the name to associate with the output. This is the name that appears in the outline pane of the Viewer associated with output produced by the procedure, as shown in Figure 18-1. It is also the command name associated with this procedure when routing output from this procedure with OMS (Output Management System).

■ In order that names associated with output do not conflict with names of existing syntax commands (when working with OMS), it is recommended that they have the form *yourorganization.com.procedurename*, as done here.

■ Pivot tables are created with the `BasePivotTable` class. For simple pivot tables consisting of a single row dimension and a single column dimension, you can use the `SimplePivotTable` method of the `BasePivotTable` class, as done here. In the current example, the pivot table

has one row dimension with two rows and one column dimension with a single column. For more information, see the topic "Creating Pivot Table Output" on p. 297.

■ Text blocks are created with the `TextBlock` class. This example includes a text block consisting of a single line of text, although the `TextBlock` class also supports multiline text blocks.

   Note: You can also use the Python `print` statement to write text output to Python's standard output, which is directed to a log item in the IBM® SPSS® Statistics Viewer, if a Viewer is available.

■ You call the `EndProcedure` function to signal the end of output creation.

To use a procedure you've written, you save it in a Python module. For instance, the definition of `MissingSummary` can be found in the Python module *samplelib.py*, located in the */examples/python* folder of the accompanying examples. A Python module is simply a text file containing Python definitions and statements. You can create a module with a Python IDE, or with any text editor, by saving a file with an extension of *.py*. The name of the file, without the *.py* extension, is then the name of the module. You can have many functions in a single module.

   Since we're concerned with Python functions that interact with SPSS Statistics, our procedures will probably call functions in the `spss` module, and possibly functions in some of the supplementary modules like `spssaux` and `spssdata`, as in this example. The module containing your procedures will need to include `import` statements for any other modules whose functions are used by the procedures.

   Finally, you must ensure that the Python interpreter can find your module, which means that the location of the module must be on the Python search path. To be sure, you can save the module to your Python *site-packages* directory.

   To run a procedure, you import the module containing it and call it with the necessary arguments. As an example, we'll run the `MissingSummary` procedure on the *demo.sav* dataset.

```
*python_missing_summary.sps.
BEGIN PROGRAM.
import samplelib
samplelib.MissingSummary("/examples/data/demo.sav")
END PROGRAM.
```

Note: To run this program block, copy the module file *samplelib.py* from the */examples/python* folder, in the accompanying examples, to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

Result

Figure 18-1
Output from the MissingSummary procedure
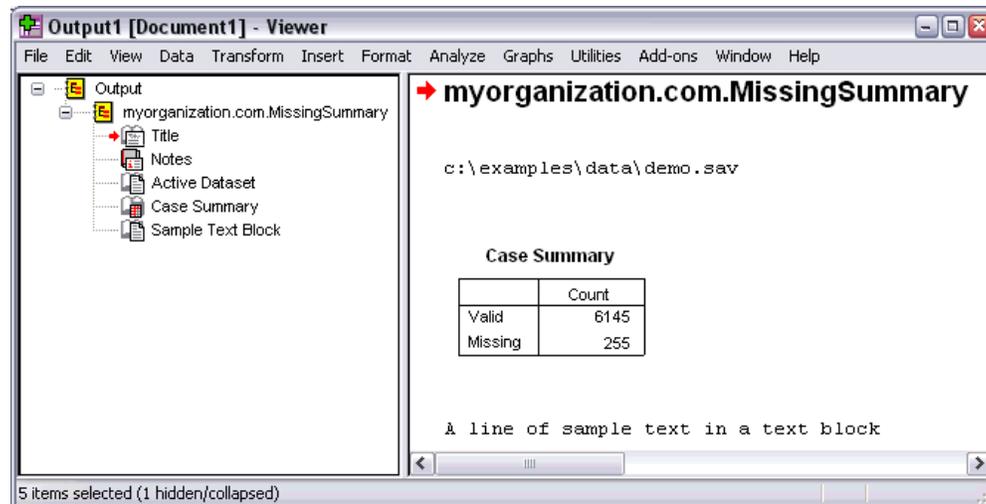
Alternative Approaches

Instead of including your procedure's code in a Python function, you can simply include it in a
`BEGIN PROGRAM-END PROGRAM` block, although this precludes you from invoking the code by
name or passing arguments. For example, a trivial piece of code to retrieve the case count from
the active dataset and create a text block with that information is:

```
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/demo.sav'.")
ncases=spss.GetCaseCount()
spss.StartProcedure("myorganization.com.MyProcedure")
spss.TextBlock("Total Case Count","Case Count: " + str(ncases))
spss.EndProcedure()
END PROGRAM.
```

By creating a command syntax file that contains this program block, you can effectively associate
a name—the name of the command syntax file—with the program block. You run the program
block by using the `INSERT` command to include the command syntax file (containing the block) in
a session.

As a further alternative to creating a procedure as a Python function, you can embed your code
in a Python class. For more information, see the topic on the `BaseProcedure` class in the
SPSS Statistics Help system.

Procedures with Multiple Data Passes

Sometimes a procedure requires more than one pass of the data, for instance, a first pass to calculate
values that depend on all cases and a second one to create new variables based on those values.

   The following example illustrates the use of a two-pass procedure. The first pass reads the data to compute group means, and the second pass adds the mean values as a new variable in the active dataset. A listing of the group means is displayed in a pivot table.

```
def GroupMeans(groupVar,sumVar):
    """Calculate group means for a selected variable using a specified
    categorical variable to define the groups. Display the group means
    in a pivot table and add a variable for the group means to the
    active dataset.
    groupVar is the name of the categorical variable (as a string) used
    to define the groups.
    sumVar is the name of the variable (as a string) for which means are
    to be calculated.
    """
    data=spssdata.Spssdata(indexes=(groupVar,sumVar),accessType='w',
                           omitmissing=True)
    Counts={};Sums={}
    # First data pass
    for item in data:
        cat=int(item[0])
        Counts[cat]=Counts.get(cat,0) + 1
        Sums[cat]=Sums.get(cat,0) + item[1]
    for cat in sorted(Counts):
        Sums[cat]=Sums[cat]/Counts[cat]
    data.restart()
    data.append(spssdata.vdef('mean_'+sumVar+'_by_'+groupVar))
    data.commitdict()
    # Second data pass
    for item in data:
        data.casevalues([Sums[int(item[0])]])
    data.close()
    spss.StartProcedure("myorganization.com.GroupMeans")
    table = spss.BasePivotTable("Mean " + sumVar + " by " + groupVar,
                                "OMS table subtype")
    table.SimplePivotTable(rowdim=groupVar,
                           rowlabels=[cat for cat in sorted(Counts)],
                           collabels=['mean ' + sumVar],
                           cells = [Sums[cat] for cat in Sums])

    spss.EndProcedure()
```

■ `GroupMeans` is a Python user-defined function containing the procedure that calculates the group means. The arguments required by the procedure are the names of the grouping variable (*groupVar*) and the variable for which group means are desired (*sumVar*).

■ An instance of the `Spssdata` class is created that provides write access to the active dataset and also allows you to retrieve case data for the variables specified as *groupVar* and *sumVar*. The argument `omitmissing=True` specifies that cases with missing values are skipped. The `Spssdata` class is part of the `spssdata` module—a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python. For more information, see the topic "Using the spssdata Module" in Chapter 15 on p. 245.

■ The two Python dictionaries *Counts* and *Sums* are built dynamically to have a key for each value of the grouping variable found in the case data. The value associated with each key in *Counts* is the number of cases with that value of *groupVar*, and the value for each key in *Sums* is the cumulative value of *sumVar* (the variable for which means are calculated) for that value of *groupVar*. The code `Counts.get(cat,0)` and `Sums.get(cat,0)` gets the dictionary value associated with the key given by the value of *cat*. If the key doesn't exist, the expression evaluates to 0.

■ At the completion of the first data pass, the cumulative values of *sumVar* (stored in the Python dictionary *Sums*) and the associated counts (stored in the Python dictionary *Counts*) are used to compute the mean of *sumVar* for each value of *groupVar* found in the data. The Python dictionary *Sums* is updated to contain the calculated means.

■ The `restart` method from the `Spssdata` class is called to reset the write cursor in preparation for another data pass. `restart` needs to be called before creating new variables on subsequent data passes.

■ The `append` method from the `Spssdata` class is used to create a new variable that is set to the mean for the group associated with each case. The case values are set on the second data pass. Since cases with missing values are skipped, such cases will have the value `SYSMIS` for the new variable.

■ The `StartProcedure` function signals the beginning of output creation for the procedure. Output will be associated with the name *myorganization.com.GroupMeans*.

■ A pivot table displaying the group means is created using the `SimplePivotTable` method from the `BasePivotTable` class. For more information, see the topic "Creating Pivot Table Output" on p. 297.

Running the Procedure

As an example, we'll calculate the mean salary by educational level for the *Employee data.sav* dataset. The grouping variable is *educ*, and *salary* is the variable for which group means will be calculated.

```
*python_group_means.sps.
BEGIN PROGRAM.
import spss, samplelib
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
samplelib.GroupMeans("educ","salary")
END PROGRAM.
```

The `BEGIN PROGRAM` block starts with a statement to import the `samplelib` module, which contains the definition for the `GroupMeans` function.

Note: To run this program block, copy the module file *samplelib.py* from the */examples/python* folder, in the accompanying examples, to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

Results

Figure 18-2
Pivot table output from the GroupMeans procedure

Mean salary by educ

educ	mean salary
8	24399.057
12	25887.158
14	31625.000
15	31685.000
16	48225.932
17	59527.273
18	65127.778
19	72520.370
20	64312.500
21	65000.000

Figure 18-3
New variable mean_salary_by_educ created by the GroupMeans procedure

	educ	jobcat	salary	salbegin	jobtime	prevexp	minority	mean_salary_by_educ
1	15	3	$57,000	$27,000	98	144	0	31685.00
2	16	1	$40,200	$18,750	98	36	0	48225.93
3	12	1	$21,450	$12,000	98	381	0	25887.16
4	8	1	$21,900	$13,200	98	190	0	24399.06
5	15	1	$45,000	$21,000	98	138	0	31685.00

Creating Pivot Table Output

Procedures can produce output in the form of pivot tables, which can be displayed in the IBM® SPSS® Statistics viewer or written to an external file using the SPSS Statistics Output Management System. The following figure shows the basic structural components of a pivot table.

Figure 18-4
Pivot table structure

Pivot tables consist of one or more dimensions, each of which can be of the type row, column, or layer. Each dimension contains a set of categories that label the elements of the dimension—for instance, row labels for a row dimension. A layer dimension allows you to display a separate two-dimensional table for each category in the layered dimension—for example, a separate table for each value of minority classification, as shown here. When layers are present, the pivot table can be thought of as stacked in layers, with only the top layer visible.

Each cell in the table can be specified by a combination of category values. In the example shown here, the indicated cell is specified by a category value of *Male* for the *Gender* dimension, *Custodial* for the *Employment Category* dimension, and *No* for the *Minority Classification* dimension.

Pivot tables are created with the `BasePivotTable` class. For the common case of creating a pivot table with a single row dimension and a single column dimension, the `BasePivotTable` class provides the `SimplePivotTable` method. As a simple example, consider a pivot table with static values.

Example

```
import spss
spss.StartProcedure("myorganization.com.SimpleTableDemo")
table = spss.BasePivotTable("Sample Pivot Table",
             "OMS table subtype",
              caption="Table created with SimplePivotTable method")

table.SimplePivotTable(rowdim = "Row",
                       rowlabels = [1,2],
                       coldim = "Column",
                       collabels = ["A","B"],
                       cells = ["1A","1B","2A","2B"])
spss.EndProcedure()
```

Result

Figure 18-5
Viewer output of simple pivot table
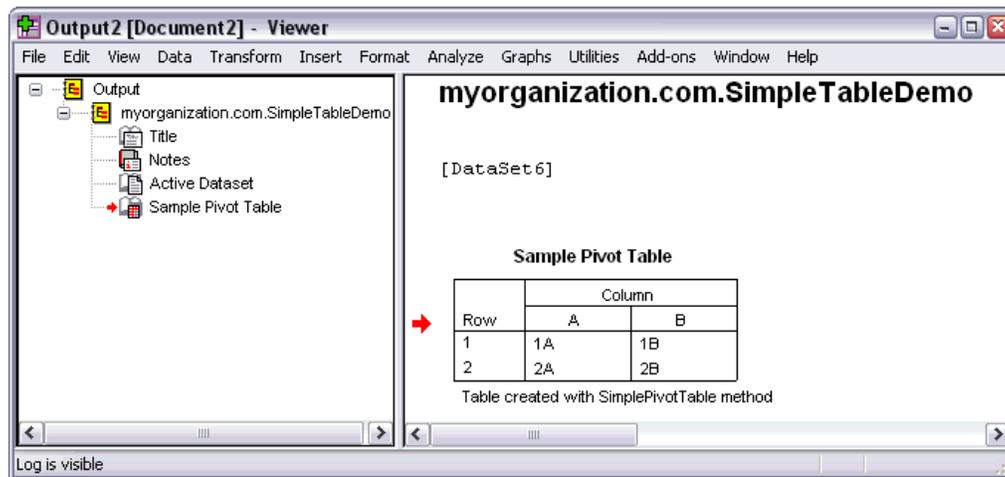
- The pivot table output is associated with the name *myorganization.com.SimpleTableDemo*. For simplicity, we've provided the code while leaving aside the context in which it might be run. For more information, see the topic "Getting Started with Procedures" on p. 291.

- To create a pivot table, you first create an instance of the `BasePivotTable` class and assign the instance to a Python variable. In this example, the Python variable *table* contains a reference to a pivot table instance.

- The first argument to the `BasePivotTable` class is a required string that specifies the title that appears with the table. Each table created by a given `StartProcedure` call should have a unique title. The title appears in the outline pane of the Viewer as shown in Figure 18-5.

- The second argument to the `BasePivotTable` class is a string that specifies the OMS (Output Management System) table subtype for this table. Unless you are routing this pivot table with OMS or need to write an autoscript for this table, you will not need to keep track of this value, although the value is still required. Specifically, it must begin with a letter and have a maximum of 64 bytes.

- Notice that the item for the table in Figure 18-5 is one level deeper than the root item for the name associated with output from this `StartProcedure` call. This is the default behavior. You can use the optional argument *outline* (to the `BasePivotTable` class) to create an item in the outline pane of the Viewer that will contain the item for the table.

- The optional argument *caption* used in this example specifies a caption for the table, as shown in Figure 18-5.

Once you've created an instance of the `BasePivotTable` class, you use the `SimplePivotTable` method to create the structure of the table and populate the table cells. The arguments to the `SimplePivotTable` method are as follows:

- **rowdim.** An optional label for the row dimension, given as a string. If empty, the row dimension label is hidden.

- **rowlabels.** An optional list of items to label the row categories. Labels can be given as numeric values or strings, or you can specify that they be treated as variable names or variable values. Treating labels as variable names means that display settings for variable names in pivot tables (names, labels, or both) are honored when creating the table. And treating labels as variable values means that display settings for variable values in pivot tables (values, labels, or both) are honored. For more information, see the topic "Treating Categories or Cells as Variable Names or Values" on p. 300.

  Note: The number of rows in the table is equal to the length of *rowlabels*, when provided. If *rowlabels* is omitted, the number of rows is equal to the number of elements in the argument *cells*.

- **coldim.** An optional label for the column dimension, given as a string. If empty, the column dimension label is hidden.

- **collabels.** An optional list of items to label the column categories. The list can contain the same types of items as *rowlabels* described above.

  Note: The number of columns in the table is equal to the length of *collabels*, when provided. If *collabels* is omitted, the number of columns is equal to the length of the first element of *cells*.

- **cells.** This argument specifies the values for the cells of the pivot table and can be given as a one- or two-dimensional sequence. In the current example, *cells* is given as the one-dimensional sequence `["1A","1B","2A","2B"]`. It could also have been specified as the two-dimensional sequence `[["1A","1B"],["2A","2B"]]`.

  Elements in the pivot table are populated in row-wise fashion from the elements of *cells*. In the current example, the table has two rows and two columns (as specified by the row and column labels), so the first row will consist of the first two elements of *cells* and the second row will consist of the last two elements. When *cells* is two-dimensional, each one-dimensional element specifies a row. For example, with *cells* given by `[["1A","1B"],["2A","2B"]]`, the first row is `["1A","1B"]` and the second row is `["2A","2B"]`.

  Cells can be given as numeric values or strings, or you can specify that they be treated as variable names or variable values (as described for *rowlabels* above). For more information, see the topic "Treating Categories or Cells as Variable Names or Values" on p. 300.

If you require more functionality than the `SimplePivotTable` method provides, there are a variety of methods for creating the table structure and populating the cells. If you're creating a pivot table from data that has splits, you'll probably want separate results displayed for each

split group. For more information, see the topics on the `BasePivotTable` class and the `SplitChange` function in the SPSS Statistics Help system.

Treating Categories or Cells as Variable Names or Values

The `BasePivotTable` class supports treating categories (row or column) and cell values as variable names or variable values. Treating categories as variable names means that display settings for variable names in pivot tables (names, labels, or both) are honored when creating the table, and treating categories as variable values means that display settings for variable values in pivot tables (values, labels, or both) are honored.

Example

In this example, we create a pivot table displaying the gender with the highest frequency count for each employment category in the *Employee data.sav* dataset. Row categories and cell values are specified as variable values and the single column category is specified as a variable name.

```
*python_ptable_VarValue_VarName.sps.
BEGIN PROGRAM.
import spss, spssdata
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
data=spssdata.Spssdata(indexes=('jobcat','gender'),omitmissing=True)
data.makemvchecker()
jobcats={1:{'f':0,'m':0},2:{'f':0,'m':0},3:{'f':0,'m':0}}
# Read the data and store gender counts for employment categories
for row in data:
   cat=int(row.jobcat)
   jobcats[cat][row.gender]+=1
data.CClose()
# Create a list of cell values for the pivot table
cell_list=[]
for cat in sorted(jobcats):
   testval = cmp(jobcats[cat]['f'],jobcats[cat]['m'])
   if testval==0:
     cell_list.append("Equal")
   else:
     cell_list.append(spss.CellText.VarValue(1,{1:'f',-1:'m'}[testval]))
# Create the pivot table
spss.StartProcedure("myorganization.com.SimpleTableDemo")
table = spss.BasePivotTable("Majority " + spss.GetVariableLabel(1) + \
                            " by " + spss.GetVariableLabel(4),
                            "OMS table subtype")
table.SimplePivotTable(rowdim = spss.GetVariableLabel(4),
                     rowlabels = [spss.CellText.VarValue(4,1),
                                  spss.CellText.VarValue(4,2),
                                  spss.CellText.VarValue(4,3)],
                     collabels = [spss.CellText.VarName(1)],
                     cells = cell_list)
spss.EndProcedure()
END PROGRAM.
```

Results

Figure 18-6
Variable names shown as labels and variable values shown as value labels

Majority Gender by Employment Category

Employment Category	Gender
Clerical	Female
Custodial	Male
Manager	Male

Figure 18-7
Variable names shown as names and variable values shown as values

Majority Gender by Employment Category

Employment Category	gender
1	f
2	m
3	m

- The code makes use of the `Spssdata` class from the `spssdata` module (a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python) to read the data (only the values for *jobcat* and *gender* are read) and skip over cases with missing values. For more information, see the topic "Using the spssdata Module" in Chapter 15 on p. 245.

- The Python dictionary *jobcats* holds the counts of each gender for each employment category. On each iteration of the first `for` loop, the Python variable *row* contains the data for the current case, so that `row.jobcat` is the employment category and `row.gender` is the gender. These values are used as keys to the appropriate element in *jobcats*, which is then incremented by 1.

- The second `for` loop iterates through the employment categories and determines the gender with the highest frequency, making use of the Python built-in function `cmp` to compare the counts for each gender. The result is appended to a list of cell values to be used in the `SimplePivotTable` method. Other than the case of a tie (equal counts for each gender), values are given as `spss.CellText.VarValue` objects, which specifies that they be treated as variable values. `spss.CellText.VarValue` objects require two arguments, the index of the associated variable (index values represent position in the active dataset, starting with 0 for the first variable in file order) and the value. In the current example, the variable index for *gender* is 1 and the value is either `'f'` or `'m'`.

- The `StartProcedure` function signals the beginning of output creation. Output will be associated with the name *myorganization.com.SimpleTableDemo*.

- The row categories for the table are the employment categories from *Employee data.sav* and are specified as variable values using `spss.CellText.VarValue` objects. The index of *jobcat* (the employment category) in *Employee data.sav* is 4, and the three employment categories have values 1, 2, and 3.

- The single column category is the name of the gender variable, given as a `spss.CellText.VarName` object, which specifies that it be treated as a variable name. `spss.CellText.VarName` objects require one argument, the index of the associated variable. The index for *gender* in *Employee data.sav* is 1.

- Figure 18-6 and Figure 18-7 show the results for two different settings of output labels for pivot tables (set from Edit > Options > Output Labels). Figure 18-6 shows the case of variable names displayed as the associated variable label and variable values as the associated value label. Figure 18-7 shows the case of variable names displayed as they appear in the data editor and variable values given as the raw value.

Specifying Formatting for Numeric Cell Values

You can change the default format used for displaying numeric values in pivot table cells and category labels, or override it for selected cells or categories.

Example: Changing the Default Format

```
*python_ptable_change_format.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
spss.StartProcedure("myorganization.com.Demo")
table = spss.BasePivotTable("Table Title","OMS table subtype")
table.SetDefaultFormatSpec(spss.FormatSpec.Count)
table.SimplePivotTable(collabels=["Integer Value"],
                       cells = [1,2])
spss.EndProcedure()
END PROGRAM.
```

Result

Figure 18-8
Integer cells formatted with the Count format

	Integer Value
row0	1
row1	2

- The `SetDefaultFormatSpec` method from the `BasePivotTable` class is used to change the default format for numeric cells. The argument is of the form `spss.FormatSpec.format` where `format` is one of those listed in the topic on the `Number` class—for example, `spss.FormatSpec.Count`. The selected format is applied to all cells. A list of available formats, as well as a brief guide to choosing a format, is provided in the documentation for the `Number` class in the IBM® SPSS® Statistics Help system. Instances of the `BasePivotTable` class have an implicit default format of `spss.FormatSpec.GeneralStat`.

- This example also illustrates that default row or column categories are provided when one or the other are omitted, as done here for row categories.

Example: Overriding the Default Format for Selected Cells

```
*python_ptable_override_format.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='/examples/data/Employee data.sav'.")
spss.StartProcedure("myorganization.com.Demo")
table = spss.BasePivotTable("Table Title","OMS table subtype")
table.SimplePivotTable(rowlabels=["Default overridden","Default used"],
     collabels=["Numeric value"],
     cells = [spss.CellText.Number(1.2345,spss.FormatSpec.Count),
              2.34])
spss.EndProcedure()
END PROGRAM.
```

Result

Figure 18-9
Default format overridden for selected cell

	Numeric value
Default overridden	1
Default used	2.340

You override the default format for a cell or category by specifying the value as an `spss.CellText.Number` object. The arguments are the numeric value and the name of a format specification, given as `spss.FormatSpec.format.`

Data Transformations

The Python module `trans`, a supplementary module available for download from Developer Central at *http://www.spss.com/devcentral/*, provides the framework for using Python functions as casewise transformation functions to be applied to a dataset. This enables an essentially limitless extension to the set of functions that can be used to transform data. You can use functions from the standard Python library, third-party Python libraries, or your own custom Python functions.

- Casewise results are stored as new variables (modification of existing variables is not supported).
- Multiple transformation functions can be applied on a single data pass.
- Results of a transformation function can be used in subsequent transformation functions.

Note: To run the examples in this chapter, download the following modules from Developer Central and save them to your Python *site-packages* directory: `trans` and `extendedTransforms`. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

Getting Started with the trans Module

The `Tfunction` class, in the `trans` module, is used to specify a set of Python functions to be applied in a given data pass and to execute the data pass on the active dataset. Each Python function creates one or more new variables in the active dataset using existing variables or Python expressions as inputs. For example, consider applying the hyperbolic sine function (available with Python but not with IBM® SPSS® Statistics) from the `math` module (a standard module included with Python) as well as a simple user-defined function named `strfunc`. For simplicity, we've included the definition of `strfunc` in a BEGIN PROGRAM-END PROGRAM block.

```
*python_trans_demo.sps.
DATA LIST LIST (,) /nvar (F) first (A30) last (A30).
BEGIN DATA
0,Rick,Arturo
1,Nancy,McNancy
-1,Yvonne,Walker
END DATA.

BEGIN PROGRAM.
import trans, math

def strfunc(pre,x,y):
    """ Concatenate a specified prefix and the first character
        of each argument. """
    return pre+"_"+x[0]+y[0]

tproc = trans.Tfunction()
tproc.append(strfunc,'strout','a8',[trans.const('cust'),'first','last'])
tproc.append(math.sinh,'numout','f',['nvar'])
tproc.execute()
END PROGRAM.
```

- The `import` statement includes the `trans` module and any modules that contain the Python functions you're using. In this example, we're using a function from the `math` module, which is always available with the Python language.

- `trans.Tfunction()` creates an instance of the `Tfunction` class, which is then stored to the Python variable *tproc*.

- The `append` method from the `Tfunction` class is used to specify the set of Python functions to be applied on the associated data pass. Functions are executed in the order in which they are appended.

  The first argument is the function name. Functions from an imported module must be specified with the module name, as in the current example, unless they were imported using `from module import <function>`.

  The second argument is a string, or a sequence of strings, specifying the names of the variables that will contain the results of the function.

  The third argument specifies the format(s) for the resulting variable(s). Formats should be given as strings that specify SPSS Statistics variable formats—for example, `'f8.2'` or `'a8'`—except `'f'` without explicit width or decimal specifications.

  The fourth argument is a sequence of strings naming the inputs to the function. These may be the names of variables in the active dataset, variables created by preceding functions applied in the same data pass, or Python expressions. The inputs must be compatible with the inputs expected by the function, both in number and type.

- In the present example, the Python function `strfunc` requires three arguments, so the call to `append` for `strfunc` contains a list with three elements, one for each argument. The first argument specifies the string constant `'cust'`. Python string expressions, to be passed as arguments, are specified as `trans.const(expression)` to distinguish them from strings representing variable names. The remaining two arguments specify variable names. The active dataset is assumed to contain the string variables *first* and *last*. The result is stored to the new string variable *strout*, which has a width of 8.

  Note: You may want to include the statement `from trans import const`, which allows you to use `const(expression)` instead of `trans.const(expression)` when specifying scalar arguments such as string constants.

- The Python function `sinh` from the `math` module requires a single argument. In this example, the argument is the variable *nvar*, which is assumed to be numeric. The result—the hyperbolic sine of the input variable—is stored to the new numeric variable *numout*.

- The `execute` method from the `Tfunction` class initiates a data pass, applying the specified functions to the active dataset. Any pending transformations are executed on the same data pass before applying the specified functions.

Figure 19-1
Resulting dataset

Missing Values

The `Tfunction` class provides options for handling missing values encountered in the case data.

- By default, the `Tfunction` class converts user-missing values to the Python data type *None* before applying the specified functions to the data (system missing values are always converted to *None*). You can override the conversion by using `Tfunction(convertUserMissing=False)` when instantiating the class.

- By default, the specified functions are applied to each case in the active dataset (filtered for any case selection), regardless of whether any variables used as inputs are system- or user-missing. You can specify that cases with system- or user-missing input values, in variables used by the functions, are excluded by using `Tfunction(listwiseDeletion=True)` when instantiating the class. When `listwiseDeletion=True`, output variables are set to system-missing for cases with missing input values. If you choose to use the default behavior, it is your responsibility to handle any system missing values in the case data—they are represented in Python as *None*.

- Python *None* values are converted to system-missing for output variables specified with a numeric format and to blanks for output variables specified with a string format.

In addition, you can use the `ismissing` function (included in the `trans` module) in your Python functions to identify missing values, allowing you to take specific actions when such values are encountered.

```
*python_trans_ismissing.sps.
DATA LIST FREE /nvar1 (F) nvar2 (F).
BEGIN DATA
1,2
3,4
5,
7,8
END DATA.

BEGIN PROGRAM.
import trans

def demo(val1,val2):
    """ Return the sum of the arguments. Arguments for which the
    case value is user- or system-missing are set to 0.
    """
    if trans.ismissing(trans.getargnames()[0],val1):
        val1=0
    if trans.ismissing(trans.getargnames()[1],val2):
        val2=0
    return val1 + val2

tproc = trans.Tfunction()
tproc.append(demo, 'result','f', ['nvar1','nvar2'])
tproc.execute()
END PROGRAM.
```

- The Python function `demo` returns the sum of its two input values. A value of 0 is used in place of any input value that is user- or system-missing. For simplicity, the function definition is included in the `BEGIN PROGRAM-END PROGRAM` block.

■ The `ismissing` function is included in `demo` to detect missing values. It requires two arguments: the name of the variable being tested for missing values and the value being tested. It returns *True* if the value is user- or system-missing and *False* otherwise.

■ In this example, the variables being tested for missing values are those used as inputs to `demo`. The names of these variables are obtained from the `getargnames` function, which returns a list containing the names of the arguments to the function currently being executed in the data pass controlled by the `Tfunction` class. In this case, `trans.getargnames()[0]` is `'nvar1'` and `trans.getargnames()[1]` is `'nvar2'`.

Figure 19-2
Resulting dataset

Performing Initialization Tasks before Passing Data

Sometimes a function used to transform data needs to be initialized before the data are passed. You can create a class that does this initialization and creates or contains the function to be applied to the cases. After the constructor call, the class must contain a function named *'func'* taking the same argument list as the constructor. For an example, see the source code for the `subs` function in the `extendedTransforms` module, available from Developer Central.

Tracking Function Calls

By default, a variable attribute recording each function call (for a given instance of `Tfunction`) is created for each output variable. The attribute name is *$Py.Function*. The attribute value contains the name of the function and the names of the input variables. You can disable this feature by setting `autoAttrib=False` when creating an instance of the `Tfunction` class.

For more information on the `Tfunction` class, use `help(trans.Tfunction)` after importing the `trans` module.

Using Functions from the extendedTransforms Module

The Python module `extendedTransforms`, available for download from Developer Central, includes a number of functions that provide transformations not available with the IBM® SPSS® Statistics transformation system. These functions are intended for use with the framework provided by the `Tfunction` class in the `trans` module but can also be used independently. It is suggested that you read the section "Getting Started with the trans Module" on p. 305 before using these functions with the framework in the `trans` module.

The search and subs Functions

The search and subs functions allow you to search for and replace patterns of characters in case data through the use of regular expressions. Regular expressions define patterns of characters that are then matched against a string to determine if the string contains the pattern. For example, you can use a regular expression to identify cases that contain a sequence of characters, such as a particular area code or Internet domain, or perhaps you want to find all cases for a specified string variable that contain one or more of the decimal digits 0–9.

Regular expressions are a powerful, specialized programming language for working with patterns of characters. For example, the regular expression [0-9] specifies a single decimal digit between 0 and 9 and will match any string containing one of these characters. If you are not familiar with the syntax of regular expressions, a good introduction can be found in the section "Regular expression operations" in the Python Library Reference, available at *http://docs.python.org/lib/module-re.html*.

Note: The search and subs functions are automatically locale sensitive.

Using the search Function

The search function applies a regular expression to a string variable and returns the part of the string that matches the pattern. It also returns the starting position of the matched string (the first character is position 0) and the length of the match, although these values can be ignored in the calling sequence. By default, the search is case sensitive. You can ignore case by setting the optional parameter *ignorecase* to *True*.

Example

In this example, we'll use the search function to extract the five-digit zip code from an address that is provided as a single string.

```
*python_extendedTransforms_search.sps.
DATA LIST /address (A50).
BEGIN DATA
374 Main St., Sunville, NV 45768-8765
25647 Longview Rd., Plainville, MT 78987
1121 Carmel Circle, Peru, IN, 37432
END DATA.

BEGIN PROGRAM.
import trans, extendedTransforms
from trans import const
tproc = trans.Tfunction()
tproc.append(extendedTransforms.search,
             'zip',
             'A5',
             ['address',const(r"\b\d{5}\b(-\d{4})?\s*\Z")])
tproc.execute()
END PROGRAM.
```

■ The first argument to the search function is the string to search, and the second argument is the regular expression. In this example, the search function is used with the Tfunction class so that the search can be performed in a casewise fashion. Values of the variable

address in the active dataset will be searched for a match to the regular expression `\b\d{5}\b(-\d{4})?\s*\Z`. The result is stored to the new variable *zip*.

- When used as an argument to the `append` method of the `Tfunction` class, a regular expression is specified as a string expression using `const(expression)`. The `r` preceding the regular expression in this example specifies a raw string, which ensures that any character sets specifying Python escape sequences—such as `\b`, which is the escape sequence for a backspace—are treated as raw characters and not the corresponding escape sequence.

- The regular expression used here will match a sequence of five digits set off by white space or a non-alphanumeric and non-underscore character (`\b\d{5}\b`), followed by an optional five-character sequence of a dash and four digits (`(-\d{4})?`), optional white space (`\s*`), and the end of the string (`\Z`).

Figure 19-3
Resulting dataset

To obtain the starting location (the first position in the string is 0) and length of the matched string, use three output variables for the function, as in:

```
tproc.append(extendedTransforms.search,
            ['zip','location','length'],
            ['A5','F','F'],
            ['address',const(r"\b\d{5}\b(-\d{4})?\s*\Z")])
```

For more information on the `search` function, use `help(extendedTransforms.search)` after importing the `extendedTransforms` module.

Using the subs Function

The `subs` function searches a string for a match to a regular expression and replaces matched occurrences with a specified string. By default, the search is case sensitive. You can ignore case by setting the optional parameter *ignorecase* to *True*.

Example

In this example, we'll use the `subs` function to create a string that has the form `'<last name>, <first name>'` from one of the form `'<first name> <last name>'`.

```
*python_extendedTransforms_subs.sps.
DATA LIST /var (A30).
BEGIN DATA
Ari Prime
Jonathan Poe
Gigi Sweet
END DATA.

BEGIN PROGRAM.
import trans, extendedTransforms
from trans import const
tproc = trans.Tfunction()
tproc.append(extendedTransforms.subs,
             'newvar',
             'A20',
             ['var',
             const(r'(?P<first>\S+)\s+(?P<last>\S+)'),
             const(r'\g<last>, \g<first>')])
tproc.execute()
END PROGRAM.
```

- The first argument to the subs function is the string on which to perform substitutions, the second argument is the regular expression, and the third argument is the string to substitute for matched values. In this example, the subs function is used with the Tfunction class so that the substitution can be performed in a casewise fashion.

- When used as an argument to the append method of the Tfunction class, a regular expression is specified as a string expression using const(expression). The r preceding the regular expression in this example specifies a raw string, which ensures that any character sets specifying Python escape sequences are treated as raw characters and not the corresponding escape sequence.

- Values of the variable *var* in the active dataset will be searched for a match to the regular expression (?P<first>\S+)\s+(?P<last>\S+), which will match two words separated by white space. The general regular expression code (?P<name>\S+) matches a sequence of nonwhitespace characters and makes the matched string accessible via the specified name. In this example, the first word is accessible via the name first and the second word is accessible via the name last.

- The replacement string is given by the expression \g<last>, \g<first>. The general code \g<name> will substitute the string matched by the expression associated with the specified name. In this example, \g<last> will substitute the second word and \g<first> will substitute the first word from the original string.

- By default, all occurrences in a given string are replaced. You can specify the maximum number of occurrences to replace with the optional parameter *count* to the subs function.

Figure 19-4
Resulting dataset

For more information on the `subs` function, use `help(extendedTransforms.subs)` after importing the `extendedTransforms` module.

The templatesub Function

The `templatesub` function substitutes variable values or constants into the template for a string and returns the completed string.

Example

In this example, the `templatesub` function is used to create string variables derived from a dynamically determined set of variables from the active datatset. The template used to construct the strings uses variable values and a variable label.

```
*python_extendedTransforms_templatesub.sps.
DATA LIST FREE / store1 (A15) store2 (A15) weekday (A10).
BEGIN DATA
gorgonzola jarlsberg mondays
roquefort cheddar tuesdays
stilton gouda wednesdays
brie edam thursdays
camembert parmesan fridays
END DATA.

VARIABLE LABELS store1 'Main St.' store2 'Village Green'.

BEGIN PROGRAM.
import trans, extendedTransforms, spssaux
from trans import const
varDict = spssaux.VariableDict()
storeList = varDict.variablesf(pattern=r'store')
template = "The $loc store is out of $type on $day."
tproc = trans.Tfunction()
for store in storeList:
    loc = varDict[store].VariableLabel
    tproc.append(extendedTransforms.templatesub,
              store+'_news',
              'A60',
              [const(template),const(loc),store,'weekday'])
tproc.execute()
END PROGRAM.
```

- This example makes use of the `VariableDict` class from the `spssaux` module (a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python) to obtain the list of variables from the active dataset whose names begin with the string `'store'`. The list is stored to the Python variable *storeList*. For more information, see the topic "Getting Started with the VariableDict Class" in Chapter 14 on p. 225.

- The `for` loop iterates through the list of variables in *storeList*. Each iteration of the loop specifies the creation of a new string variable using the `templatesub` function. The `templatesub` function is used with the `Tfunction` class so that the substitution can be performed in a casewise fashion. The new variables are created when the data pass is executed with `tproc.execute()`.

- The code `varDict[store].VariableLabel` is the variable label associated with the value of *store*. The label contains the store location and is stored to the Python variable *loc*.

■ The first argument to the `templatesub` function is the template, specified as a string. A template consists of text and field names that mark the points at which substitutions are to be made. Field names are strings starting with `$`. In this example, the template is stored in the Python variable *template*. Values will be substituted for the fields *$loc*, *$type*, and *$day* in the template. Fields in the template are matched in order with the sequence of variables or constants following the template, in the argument set passed to the `templatesub` function. The first field in the template matches the first variable or constant, and so on. If a field name occurs more than once, its first occurrence determines the order.

■ On each iteration of the loop, the value of the Python variable *loc* is substituted for the template field *$loc*, casewise values of the variable specified by *store* will be substituted for *$type*, and casewise values of the variable *weekday* will be substituted for *$day*. The resulting string is stored to a new variable whose name is dynamically created by the expression `store+'_news'`—for example, *store1_news*.

Figure 19-5
Resulting dataset

Notes

■ If a field name in the template is followed by text that might be confused with the name, enclose the field name in `{}`, as in `${day}`.

■ Field values are converted to strings if necessary, and trailing blanks are trimmed.

For more information on the `templatesub` function, use `help(extendedTransforms.templatesub)` after importing the `extendedTransforms` module.

The levenshteindistance Function

The `levenshteindistance` function calculates the Levenshtein distance between two strings, after removing trailing blanks. The Levenshtein distance between two strings is the minimum number of operations (insertion, deletion, or substitutions) required to transform one string into the other. Case is significant in counting these operations. Identical strings have distance zero, and larger distances mean greater differences between the strings.

Example

```
*python_extendedTransforms_levenshtein.sps.
DATA LIST FREE /str1 (A8) str2 (A8).
BEGIN DATA
untied united
END DATA.

BEGIN PROGRAM.
import trans, extendedTransforms
tproc = trans.Tfunction()
tproc.append(extendedTransforms.levenshteindistance,
             'ldistance',
             'f',
             ['str1','str2'])
tproc.execute()
END PROGRAM.
```

The `levenshteindistance` function takes two arguments, the two strings to compare. In this example, the function is used with the `Tfunction` class so that the analysis can be performed in a casewise fashion. The result is stored to the new variable *ldistance*. For the single case shown here, the Levenshtein distance is 2.

For more information on the `levenshteindistance` function, use `help(extendedTransforms.levenshteindistance)` after importing the `extendedTransforms` module.

The soundex and nysiis Functions

The `soundex` and `nysiis` functions implement two popular phonetic algorithms for indexing names by their sound as pronounced in English. The purpose is to encode names having the same pronunciation to the same string so that matching can occur despite differences in spelling.

Example

```
*python_extendedTransforms_soundex.sps.
DATA LIST FREE /name (A20).
BEGIN DATA
Abercromby
Abercrombie
END DATA.

BEGIN PROGRAM.
import trans, extendedTransforms
tproc = trans.Tfunction(listwiseDeletion=True)
tproc.append(extendedTransforms.soundex,'soundex','A20',['name'])
tproc.append(extendedTransforms.nysiis,'nsyiis','A20',['name'])
tproc.execute()
END PROGRAM.
```

- The single argument to the `soundex` and `nysiis` functions is the string to encode. In this example, the function is used with the `Tfunction` class so that the analysis can be performed in a casewise fashion. The results are stored to the new variables *soundex* and *nsyiis*.

- The two spellings *Abercromby* and *Abercrombie* are phonetically the same and are encoded to the same value.

Figure 19-6
Resulting dataset

If you need to encode strings containing multiple words, consider using the
`soundexallwords` function. It transforms each word in a string of free text into
its soundex value and returns a string of blank-separated soundex values. For more
information, use `help(extendedTransforms.soundexallwords)` after importing the
`extendedTransforms` module.

The strtodatetime Function

The `strtodatetime` function converts a string value to a datetime value according to a specified
pattern. If a value does not match the pattern, the function returns *None*. Patterns are constructed
from a set of format codes representing pieces of a datetime specification, such as day of month,
year with century, hour, and so on. The large set of available format codes and the ability to
specify which formats are used in a given pattern greatly extends the limited set of datetime
formats available with command syntax.

Example

```
*python_extendedTransforms_strtodatetime.sps.
DATA LIST FIXED/strdatetime (A20).
BEGIN DATA
DEC 7, 2006 12:31
END DATA.

BEGIN PROGRAM.
import spss, extendedTransforms, trans
from trans import const
tproc = trans.Tfunction()
tproc.append(extendedTransforms.strtodatetime,
             'datetime',
             'DATETIME17',
             ['strdatetime',const("%b %d, %Y %H:%M ")])
tproc.execute()
END PROGRAM.
```

- The first argument to the `strtodatetime` function is the string to convert. The second
  argument is the pattern describing the datetime format, given as a string. In this example,
  we're converting a string variable containing dates of the form `'mmm dd, yyyy hh:mm'`.
  The associated pattern is `"%b %d, %Y %H:%M "`. Delimiters, such as commas, contained
  in the string to convert should also be included in the pattern as was done here. A single
  blank in the pattern matches any amount of white space. In particular, the single blank at
  the end of the pattern is required to match any trailing blanks in the string. A partial list
  of the allowed patterns, along with more usage details, is provided in the documentation

for the `strtodatetime` function, which can be viewed by including the statement `help(extendedTransforms.strtodatetime)` in a program block, after importing the `extendedTransforms` module.

■ In this example, the `strtodatetime` function is used with the `Tfunction` class so that the substitution can be performed in a casewise fashion. The converted string values are stored to the datetime variable *datetime* with a format of `DATETIME17`. The pattern argument to the `strtodatetime` function is a string constant, so it is specified with `const()`.

The datetimetostr Function

The `datetimetostr` function converts a datetime value to a string according to a specified pattern. Values that can't be converted are returned as a blank. Patterns are constructed from a set of format codes representing pieces of a datetime specification, such as day of month, year with century, hour, and so on. The large set of available format codes and the ability to specify which formats are used in a given pattern greatly extends the limited set of datetime formats available with command syntax.

Example

```
*python_extendedTransforms_datetimetostr.sps.
DATA LIST FIXED/dtime (DATETIME17).
BEGIN DATA
06-DEC-2006 21:50
END DATA.

BEGIN PROGRAM.
import spss, extendedTransforms, trans
from trans import const
tproc = trans.Tfunction()
tproc.append(extendedTransforms.datetimetostr,
             'strdtime',
             'A30',
             ['dtime',const("%b %d, %Y %I:%M %p")])
tproc.execute()
END PROGRAM.
```

■ The first argument to the `datetimetostr` function is the datetime value to convert. The second argument is the pattern describing the resulting string. In this example, we're converting a datetime variable with a date and time format to a string of the form `'mmm dd, yyyy hh:mm p'`, where `p` specifies AM or PM (or the current locale's equivalent). The associated pattern is `"%b %d, %Y %I:%M %p"`. Delimiters, such as commas, included in the pattern will be included in the result, as in this example. A partial list of the allowed patterns is provided in the documentation for the `strtodatetime` function, which can be viewed by including the statement `help(extendedTransforms.strtodatetime)` in a program block, after importing the `extendedTransforms` module.

■ In this example, the `datetimetostr` function is used with the `Tfunction` class so that the substitution can be performed in a casewise fashion. The converted datetime values are stored to the string variable *strdtime*. The pattern argument to the `datetimetostr` function is a string constant so it is specified with `const()`.

The lookup Function

The `lookup` function performs a table lookup given a key value and a Python dictionary containing keys and associated values.

Example

In this example, we look up state names given the two-letter state code.

```
*python_extendedTransforms_lookup.sps.
DATA LIST LIST (",")/street (A30) city (A30) st (A2) zip(A10).
BEGIN DATA
222 Main St,Springfield,IL,12345
919 Locust Lane,Treeville,IN,90909
11 Linden Lane,Deepwoods,,44074
47 Briar Patch Parkway,Riverdale,MD,07000
END DATA.

BEGIN PROGRAM.
import extendedTransforms, trans
from trans import const
statedict = {"IL":"Illinois", "IN":"Indiana","MD":"Maryland",
             "DC":"District of Columbia","CT":"Connecticut",
             "RI":"Rhode Island","MA":"Massachusetts"}
tproc = trans.Tfunction(autoAttrib=False)
tproc.append(extendedTransforms.lookup,
             'statename',
             'a24',
             ['st',const(statedict),const("")])
tproc.execute()
END PROGRAM.
```

- The Python variable *statedict* is a Python dictionary whose keys are the two-letter states codes and whose values are the associated state names.

- The first argument to the `lookup` function is the key whose value is to be returned. If it is a string, trailing blanks are removed. In this example, the argument is the two-letter state code given by the variable *st*. The second argument is the Python dictionary containing the keys and associated values. The third argument is the value to return if the key is not found in the dictionary—in this example, a blank string.

- In this example, the `lookup` function is used with the `Tfunction` class so that the substitution can be performed in a casewise fashion. The full state name returned from the table lookup is stored to the string variable *statename*. Both the second and third arguments to the `lookup` function are specified with `const()`, which is used to distinguish scalar arguments from variable names. In this case, there are two scalar arguments—the name of the Python dictionary *statedict* and the blank string.

- The optional argument *autoAttrib* to the `Tfunction` class is set to *False* to suppress the creation of a variable attribute associated with the output variable *statename*. Variable attributes are provided for tracking purposes but can become very verbose when associated with the `lookup` function because the attribute contains the full dictionary used for the lookup. An alternative to suppressing the attribute is to specify a maximum length, as in `autoAttrib=50`.

For more information, use `help(extendedTransforms.lookup)` after importing the `extendedTransforms` module.

Modifying and Exporting Output Items

The IBM® SPSS® Statistics - Integration Plug-In for Python provides the ability to customize pivot tables and export items, such as charts and tables, in a variety of formats. This is accomplished using the `SpssClient` module.

The `SpssClient` module can be used in a standalone Python script that you can manually invoke as needed or it can be used within a Python program to modify or export output generated by the program. For more information, see the topic "The SpssClient Python Module" in Chapter 12 on p. 178.

Modifying Pivot Tables

The `SpssPivotTable` class, from the `SpssClient` module, provides methods to customize pivot tables in output documents. As an example, we'll create a Python script that changes the text style of specified column labels to bold for a chosen set of pivot tables. In particular, we'll change the column label *Mean* to bold for all Descriptive Statistics tables in the designated output document.

```
#MakeColLabelBold.py.
import SpssClient
SpssClient.StartClient()

itemlabel = "Descriptive Statistics"
collabel = "Mean"
OutputDoc = SpssClient.GetDesignatedOutputDoc()
OutputItems = OutputDoc.GetOutputItems()

for index in range(OutputItems.Size()):
    OutputItem = OutputItems.GetItemAt(index)
    if (OutputItem.GetType() == SpssClient.OutputItemType.PIVOT) and \
       (OutputItem.GetDescription() == itemlabel):
        PivotTable = OutputItem.GetSpecificType()
        SpssLabels = PivotTable.ColumnLabelArray()
        for i in range(SpssLabels.GetNumRows()):
            for j in range(SpssLabels.GetNumColumns()):
                if SpssLabels.GetValueAt(i,j)==collabel:
                    SpssLabels.SelectLabelAt(i,j)
        PivotTable.SetTextStyle(SpssClient.SpssTextStyleTypes.SpssTSBold)
SpssClient.StopClient()
```

■ The `GetDesignatedOutputDoc` method of the `SpssClient` class returns an object representing the designated output document (the current document to which output is routed). The `GetOutputItems` method of the output document object returns a list of objects representing the items in the output document, such as pivot tables, charts, and log items.

■ The `for` loop iterates through the list of items in the output document. The `GetType` method of an output item object returns the type of item. Pivot tables are identified as an output item type of `SpssClient.OutputItemType.PIVOT`. The `GetDescription` method of an output item object returns the name of the output item as it appears in the outline pane of the Viewer.

- If a Descriptive Statistics table is found, you call the `GetSpecificType` method on the output item object to get an object representing the pivot table. In this example, the Python variable *PivotTable* is an instance of the `SpssPivotTable` class.

- The `ColumnLabelArray` method of the pivot table object returns an object that provides access to the column label array. The `GetNumRows` and `GetNumColumns` methods of the object return the number of rows and columns in the column label array.

- The inner `for` loops indexed by *i* and *j* iterate through the elements in the column label array. The `GetValueAt` method is used to access the value of a specified column label. If the label matches the specified value of *Mean*, the label cell is selected using the `SelectLabelAt` method.

- The `SetTextStyle` method of the pivot table object is called to set the text style of any selected labels to bold.

Note: You may also want to consider using the SPSSINC MODIFY TABLES extension command to customize your pivot table output. The command is available from Developer Central at *http://www.spss.com/devcentral/*.

Exporting Output Items

Using the `SpssClient` module, you can export multiple items, including charts and tables, to a single file in a variety of formats, including: Word, Excel, PowerPoint (Windows operating systems only), PDF, and HTML. You can also export chart items to individual image files in many formats, including: JPG, PNG, TIFF, and BMP.

This section covers only a few of the available formats and options. For more information, see the descriptions of the `ExportDocument` and `ExportCharts` methods in the IBM® SPSS® Statistics Help system. You can also export output items using the OUTPUT EXPORT or OMS commands within command syntax. For more information, see the topic "Exporting Results" in Chapter 9 on p. 137.

Example: Exporting All Items

In this example, we create a Python script to export the contents of the designated output document to a PDF file.

```
#ExportAllToPDF.py
import SpssClient, sys
SpssClient.StartClient()
OutputDoc = SpssClient.GetDesignatedOutputDoc()
try:
   OutputDoc.ExportDocument(SpssClient.SpssExportSubset.SpssAll,
                            "/temp/output1.pdf",
                            SpssClient.DocExportFormat.SpssFormatPdf)
except:
   print sys.exc_info()[1]
SpssClient.StopClient()
```

- The script utilizes the standard module `sys` (used here to extract information about an exception), so it includes `sys` in the `import` statement.

■ The `ExportDocument` method of the output document object performs the export. The first argument specifies whether all items, all selected items, or all visible items are exported. In this example, all items are exported, as specified by `SpssClient.SpssExportSubset.SpssAll`. The second argument specifies the destination file. The third argument specifies the export format—in this example, PDF, as specified by `SpssClient.DocExportFormat.SpssFormatPdf`.

■ If the save attempt fails for any reason, the except clause is invoked. `sys.exc_info()` returns a tuple of three values that provides information about the current exception. The value with an index of 1 contains the most descriptive information.

You can export a specified output item using the `ExportToDocument` (nonchart items) and `ExportToImage` (chart items) methods from the `SpssOutputItem` class.

Example: Exporting All Charts

In this example, we create a Python script to export all charts in the designated output document to PNG files.

```
#ExportChartsToPNG.py
import SpssClient, sys
SpssClient.StartClient()
OutputDoc = SpssClient.GetDesignatedOutputDoc()
try:
   OutputDoc.ExportCharts(SpssClient.SpssExportSubset.SpssAll,
                          "/temp/chart_",
                          SpssClient.ChartExportFormat.png)
except:
   print sys.exc_info()[1]
SpssClient.StopClient()
```

■ The `ExportCharts` method of the output document object performs the export. The first argument specifies whether all items, all selected items, or all visible items are exported. In this example, all items are exported, as specified by `SpssClient.SpssExportSubset.SpssAll`. The second argument specifies the path and prefix for the destination files (each chart is exported to a separate file). The third argument specifies the export format—in this example, PNG, as specified by `SpssClient.ChartExportFormat.png`.

You can export a specified chart using the `ExportToImage` method from the `SpssOutputItem` class.

Example: Exporting To Excel

When exporting to Excel, you can specify the sheet name in a new workbook, create a new sheet in an existing workbook, or append output to an existing sheet. In this example, we will export any output from the Descriptives and Frequencies procedures to worksheets named *Descriptives* and *Frequencies* in a new workbook.

```
#ExportToExcel.py
import SpssClient,sys
SpssClient.StartClient()
OutputDoc = SpssClient.GetDesignatedOutputDoc()
OutputDoc.ClearSelection()

# Create a new workbook and export all Descriptives output to a worksheet
# named Descriptives.
OutputDoc.SetOutputOptions(SpssClient.DocExportOption.ExcelOperationOptions,
                           "CreateWorkbook")
OutputDoc.SetOutputOptions(SpssClient.DocExportOption.ExcelSheetNames,
                           "Descriptives")
OutputItems = OutputDoc.GetOutputItems()
for index in range(OutputItems.Size()):
    OutputItem = OutputItems.GetItemAt(index)
    if (OutputItem.GetType() == SpssClient.OutputItemType.HEAD and
        OutputItem.GetDescription() == "Descriptives"):
            OutputItem.SetSelected(True)
try:
    OutputDoc.ExportDocument(SpssClient.SpssExportSubset.SpssSelected,
                             "/temp/myexport.xls",
                             SpssClient.DocExportFormat.SpssFormatXls)
except:
   print sys.exc_info()[1]
OutputDoc.ClearSelection()

# Export all Frequencies output to a new worksheet named Frequencies.
OutputDoc.SetOutputOptions(SpssClient.DocExportOption.ExcelOperationOptions,
                           "CreateWorksheet")
OutputDoc.SetOutputOptions(SpssClient.DocExportOption.ExcelSheetNames,
                           "Frequencies")
for index in range(OutputItems.Size()):
    OutputItem = OutputItems.GetItemAt(index)
    if (OutputItem.GetType() == SpssClient.OutputItemType.HEAD and
        OutputItem.GetDescription() == "Frequencies"):
            OutputItem.SetSelected(True)
try:
    OutputDoc.ExportDocument(SpssClient.SpssExportSubset.SpssSelected,
                             "/temp/myexport.xls",
                             SpssClient.DocExportFormat.SpssFormatXls)
except:
   print sys.exc_info()[1]
OutputDoc.ClearSelection()
SpssClient.StopClient()
```

■ The `SetOutputOptions` method of the output document object specifies details of the
  export such as whether output will be written to a new or existing worksheet as well as
  specifying the name of the worksheet. The first step is to specify a new workbook and to
  direct export to a worksheet named *Descriptives* in that workbook. This is accomplished by
  the first two calls to `SetOutputOptions`.

  The first argument to `SetOutputOptions` specifies the type of
  option and the second argument specifies the value of the option.
  `SpssClient.DocExportOption.ExcelOperationOptions` indicates
  whether a new workbook is created, a new worksheet is created, or an existing worksheet is
  modified. The associated value `"CreateWorkbook"` specifies a new workbook.

  The option `SpssClient.DocExportOption.ExcelSheetNames` specifies the name of
  the worksheet to which export is directed. The associated value specifies the worksheet
  named *Descriptives*.

■ The first `for` loop iterates through all of the items in the output document and selects all
  header items named *"Descriptives"*. For descriptions of the methods used in the loop, see
  "Modifying Pivot Tables" on p. 319.

■ The selected items are then exported using the `ExportDocument` method.
The first argument specifies to export all selected items, as given by
`SpssClient.SpssExportSubset.SpssSelected`. The second argument specifies
the destination file. The third argument specifies export to Excel, as given by
`SpssClient.DocExportFormat.SpssFormatXls`.

■ The `ClearSelection` method deselects all selected output items in preparation for selecting
the *"Frequencies"* items.

■ The `SetOutputOptions` method is then called twice. The first call specifies that output will
be directed to a new worksheet, and the second call specifies that the name of the worksheet is
Frequencies. Note that these calls to `SetOutputOptions` override the previous settings.

■ The second `for` loop iterates through all of the items in the output document and selects
all header items named *"Frequencies"*. The selected items are then exported to the new
worksheet using the `ExportDocument` method.

Example: Handling Export of Wide Tables

When exporting to Word or PowerPoint, you can control the display of wide tables. In this
example, we will export any output from the Crosstabs procedure to a Word document, and
specify that wide tables be reduced to fit within the document width.

```
#ExportToWord.py
import SpssClient,sys
SpssClient.StartClient()
OutputDoc = SpssClient.GetDesignatedOutputDoc()
OutputDoc.ClearSelection()
OutputDoc.SetOutputOptions(SpssClient.DocExportOption.WideTablesOptions,
                           "WT_Shrink")
OutputItems = OutputDoc.GetOutputItems()
for index in range(OutputItems.Size()):
    OutputItem = OutputItems.GetItemAt(index)
    if (OutputItem.GetType() == SpssClient.OutputItemType.HEAD and
        OutputItem.GetDescription() == "Crosstabs"):
            OutputItem.SetSelected(True)
try:
    OutputDoc.ExportDocument(SpssClient.SpssExportSubset.SpssSelected,
                             "/temp/myexport.doc",
                             SpssClient.DocExportFormat.SpssFormatDoc)
except:
   print sys.exc_info()[1]

OutputDoc.ClearSelection()
SpssClient.StopClient()
```

■ The option `SpssClient.DocExportOption.WideTablesOptions` to the
`SetOutputOptions` method specifies the handling of pivot tables that are too wide for the
document width. The value `WT_Shrink` specifies that font size and column width are reduced
so that tables fit within the document width.

■ The `for` loop iterates through all of the items in the output document and selects all header
items named *"Crosstabs"*. For descriptions of the methods used in the loop, see "Modifying
Pivot Tables" on p. 319.

■ The selected items are then exported using the `ExportDocument` method. The
first argument specifies to export all selected items. The second argument specifies
the destination file. The third argument specifies export to Word, as given by
`SpssClient.DocExportFormat.SpssFormatDoc`.

The `SetOutputOptions` method also provides settings for specifying page dimensions. For details, see the SPSS Statistics Help system.

Tips on Migrating Command Syntax and Macro Jobs to Python

Exploiting the power that the IBM® SPSS® Statistics - Integration Plug-In for Python offers may mean converting an existing command syntax job or macro to Python. This is particularly straightforward for command syntax jobs, since you can run command syntax from Python using a function from the `spss` module (available once you install the plug-in). Converting macros is more complicated, since you need to translate from the macro language, but there are some simple rules that facilitate the conversion. This chapter provides a concrete example for each type of conversion and any general rules that apply.

Migrating Command Syntax Jobs to Python

Converting a command syntax job to run from Python allows you to control the execution flow based on variable dictionary information, case data, procedure output, or error-level return codes. As an example, consider the following simple syntax job that reads a file, creates a split on gender, and uses `DESCRIPTIVES` to create summary statistics.

```
GET FILE="/examples/data/Employee data.sav".
SORT CASES BY gender.
SPLIT FILE
  LAYERED BY gender.
DESCRIPTIVES
  VARIABLES=salary salbegin jobtime prevexp
  /STATISTICS=MEAN STDDEV MIN MAX.
SPLIT FILE OFF.
```

You convert a block of command syntax to run from Python simply by wrapping the block in triple quotes and including it as the argument to the `Submit` function in the `spss` module. For the current example, this looks like:

```
spss.Submit(r"""
GET FILE='/examples/data/Employee data.sav'.
SORT CASES BY gender.
SPLIT FILE
  LAYERED BY gender.
DESCRIPTIVES
  VARIABLES=salary salbegin jobtime prevexp
  /STATISTICS=MEAN STDDEV MIN MAX.
SPLIT FILE OFF.
""")
```

- The `Submit` function takes a string argument containing command syntax and submits the syntax to IBM® SPSS® Statistics for processing. By wrapping the command syntax in triple quotes, you can specify blocks of commands on multiple lines in the way that you might normally write command syntax. You can use either triple single quotes or triple double quotes, but you must use the same type (single or double) on both sides of the expression. If

your syntax contains a triple quote, be sure that it's not the same type that you are using to wrap the syntax; otherwise, Python will treat it as the end of the argument.

Note also that Python treats doubled quotes, contained within quotes of that same type, differently from SPSS Statistics. For example, in Python, `"string with ""quoted"" text"` is treated as `string with quoted text`. Python treats each pair of double quotes as a separate string and simply concatenates the strings as follows: `"string with "+"quoted"+" text"`.

■ Notice that the triple-quoted expression is prefixed with the letter `r`. The `r` prefix to a string specifies Python's raw mode. This allows you to use the single backslash (\) notation for file paths on Windows. That said, it is a good practice to use forward slashes (/) in file paths on Windows, since you may at times forget to use raw mode, and SPSS Statistics accepts a forward slash (/) for any backslash in a file specification. For more information, see the topic "Using Raw Strings in Python" in Chapter 13 on p. 202.

Having converted your command syntax job so that it can run from Python, you have two options: include this in a `BEGIN PROGRAM` block and run it from SPSS Statistics, or run it from a Python IDE (Integrated Development Environment) or shell. Using a Python IDE can be a very attractive way to develop and debug your code because of the syntax assistance and debugging tools provided. For more information, see the topic "Running Your Code from a Python IDE" in Chapter 12 on p. 176. To run your job from SPSS Statistics, simply enclose it in a `BEGIN PROGRAM-END PROGRAM` block and include an `import spss` statement as the first line in the program block, as in:

```
BEGIN PROGRAM.
import spss
spss.Submit(r"""
GET FILE='/examples/data/Employee data.sav'.
SORT CASES BY gender.
SPLIT FILE
  LAYERED BY gender.
DESCRIPTIVES
  VARIABLES=salary salbegin jobtime prevexp
  /STATISTICS=MEAN STDDEV MIN MAX.
SPLIT FILE OFF.
""")
END PROGRAM.
```

You have taken a command syntax job and converted it into a Python job. As it stands, the Python job does exactly what the SPSS Statistics job did. Presumably, though, you're going to all this trouble to exploit functionality that was awkward or just not possible with standard command syntax. For example, you may need to run your analysis on many datasets, some of which have a gender variable and some of which do not. For datasets without a gender variable, you'll generate an error if you attempt a split on gender, so you'd like to run `DESCRIPTIVES` without the split. Following is an example of how you might extend your Python job to accomplish this, leaving aside the issue of how you obtain the paths to the datasets. As in the example above, you have the option of running this from SPSS Statistics by wrapping the code in a program block, as shown here, or running it from a Python IDE.

```
*python_converted_syntax.sps.
BEGIN PROGRAM.
import spss
filestring = r'/examples/data/Employee data.sav'
spss.Submit("GET FILE='%s'."%(filestring))
for i in range(spss.GetVariableCount()):
   if spss.GetVariableLabel(i).lower()=='gender':
      genderVar=spss.GetVariableName(i)
      spss.Submit("""
      SORT CASES BY %s.
      SPLIT FILE
        LAYERED BY %s.
      """ %(genderVar,genderVar))
      break
spss.Submit("""
DESCRIPTIVES
  VARIABLES=salary salbegin jobtime prevexp
  /STATISTICS=MEAN STDDEV MIN MAX.
SPLIT FILE OFF.
""")
END PROGRAM.
```

■ The string for the GET command includes the expression %s, which marks the point at which a string value is to be inserted. The particular value to insert is taken from the % expression that follows the string. In this case, the value of the variable *filestring* replaces the occurrence of %s. Note that the same technique (using multiple substitutions) is used to substitute the gender variable name into the strings for the SORT and SPLIT FILE commands. For more information, see the topic "Dynamically Specifying Command Syntax Using String Substitution" in Chapter 13 on p. 200.

■ The example uses a number of functions in the spss module, whose names are descriptive of their function: GetVariableCount, GetVariableLabel, GetVariableName. These functions access the dictionary for the active dataset and allow for conditional processing based on dictionary information.

■ A SORT command followed by a SPLIT FILE command is run only when a gender variable is found.

Note: When working with code that contains string substitution (whether in a program block or a Python IDE), it's a good idea for debugging to turn on both PRINTBACK and MPRINT with the command SET PRINTBACK ON MPRINT ON. This will display the actual command syntax that was run.

Migrating Macros to Python

The ability to use Python to dynamically create and control command syntax renders IBM® SPSS® Statistics macros obsolete for most purposes. Macros are still important, however, for passing information from a BEGIN PROGRAM block so that it is available to command syntax outside of the block. For more information, see the topic "Mixing Command Syntax and Program Blocks" in Chapter 12 on p. 187. You can continue to run your existing macros, but you may want to consider converting some to Python, especially if you've struggled with limitations of the macro language and want to exploit the more powerful programming features available with

Python. There is no simple recipe for converting a macro to Python, but a few general rules will help get you started:

- The analog of a macro in SPSS Statistics is a Python user-defined function. A user-defined function is a named piece of code in Python that is callable and accepts parameters. For more information, see the topic "Creating User-Defined Functions in Python" in Chapter 13 on p. 203.

- A block of command syntax within a macro is converted to run in a Python function by wrapping the block in triple quotes and including it as the argument to the Submit function in the spss module. Macro arguments that form part of a command, such as a variable list, become Python variables whose value is inserted into the command specification using string substitution.

As an example, consider converting the following macro, which selects a random set of cases from a data file. Macro arguments provide the number of cases to be selected and the criteria used to determine whether a given case is included in the population to be sampled. We'll assume that you're familiar with the macro language and will focus on the basics of the conversion to Python.

```
SET MPRINT=OFF.
DEFINE !SelectCases (
    nb=!TOKENS(1)   /crit=!ENCLOSE('(',')')
  /FPath=!TOKENS(1) /RPath=!TOKENS(1))
GET FILE=!FPath.
COMPUTE casenum=$CASENUM.
DATASET COPY temp_save.
SELECT IF !crit.
COMPUTE draw=UNIFORM(1).
SORT CASES BY draw.
N OF CASES !nb.
SORT CASES BY casenum.
MATCH FILES FILE=*
   /IN=ingrp
   /FILE=temp_save
   /BY=casenum
   /DROP=draw casenum.
SAVE OUTFILE=!RPath.
DATASET CLOSE temp_save.
!ENDDEFINE.

SET MPRINT=ON.
!SelectCases nb=5  crit=(gender='m' AND jobcat=1 AND educ<16)
       FPath= '/examples/data/employee data.sav'
       RPath= '/temp/results.sav'.
```

- The name of the macro is *SelectCases*, and it has four arguments: the number of cases to select, the criteria to determine if a case is eligible for selection, the name and path of the source data file, and the result file.

- In terms of the macro language, this macro is very simple, since it consists only of command syntax, parts of which are specified by the arguments to the macro.

- The macro call specifies a random sample of five cases satisfying the criteria specified by *crit*. The name and path of the source data file and the result file are provided as *FPath* and *RPath*, respectively.

The macro translates into the following Python user-defined function:

```
def SelectCases(nb,crit,FPath,RPath):
    """Select a random set of cases from a data file using a
    specified criteria to determine whether a given case is
    included in the population to be sampled.
    nb is the number of cases to be selected.
    crit is the criteria to use for selecting the sample population.
    FPath is the path to the source data file.
    RPath is the path to the result file.
    """
    spss.Submit("""
GET FILE='%(FPath)s'.
COMPUTE casenum=$CASENUM.
DATASET COPY temp_save.
SELECT IF %(crit)s.
COMPUTE draw=UNIFORM(1).
SORT CASES BY draw.
N OF CASES %(nb)s.
SORT CASES BY casenum.
MATCH FILES FILE=*
    /IN=ingrp
    /FILE=temp_save
    /BY=casenum
    /DROP=draw casenum.
SAVE OUTFILE="%(RPath)s".
DATASET CLOSE temp_save.
"""%locals())
```

- The `def` statement signals the beginning of a function definition—in this case, the function named *SelectCases*. The colon at the end of the `def` statement is required.

- The function takes the same four arguments as the macro. Note, however, that you simply specify the names of the arguments. No other defining characteristics are required, although Python supports various options for specifying function arguments, such as defining a default value for an optional argument.

- The body of the macro consists solely of a block of command syntax. When converting the macro to Python, you simply enclose the block in triple quotes and include it as the argument to the `Submit` function. The `Submit` function—a function in the `spss` module—takes a string argument containing command syntax and submits the syntax to SPSS Statistics for processing. Enclosing the command syntax in triple quotes allows you to specify a block of commands that spans multiple lines without having to be concerned about line continuation characters.

- Notice that the code within the Python function is indented. Python uses indentation to specify the grouping of statements, such as the statements in a user-defined function. Had the code not been indented, Python would process the function as consisting only of the `def` statement, and an exception would occur.

- The points in the command syntax where macro arguments occurred, such as `SELECT IF !crit`, translate to specifications for string substitutions in Python, such as `SELECT IF %(crit)s`. To make the conversion more transparent, we've used the same names for the arguments in the Python function as were used in the macro. Using the `locals` function for the string substitution, as in `%locals()`, allows you to insert the value of any locally defined variable into the string simply by providing the name of the variable. For example, the value of the variable *crit* is inserted at each occurrence of the expression `%(crit)s`. For more information, see the topic "Dynamically Specifying Command Syntax Using String Substitution" in Chapter 13 on p. 200.

Once you've translated a macro into a Python user-defined function, you'll want to include the function in a Python module on the Python search path. You can then call your function from within a `BEGIN PROGRAM-END PROGRAM` block in SPSS Statistics, as shown in the example that follows, or call it from within a Python IDE. To learn how to include a function in a Python module and make sure it can be found by Python, see "Creating User-Defined Functions in Python" on p. 203. To learn how to run code from a Python IDE, see "Running Your Code from a Python IDE" on p. 176.

Example

This example calls the Python function *SelectCases* with the same parameter values used in the call to the macro *SelectCases*.

```
*python_select_cases.sps.
BEGIN PROGRAM.
import samplelib
crit="(gender='m' AND jobcat=1 AND educ<16)"
samplelib.SelectCases(5,crit,
                 r'/examples/data/Employee data.sav',
                 r'/temp/results.sav')
END PROGRAM.
```

- Once you've created a user-defined function and saved it to a module file, you can call it from a `BEGIN PROGRAM` block that includes the statement to import the module. In this case, the `SelectCases` function is contained in the `samplelib` module, so the program block includes the `import samplelib` statement.

    Note: To run this program block, copy the module file *samplelib.py* from the */examples/python* folder, in the accompanying examples, to your Python *site-packages* directory. For help in locating your Python *site-packages* directory, see "Using This Book" on p. 1.

Runtime Behavior of Macros and Python Programs

Both macros and Python programs are defined when read, but when called, a macro is expanded before any of it is executed, while Python programs are evaluated line by line. This means that a Python program can respond to changes in the state of the SPSS Statistics dictionary that occur during the course of its execution, while a macro cannot.

Special Topics

Using Regular Expressions

Regular expressions define patterns of characters that are matched against a string to determine if the string contains the pattern. In addition to identifying matches, you can extract the part of a string matching the pattern, replace the matched part with a specified string, or split the string apart wherever the pattern matches, returning a list of the pieces. As implemented in the Python programming language, regular expressions provide a powerful tool for working with strings that greatly extends the built-in string operations supplied with the language.

   Constructing regular expressions in Python requires learning a highly specialized programming language embedded within the Python language. The example in this section uses a number of elements of this language and is meant to demonstrate the power of regular expressions rather than serve as a tutorial on them. A good introduction to regular expressions in the Python language can be found in the section "Regular expression operations" in the Python Library Reference, available at *http://docs.python.org/lib/module-re.html*.

Example

In this example, we'll use a regular expression to extract the two-character state code from an address that is provided as a single string. A table lookup is used to obtain the state name, which is then added as a new variable to the active dataset.

Figure 22-1
Dataset with addresses containing state codes
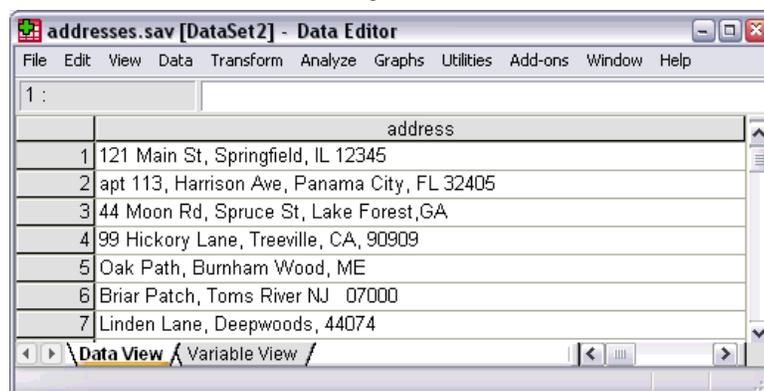
```
*python_re_state_lookup.sps.
BEGIN PROGRAM.
import spss, spssaux, spssdata, re
spssaux.OpenDataFile('/examples/data/addresses.sav')

statecodeRegexObj = re.compile(r"\b([A-Z]{2})\b,?\s*\d*\s*\Z")

stateCodes = {"IL":"Illinois", "NJ":"New Jersey","GA":"Georgia",
              "CA":"California","ME":"Maine"}
curs = spssdata.Spssdata(accessType='w')
curs.append(spssdata.vdef("stateName", vfmt=("A", 24)))
curs.commitdict()

for case in curs:
    try:
        matchObj=statecodeRegexObj.search(case.address.rstrip())
        code=matchObj.groups()[0]
        curs.casevalues([stateCodes[code]])
    except (AttributeError, KeyError):
        pass
curs.close()
END PROGRAM.
```

- This example makes use of the built-in Python module `re` for working with regular expressions, so the `import` statement includes it. The example also makes use of the `spssaux` and `spssdata` modules—supplementary modules installed with the IBM® SPSS® Statistics - Integration Plug-In for Python.

- The `OpenDataFile` function from the `spssaux` module opens an external IBM® SPSS® Statistics data file. The argument is the file path specified as a string. In this example, we use the *addresses.sav* dataset. It contains the single variable *address* from which state codes will be extracted.

- The regular expression for matching state codes is `\b([A-Z]{2})\b,?\s*\d*\s*\Z`. It is written to be as robust as possible to variations in the address field and will match a sequence of two uppercase letters set off by punctuation or white space, followed by an optional comma, optional white space, an optional string of digits, more optional white space, and the end of the string.

  Briefly, `[A-Z]{2}` matches two uppercase letters and `\b` matches the empty string at the beginning or end of a word, so `\b[A-Z]{2}\b` will match a word consisting of two uppercase letters. The parentheses enclosing `[A-Z]{2}` specify the start and end of a group. The contents of a group—in this case, the two-character state code—can be retrieved after a match has been performed.

  The sequence `,?\s*\d*\s*\Z` specifies the pattern of characters that must follow a two-letter word in order to provide a match. It specifies an optional comma (`,?`), optional white space (`\s*`), an optional string of digits (`\d*`), more optional white space (`\s*`), and the end of the string (`\Z`).

- The `compile` function from the `re` module compiles a regular expression. Compiling regular expressions is optional but increases the efficiency of matching when the expression is used several times in a single program. The argument is the regular expression as a string. The result of the `compile` function is a regular expression object, which in this example is stored to the Python variable *statecodeRegexObj*.

> *Note*: The `r` preceding the regular expression specifies a raw string, which ensures that any character sets specifying Python escape sequences—such as `\b`, which is the escape sequence for a backspace—are treated as raw characters and not the corresponding escape sequence.

- The variable *stateCodes* is a Python dictionary. A Python dictionary consists of a set of keys, each of which has an associated value that can be accessed simply by specifying the key. In this example, the keys are the state codes and the associated values are the full state names.

- The code `spssdata.Spssdata(accessType='w')` creates an instance of the `Spssdata` class (from the `spssdata` module), which allows you to add new variables to the active dataset. The instance in this example is stored to the Python variable *curs*.

- In this example, we'll add a string variable of width 24 bytes for the full state name. The specifications for the new variable *stateName* are created with the `append` method from the `Spssdata` class, and the variable is created with the `commitdict` method. For more information, see the topic "Using the spssdata Module" in Chapter 15 on p. 245.

- The `for` loop iterates through each of the cases in the active dataset. For each case, the Python variable *case* contains the values of the variables for that case. The Python code to extract the state code and obtain the associated state name generates an exception if no state code is found or the code doesn't exist in *stateCodes*. These two exception types are handled by the `try` and `except` statements. In the case of an exception, there is no action to take so the `except` clause simply contains the `pass` statement and processing continues to the next case.

- The `search` method of the compiled regular expression object scans a string for a match to the regular expression associated with the object. In this example, the string to scan is the value of the variable *address*, which is given by `case.address`. The string method `rstrip` is used to strip trailing blanks from the address. The result of the `search` method is a match object, which in this example is stored to the Python variable *matchObj*.

- The `groups` method of the match object returns a Python tuple containing the strings that match each of the groups defined in the regular expression. In this example, the regular expression contains a single group for the two-letter state code—that is, `([A-Z]{2})`—which is then stored to the Python variable *code*.

- The `casevalues` method of the `Spssdata` class is used to assign the values of new variables for the current case. The argument is a sequence of values, one for each new variable, in the order created. In this example, `casevalues` is used to assign the value of the variable *stateName* for the current case. The full state name is obtained by looking up the two-letter state code in the Python dictionary *stateCodes*. For example, `stateCodes['GA']` is `'Georgia'`.

For an example of using regular expressions to select a subset of variables in the active dataset, see "Using Regular Expressions to Select Variables" on p. 230. For examples of using regular expressions to search for and replace patterns of characters in case data, see "The search and subs Functions" on p. 309.

Locale Issues

For users who need to pay attention to locale issues, a few initial points are noteworthy.

- When used with IBM® SPSS® Statistics, the Python interpreter runs in the same locale as SPSS Statistics.

- Although the Python language provides the built-in module `locale` for dealing with locale issues, you should only change the locale with `SET LOCALE` command syntax. You may, however, want to use the `locale` module to retrieve information about the current locale.

Displaying Textual Output

In the Python language, the locale setting can affect how text is displayed in the output, including Python output displayed in the SPSS Statistics Viewer. In particular, the result of a Python `print` statement may include hex escape sequences when the expression to be printed is something other than a string, such as a list. For SPSS Statistics 16.0 and higher, the situation is further complicated because the SPSS Statistics processor can operate in code page mode (the default) or Unicode mode. This simple example illustrates the behavior and the general approach, for both Unicode mode and code page mode, for some accented characters.

```
BEGIN PROGRAM.
import spss, spssaux
from spssaux import u
spss.Submit("SET LOCALE='english'.")
list=[u("a"),u("ô"),u("é")]
print list
print " ".join(list)
END PROGRAM.
```

Result for Code Page Mode

```
['a', '\xf4', '\xe9']
a ô é
```

- Python string literals used in command syntax files, as done here, require special handling when working in Unicode mode. Specifically, they need to be explicitly expressed as UTF-16 strings. The `u` function from the `spssaux` module handles any necessary conversion for you, returning the appropriate value whether you are working in Unicode mode or code page mode. Unless your string literals only consist of plain Roman characters (7-bit ASCII), you should always use the `u` function for string literals in command syntax files. For more information, see the topic "Working in Unicode Mode" in Chapter 15 on p. 239.

  Note: Although Unicode mode is new in SPSS Statistics 16.0, the `u` function is compatible with earlier SPSS Statistics versions.

- The expression used for the first `print` statement is a list whose elements are strings. The accented characters in the strings are rendered as hex escape sequences in the output. When conversions to text are required, as with rendering a list in textual output, the Python interpreter produces output that is valid for use in Python syntax, and as this example shows, may not be what you expect.

- In the second `print` statement, the list is converted to a string using the Python string method `join`, which creates a string from a list by concatenating the elements of the list, using a specified string as the separator between elements. In this case, the separator is a single space. The `print` statement renders the resulting string as you would expect.

In general, if items render with hex escape sequences in output, convert those items to strings before including them on a `print` statement.

Regular Expressions

When working with regular expressions in the Python language, special sequences such as `\w` do not, by default, take into account characters specific to the current locale. For example, in French, the expression `\w` will match the alphanumeric characters `a-z`, `A-Z`, `0-9`, and the underscore (_), but not accented characters such as ô and é. You can use the `LOCALE` flag with the `compile`, `search`, `match`, and `findall` functions from the Python `re` module to specify that all alphanumeric characters specific to the current locale be included in matches. For example:

```
SET LOCALE = 'German'.
BEGIN PROGRAM.
import spss, re
s = "abcüô"
print(" ".join(re.findall("\w+", s)))
print(" ".join(re.findall("\w+", s, re.LOCALE)))
END PROGRAM.
```

■ The first `findall` returns `['abc']` while the second one gets all of the characters in the Python variable *s*.

Note: The `extendedTransforms` module, available from Developer Central, has a `subs` function that automatically applies the `re.LOCALE` flag.

Part III:
Programming with R

Introduction

The IBM® SPSS® Statistics - Integration Plug-In for R is one of a family of Integration Plug-Ins that also includes Python and .NET. It extends the IBM® SPSS® Statistics command syntax language with the full capabilities of the R programming language and is available on Windows, Linux, and Mac OS, as well as for SPSS Statistics Server. With this feature, R programs can access SPSS Statistics variable dictionary information, case data, and procedure output, as well as create new datasets and output in the form of pivot tables and R graphics.

Using this technology, you can write custom procedures in R that read the case data from the active dataset, apply algorithms written in R to the data, and write the results back as a new dataset or as pivot table output directed to the Viewer or exported via the Output Management System (OMS). You can analyze your data with an R function that you write or you can use a function from the extensive set of statistical routines available with R, all from within SPSS Statistics.

Prerequisites

The Integration Plug-In for R works with SPSS Statistics release 16.0 or later and only requires the Core system. The Plug-In, along with installation instructions, is available for download from Developer Central at *http://www.spss.com/devcentral/*. For release 18 on Windows, the Plug-In is installed with R Essentials. For release 19 and higher and for all platforms, the Plug-In is installed with IBM® SPSS® Statistics - Essentials for R.

The chapters that follow include hands-on examples of R programs and assume a basic working knowledge of the R programming language, although aspects of the language are discussed when deemed necessary. For help getting started with the R programming language, see "An Introduction to R," available at *http://cran.r-project.org/*.

Note: SPSS Inc. is not the owner or licensor of the R software. Any user of the R programming language must agree to the terms of the R license agreement located on the Web site for the R Project for Statistical Computing. SPSS Inc. is not making any statement about the quality of the R program. SPSS Inc. fully disclaims all liability associated with your use of the R program.

Additional Plug-Ins

The IBM® SPSS® Statistics Programmability Extension, included with the Core system, provides a general framework for supporting external languages through Integration Plug-Ins, such as the Integration Plug-In for R. In particular, there are also freeware Integration Plug-Ins for Python and .NET, available from Developer Central. The IBM® SPSS® Statistics - Integration Plug-In for Microsoft® .NET supports development in any .NET language and is intended for applications that interact with the SPSS Statistics back end but present their own user interface and provide their own means for displaying output. The IBM® SPSS® Statistics - Integration Plug-In for Python provides interfaces for extending the SPSS Statistics command syntax language with the full capabilities of the Python programming language and for operating on user interface and output objects. For more information, see Programming with Python on p. 171.

Getting Started with R Program Blocks

Once you've installed R and the IBM® SPSS® Statistics - Integration Plug-In for R, you have full access to all of the functionality of the R programming language and any installed R packages from within `BEGIN PROGRAM R-END PROGRAM` program blocks in command syntax. The basic structure is:

```
BEGIN PROGRAM R.
R statements
END PROGRAM.
```

Within an R program block, the R processor is in control, so all statements must be valid R statements. Even though program blocks are part of command syntax, you can't include syntax commands as statements in a program block. For example,

```
BEGIN PROGRAM R.
FREQUENCIES VARIABLES=var1, var2, var3.
END PROGRAM.
```

will generate an error because `FREQUENCIES` is not an R command. As an example of a valid R program block, here is the classic "Hello, world!":

```
BEGIN PROGRAM R.
cat("Hello, world!")
END PROGRAM.
```

The example uses the R `cat` function to write output to R's standard output, which is directed to a log item in the IBM® SPSS® Statistics Viewer if a Viewer is available. Likewise, the result of the R `print` function, which also writes to R's standard output, is directed to a log item.

Figure 24-1
Output from BEGIN PROGRAM R displayed in a log item

Displaying Output from R

For SPSS Statistics version 18 and higher, and by default, console output and graphics from R are redirected to the SPSS Statistics Viewer. This includes implicit output from R functions that would be generated when running those functions from within an R console—for example, the model coefficients and various statistics displayed by the `glm` function, or the mean value displayed by

the `mean` function. You can toggle the display of output from R with the `spsspkg.SetOutput` function.

Accessing R Help Within IBM SPSS Statistics

You can access help for R functions from within SPSS Statistics. Simply include a call to the R `help` function in a `BEGIN PROGRAM R-END PROGRAM` block and run the block. For example:

```
BEGIN PROGRAM R.
help(paste)
END PROGRAM.
```

to obtain help for the R `paste` function.

You can access R's main html help page with:

```
BEGIN PROGRAM R.
help.start()
END PROGRAM.
```

Debugging

For SPSS Statistics version 18 and higher, you can use the R `browser`, `debug`, and `undebug` functions within `BEGIN PROGRAM R-END PROGRAM` blocks, as well as from within implementation code for extension commands implemented in R. This allows you to use some of the same debugging tools available in an R console. Briefly, the `browser` function interrupts execution and displays a console window that allows you to inspect objects in the associated environment, such as variable values and expressions. The `debug` function is used to flag a specific R function—for instance, an R function that implements an extension command—for debugging. When the function is called, a console window is displayed and you can step through the function one statement at a time, inspecting variable values and expressions.

■ Results displayed in a console window associated with use of the `browser` or `debug` function are displayed in the SPSS Statistics Viewer after the completion of the program block or extension command containing the function call.

Note: When a call to a function that generates explicit output—such as the R `print` function—precedes a call to `browser` or `debug`, the resulting output is displayed in the SPSS Statistics Viewer after the completion of the program block or extension command containing the function call. You can cause such output to be displayed in the R console window associated with `browser` or `debug` by ensuring that the call to `browser` or `debug` precedes the function that generates the output and then stepping through the call to the output function.

■ Use of the `debug` and `browser` functions is not supported in distributed mode.

For more information on the use of the `debug` and `browser` functions, see the R help for those functions.

R Functions that Read from stdin

Some R functions take input data from an external file or from the standard input connection `stdin`. For example, by default, the `scan` function reads from `stdin` but can also read from an external file specified by the *file* argument. When working with R functions within BEGIN

PROGRAM R-END PROGRAM blocks, reading data from stdin is not supported, due to the fact that R is embedded within SPSS Statistics. For such functions, you will need to read data from an external file. For example:

```
BEGIN PROGRAM R.
data <- scan(file="/Rdata.txt")
END PROGRAM.
```

The spss R Package

The spss R package, installed with the Integration Plug-In for R, contains the SPSS Statistics-specific R functions that enable the process of using the R programming language from within SPSS Statistics command syntax. The package provides functions to:

- Read case data from the active dataset into R.
- Get information about data in the active dataset.
- Get output results from SPSS Statistics syntax commands.
- Write results from R back to SPSS Statistics.
- Display R graphics in the SPSS Statistics Viewer.

BEGIN PROGRAM R-END PROGRAM blocks do not need and, in general, should not contain an R library command for loading the spss R package. The correct version of the package is automatically loaded for you.

R Syntax Rules

R is case-sensitive. This includes variable names, function names, and pretty much anything else you can think of. A variable name of *myRvariable* is not the same as *MyRVariable*, and the function GetCaseCount() cannot be written as getcasecount().

R uses a less-than sign followed by a dash (<-) for assignment. For example:

```
var1 <- var2+1
```

Note: In most contexts, you can also use an equals sign (=) for assignment.

R commands are terminated with a semicolon or new line; continuation lines do not require special characters or indentation. For example,

```
var1 <- var2+
3
```

is read as var1<-var2+3, since R continues to read input until a command is syntactically complete. However,

```
var1 <- var2
+3
```

will be read as two separate commands, and *var1* will be set to the value of *var2*.

Groupings of statements are indicated by braces. Groups of statements in structures such as loops, conditional expressions, and functions are indicated by enclosing the statements in braces, as in:

```
while (!spssdata.IsLastSplit()){
   data <- spssdata.GetSplitDataFromSPSS()
   cat("\nCases in Split: ",length(data[,1]))
}
```

R Quoting Conventions

■ Strings in the R programming language can be enclosed in matching single quotes (`'`) or double quotes (`"`), as in IBM® SPSS® Statistics.

■ To specify an apostrophe (single quote) within a string, enclose the string in double quotes. For example,

```
"Joe's Bar and Grille"
```

is treated as

```
Joe's Bar and Grille
```

You can also use all single quotes, escaping the quote to be used as an apostrophe, as in
```
'Joe\'s Bar and Grille'
```

■ To specify quotation marks (double quote) within a string, use single quotes to enclose the string, as in

```
'Categories Labeled "UNSTANDARD" in the Report'
```

■ In the R programming language, doubled quotes of the same type as the outer quotes are not allowed. For example,

```
'Joe''s Bar and Grille'
```

results in an error.

File Specifications. Since escape sequences in the R programming language begin with a backslash (\)—such as `\n` for newline and `\t` for tab—it is recommended to use forward slashes (/) in file specifications on Windows. In this regard, SPSS Statistics always accepts a forward slash in file specifications.

```
spssRGraphics.Submit("/temp/R_graphic.jpg")
```

Alternatively, you can escape each backslash with another backslash, as in:

```
spssRGraphics.Submit("\\temp\\R_graphic.jpg")
```

Mixing Command Syntax and R Program Blocks

Within a given command syntax job, you can intersperse `BEGIN PROGRAM R-END PROGRAM` blocks with any other syntax commands, and you can have multiple R program blocks in a given job. R variables assigned in a particular program block are available to subsequent program blocks as shown in this simple example:

Example

```
*R_multiple_program_blocks.sps.
DATA LIST FREE /var1.
BEGIN DATA
1
END DATA.
DATASET NAME File1.
BEGIN PROGRAM R.
File1N <- spssdata.GetCaseCount()
END PROGRAM.
DATA LIST FREE /var1.
BEGIN DATA
1
2
END DATA.
DATASET NAME File2.
BEGIN PROGRAM R.
File2N <- spssdata.GetCaseCount()
{if (File2N > File1N)
  message <- "File2 has more cases than File1."
else if (File1N > File2N)
  message <- "File1 has more cases than File2."
else
  message <- "Both files have the same number of cases."
}
cat(message)
END PROGRAM.
```

- The first program block defines a programmatic variable, *File1N*, with a value set to the number of cases in the active dataset.

- The first program block is followed by command syntax that creates and names a new active dataset. Although you cannot execute IBM® SPSS® Statistics command syntax from within an R program block, you can have multiple R program blocks separated by command syntax that performs any necessary actions.

- The second program block defines a programmatic variable, *File2N*, with a value set to the number of cases in the SPSS Statistics dataset named *File2*. The value of *File1N* persists from the first program block, so the two case counts can be compared in the second program block.

- The R function `cat` is used to display the value of the R variable *message*. Output written to R's standard output—for instance, with the `cat` or `print` function—is directed to a log item in the SPSS Statistics Viewer.

Notes

- BEGIN PROGRAM R blocks cannot be nested. However, you can nest a BEGIN PROGRAM R block within a BEGIN PROGRAM PYTHON block. For more information, see the topic "Nested Program Blocks" in Chapter 12 on p. 189.

- To minimize R memory usage, you may want to delete large objects such as SPSS Statistics datasets at the end of your R program block—for example, rm(data).

Getting Help

Help with using the features of the IBM® SPSS® Statistics - Integration Plug-In for R is available from the following resources:

- Complete documentation for all of the functions available with the Integration Plug-In for R is available in the IBM® SPSS® Statistics Help system, under Integration Plug-In for R Help, along with simple examples of performing the common tasks of retrieving data and writing results back as pivot table output or new datasets. The documentation is also available in PDF from Help>Programmability>R Plug-in, within SPSS Statistics, once the Integration Plug-In for R is installed.

- Detailed command syntax reference information for BEGIN PROGRAM-END PROGRAM can be found in the SPSS Statistics Help system.

- Help for getting started with the R programming language can be found in "An Introduction to R," available at *http://cran.r-project.org/*.

- You can also post questions about using R with SPSS Statistics to the R Forum on Developer Central.

Retrieving Variable Dictionary Information

The IBM® SPSS® Statistics - Integration Plug-In for R provides a number of functions for retrieving dictionary information from the active dataset. It includes functions to retrieve:

- Variable names
- Variable labels
- Variable type (numeric or string)
- Display formats of variables
- Measurement levels of variables
- Names of any split variables
- Missing values
- Value labels
- Custom variable attributes
- Datafile attributes
- Multiple response sets
- Weight variable, if any

Basic information for each of the variables in the active dataset is available from the `spssdictionary.GetDictionaryFromSPSS` function, shown in the following example. Functions that retrieve a specific variable property, such as the variable label or the measurement level, are also available. See the topic on the Integration Plug-In for R in the IBM® SPSS® Statistics Help system for details.

Example

```
*R_GetDictionaryFromSPSS.sps.
DATA LIST FREE /id (F4) gender (A1) training (F1).
VARIABLE LABELS id 'Employee ID'
   /training 'Training Level'.
VARIABLE LEVEL id (SCALE)
   /gender (NOMINAL)
   /training (ORDINAL).
VALUE LABELS training 1 'Beginning' 2 'Intermediate' 3 'Advanced'
   /gender 'f' 'Female' 'm' 'Male'.
BEGIN DATA
18 m 1
37 f 2
10 f 3
END DATA.
BEGIN PROGRAM R.
vardict <- spssdictionary.GetDictionaryFromSPSS()
print(vardict)
END PROGRAM.
```

347

Result

```
                              X1        X2           X3
varName                       id    gender     training
varLabel            Employee ID            Training Level
varType                        0         1            0
varFormat                     F4        A1           F1
varMeasurementLevel        scale   nominal      ordinal
```

The result is an R data frame. Each column of the data frame contains the information for a single variable from the active dataset. The information for each variable consists of the variable name, the variable label, the variable type (0 for numeric variables and an integer equal to the defined length for string variables), the display format, and the measurement level.

Working with the Data Frame Representation of a Dictionary

The data frame returned by the GetDictionaryFromSPSS function contains the row labels *varName*, *varLabel*, *varType*, *varFormat*, and *varMeasurementLevel*. You can use these labels to specify the corresponding row. For example, the following code extracts the variable names:

```
varNames <- vardict["varName",]
```

It is often convenient to obtain separate lists of categorical and scale variables. The following code shows how to do this using the data frame representation of the SPSS Statistics dictionary. The results are stored in the two R vectors *scaleVars* and *catVars*.

```
scaleVars<-vardict["varName",][vardict["varMeasurementLevel",]=="scale"]
catVars<-vardict["varName",][vardict["varMeasurementLevel",]=="nominal" |
                            vardict["varMeasurementLevel",]=="ordinal"]
```

Retrieving Definitions of User-Missing Values

The spssdictionary.GetUserMissingValues function returns the user-missing values for a specified variable.

```
*R_user_missing_defs.sps.
data  list list (,)/v1 to v4(4f) v5(a4).
begin data.
0,0,0,0,a
end data.

missing values v2(0,9) v3(0 thru 1.5) v4 (0 thru 1.5, 9) v5(' ').

BEGIN PROGRAM R.
dict <- spssdictionary.GetDictionaryFromSPSS()
varnames <- dict["varName",]
for (name in varnames){
   vals <- spssdictionary.GetUserMissingValues(name)
   {if (is.nan(vals$missing[[1]]) | is.na(vals$missing[[1]])){
       res <- "no missing values"}
    else
       res <- vals
   }
   if (is.null(vals$type))
      vals$type <- "NULL"
   cat(name,":",vals$type,"\n")
   print(vals$missing)
}
END PROGRAM.
```

Result

```
v1 : Discrete
[1] NaN NaN NaN
v2 : Discrete
[1]   0   9 NaN
v3 : Range
[1] 0.0 1.5 NaN
v4 : Range Discrete
[1] 0.0 1.5 9.0
v5 : NULL
[1] "    " NA      NA
```

- The `GetDictionaryFromSPSS` function is used to obtain the variable names from the active dataset, which are stored to the R variable *varnames*.

- The `GetUserMissingValues` function returns a list containing any user-missing values for the specified variable. The argument specifies the variable and can be a character string consisting of the variable name (as shown here) or an integer specifying the index value of the variable (index values represent position in the dataset, starting with 0 for the first variable in file order). Variable names must match case with the names as they exist in the active dataset's dictionary.

- The list returned by `GetUserMissingValues` consists of the two named components *type* and *missing*. *type* is a character string specifying the missing value type: 'Discrete' for discrete numeric values, 'Range' for a range of values, 'Range Discrete' for a range of values and a single discrete value, and *NULL* for missing values of a string variable. The component *missing* is a vector containing the missing values.

- For variables with no missing values, the first element of *missing* is *NaN* for a numeric variable and *NA* for a string variable. Testing the first element of *missing* is then sufficient to determine the absence of missing values, as is the case for the IBM® SPSS® Statistics variable *v1*.

- For numeric variables with discrete missing values, the elements of *missing* specify the missing values. The result will contain one or more *NaN* values when there are less than three missing values, as for the variable *v2* in the current example.

- For variables with a range of missing values, the first and second elements of *missing* specify the lower and upper limits of the range respectively. In the current example, the range 0 to 1.5 is specified as missing for the variable *v3*.

- For variables with a range of missing values and a single discrete missing value, the first and second elements of *missing* specify the range and the third element specifies the discrete value. In the current example, the range 0 to 1.5 is specified as missing for the variable *v4*, along with the discrete value 9.

- For string variables, *type* is always *NULL* (converted to the string "NULL" in the displayed result). The vector *missing* will contain one or more *NA* values when there are less than three missing values, as for the variable *v5* in the current example. The returned values are right-padded to the defined width of the string variable. In the current example, the single missing value is a blank string, so the returned value is a string of width 4 consisting of blanks.

Identifying Variables without Value Labels

The spssdictionary.GetValueLabels function returns the value labels for a specified variable. The following example shows how to obtain a list of variables that do not have value labels.

```
*R_vars_no_value_labels.sps.
BEGIN PROGRAM R.
novallabelList <- vector()
dict <- spssdictionary.GetDictionaryFromSPSS()
varnames <- dict["varName",]
for (name in varnames){
    if (length(spssdictionary.GetValueLabels(name)$values)==0)
        novallabelList <- append(novallabelList,name)
}
{if (length(novallabelList) > 0) {
    cat("Variables without value labels:\n")
    cat(novallabelList,sep="\n")}
else
    cat("All variables have value labels")
}
END PROGRAM.
```

- The GetDictionaryFromSPSS function is used to obtain the variable names from the active dataset, which are stored to the R variable *varnames*.

- The GetValueLabels function returns a list containing any value labels for the specified variable. The argument specifies the variable and can be a character string consisting of the variable name (as shown here) or an integer specifying the index value of the variable (index values represent position in the dataset, starting with 0 for the first variable in file order). Variable names must match case with the names as they exist in the active dataset's dictionary.

- The list returned by GetValueLabels consists of the two named components *values* and *labels*. *values* is a vector of values that have associated labels and *labels* is a vector with the associated labels. If there are no value labels, the returned list contains empty vectors, so checking the length of either of the vectors is sufficient to determine the absence of value labels.

Identifying Variables with Custom Attributes

The `spssdictionary.GetVariableAttributeNames` and
`spssdictionary.GetVariableAttributes` functions allow you to retrieve information
about any custom variable attributes for the active dataset.

Example

A number of variables in the sample dataset *employee_data_attrs.sav* have a variable attribute
named `'DemographicVars'`. Create a list of these variables.

```
*R_var_attr.sps.
GET FILE='/examples/data/employee_data_attrs.sav'.
BEGIN PROGRAM R.
varList <- vector()
attribute <- "DemographicVars"
dict <- spssdictionary.GetDictionaryFromSPSS()
varnames <- dict["varName",]
for (name in varnames){
   if (any(attribute==spssdictionary.GetVariableAttributeNames(name)))
       varList <- c(varList,name)
}
{if (length(varList) > 0){
   cat(paste("Variables with attribute ",attribute,":\n"))
   cat(varList,sep="\n")}
else
   cat(paste("No variables have the attribute ",attribute))
}
END PROGRAM.
```

- The `GET` command is called outside of the R program block to get the desired file, since
  command syntax cannot be submitted from within an R program block.

- The `GetDictionaryFromSPSS` function is used to obtain the variable names from the active
  dataset, which are stored to the R variable *varnames*.

- The `GetVariableAttributeNames` function returns a vector containing the names of any
  variable attributes for the specified variable. The argument specifies the variable and can be
  a character string consisting of the variable name (as shown here) or an integer specifying
  the index value of the variable (index values represent position in the dataset, starting with
  0 for the first variable in file order). Variable names must match case with the names as
  they exist in the active dataset's dictionary.

Retrieving Datafile Attributes

The `spssdictionary.GetDataFileAttributeNames` and
`spssdictionary.GetDataFileAttributes` functions allow you to retrieve information
about any datafile attributes for the active dataset.

Example

The sample dataset *employee_data_attrs.sav* has a number of datafile attributes. Determine if the
dataset has a datafile attribute named `'LastRevised'`. If the attribute exists, retrieve its value.

```
*R_file_attr.sps.
GET FILE='/examples/data/employee_data_attrs.sav'.
BEGIN PROGRAM R.
names <- spssdictionary.GetDataFileAttributeNames()
for(attr in names)
   if (attr == 'LastRevised')
      cat("Dataset last revised on:",
            spssdictionary.GetDataFileAttributes(attrName = attr))
END PROGRAM.
```

- The GET command is called outside of the R program block to get the desired file, since command syntax cannot be submitted from within an R program block.

- The GetDataFileAttributeNames function returns a vector consisting of the names of any datafile attributes for the active dataset.

- The GetDataFileAttributes function returns a vector of the values for the specified attribute (datafile attributes can consist of an array of values). The argument *attrName* is a string that specifies the name of the attribute—for instance, a name returned by GetDataFileAttributeNames, as in this example.

Retrieving Multiple Response Sets

The spssdictionary.GetMultiResponseSetNames and spssdictionary.GetMultiResponseSet functions allow you to retrieve information about any multiple response sets for the active dataset.

Example

The sample dataset *telco_extra_mrsets.sav* has a number of multiple response sets. Display the elementary variables associated with each set.

```
*R_mrset.sps.
GET FILE='/examples/data/telco_extra_mrsets.sav'.
BEGIN PROGRAM R.
names <- spssdictionary.GetMultiResponseSetNames()
for (set in names)
{
    mrset <- spssdictionary.GetMultiResponseSet(mrsetName = set)
    cat("\nElementary variables for:",set,"\n")
    cat(mrset$vars,sep="\n")
}
END PROGRAM.
```

- The GET command is called outside of the R program block to get the desired file, since command syntax cannot be submitted from within an R program block.

- The GetMultiResponseSetNames function returns a vector of names of the multiple response sets, if any, for the active dataset.

- The GetMultiResponseSet function returns the details of the specified multiple response set. The argument *mrsetName* is a string that specifies the name of the multiple response set—for instance, a name returned by GetMultiResponseSetNames, as in this example. The result is a list with the following named components: *label* (the label, if any, for the set), *codeAs* ("Dichotomies" or "Categories"), *countedValue* (the counted value—applies only to

multiple dichotomy sets), *type* ("Numeric" or "String"), and *vars* (a vector of the elementary variables that define the set).

Reading Case Data from IBM SPSS Statistics

The IBM® SPSS® Statistics - Integration Plug-In for R provides the ability to read case data from IBM® SPSS® Statistics into R. You can choose to retrieve the cases for all variables or a selected subset of the variables in the active dataset.

Using the spssdata.GetDataFromSPSS Function

The `spssdata.GetDataFromSPSS` function is used to read case data and is intended for use with datasets that do not have split groups. If you need to read from a dataset with splits, use the `spssdata.GetSplitDataFromSPSS` function (see "Handling Data with Splits" on p. 358). When retrieving case data from IBM® SPSS® Statistics the following rules apply, whether you use `GetDataFromSPSS` or `GetSplitDataFromSPSS`:

- String values are right-padded to the defined width of the string variable.
- Values retrieved from SPSS Statistics variables with time formats are returned as integers representing the number of seconds from midnight.

When reading categorical data, note that the analogue of a categorical variable in SPSS Statistics is a factor in R. See "Working with Categorical Variables" for details. For the handling of missing values, see "Missing Data" on p. 357.

Example: Retrieving Cases for All Variables

```
*R_get_all_cases.sps.
DATA LIST FREE /age (F4) income (F8.2) car (F8.2) employ (F4).
BEGIN DATA.
55  72  36.20  23
56  153 76.90  35
28  28  13.70  4
END DATA.
BEGIN PROGRAM R.
casedata <- spssdata.GetDataFromSPSS()
print(casedata)
END PROGRAM.
```

Result

```
  age income  car employ
1  55     72 36.2     23
2  56    153 76.9     35
3  28     28 13.7      4
```

The result from the `GetDataFromSPSS` function is an R data frame. In that regard, unlike SPSS Statistics, the data retrieved by `GetDataFromSPSS` (or `GetSplitDataFromSPSS`) are held in memory. Each column of the returned data frame contains the case data for a single variable from the active dataset (for string variables, the columns are factors). The column name is the

variable name (in the same case as stored in the SPSS Statistics dictionary) and can be used to extract the data for that variable, as in:

```
income <- casedata$income
```

The R variable *income* is a vector containing the case values of the SPSS Statistics variable of the same name. You can iterate over these case values, as in:

```
for (value in income) print(value)
```

Each row of the returned data frame contains the data for a single case. By default, the rows are labeled with consecutive integers. You can iterate over the rows of the data frame, effectively iterating over the cases, as in:

```
for (i in c(1:spssdata.GetCaseCount())) print(casedata[i,]$age)
```

■ The `spssdata.GetCaseCount` function returns the number of cases in the active dataset. The R `c()` function is then used to create a vector of row labels—in this case, just the integers from 1 to the number of cases.

■ On each iteration of the loop, `casedata[i,]` is a list with a named component for each of the variables in the data frame, so `casedata[i,]$age` is the value of *age* for the current case.

Note: When calling `GetDataFromSPSS`, you can include the optional argument *row.label* to specify a variable from the active dataset whose case values will be the row labels of the resulting data frame. The specified variable must not contain duplicate values.

Example: Retrieving Cases for Selected Variables

```
*R_get_specified_variables.sps.
DATA LIST FREE /age (F4) income (F8.2) car (F8.2) employ (F4).
BEGIN DATA.
55   72  36.20   23
56   153 76.90   35
28   28  13.70   4
END DATA.
BEGIN PROGRAM R.
casedata <- spssdata.GetDataFromSPSS(variables=c("age","income","employ"))
END PROGRAM.
```

The argument *variables* to the `GetDataFromSPSS` function is an R vector specifying a subset of variables for which case data will be retrieved. In this example, the R function `c()` is used to create a character vector of variable names. The names must be specified in the same case as the associated variables in the dictionary for the active dataset. The resulting R data frame (*casedata*) will contain the three columns labeled *age*, *income*, and *employ*.

You can use the TO keyword to specify a range of variables as you can in SPSS Statistics—for example, `variables=c("age TO car")`. Unlike the variable names, the case of the TO keyword does not matter. If you prefer to work with variable index values (index values represent position in the dataset, starting with 0 for the first variable in file order), you can specify a range of variables with an expression such as `variables=c(0:2)`. The R code `c(0:2)` creates a vector consisting of the integers between 0 and 2 inclusive.

Missing Data

By default, user-missing values for numeric variables are converted to the R *NaN* value, and user-missing values of string variables are converted to the R *NA* value. System-missing values are always converted to the R *NaN* value. In the following example, we create a dataset that includes both system-missing and user-missing values.

```
*R_get_missing_data.sps.
DATA LIST LIST (',') /numVar (f) stringVar (a4).
BEGIN DATA
1,a
,b
3,,
9,d
END DATA.
MISSING VALUES numVar (9) stringVar (' ').
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS()
cat("Case data with missing values:\n")
print(data)
END PROGRAM.
```

Result

```
Case data with missing values:

  numVar stringVar
1      1         a
2    NaN         b
3      3      <NA>
4    NaN         d
```

Note: You can specify that missing values of numeric variables be converted to the R *NA* value, with the *missingValueToNA* argument, as in:

```
data<-spssdata.GetDataFromSPSS(missingValueToNA=TRUE)
```

You can specify that user-missing values be treated as valid data by setting the optional argument *keepUserMissing* to *TRUE*, as shown in the following reworking of the previous example.

```
DATA LIST LIST (',') /numVar (f) stringVar (a4).
BEGIN DATA
1,a
,b
3,,
9,d
END DATA.
MISSING VALUES numVar (9) stringVar (' ').
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS(keepUserMissing=TRUE)
cat("Case data with user-missing values treated as valid:\n")
print(data)
END PROGRAM.
```

Result

```
Case data with user-missing values treated as valid:

  numVar stringVar
1      1         a
2    NaN         b
3      3
4      9         d
```

Handling IBM SPSS Statistics Datetime Values

When retrieving values of IBM® SPSS® Statistics variables with date or datetime formats, you'll most likely want to convert the values to R date/time (POSIXt) objects. By default, such variables are not converted and are simply returned in the internal representation used by SPSS Statistics (floating point numbers representing some number of seconds and fractional seconds from an initial date and time). To convert variables with date or datetime formats to R date/time objects, you use the *rDate* argument of the `GetDataFromSPSS` function.

```
*R_retrieve_datetime_values.sps.
DATA LIST FREE /bdate (ADATE10).
BEGIN DATA
05/02/2009
END DATA.
BEGIN PROGRAM R.
data<-spssdata.GetDataFromSPSS(rDate="POSIXct")
data
END PROGRAM.
```

Result

```
        bdate
1 2009-05-02
```

Handling Data with Splits

When reading from IBM® SPSS® Statistics datasets with split groups, use the `spssdata.GetSplitDataFromSPSS` function to retrieve each split separately. The first call to `GetSplitDataFromSPSS` returns the data for the first split group, the second call returns the data for the second split group, and so on. In the case that the active dataset has no split groups, `GetSplitDataFromSPSS` returns all cases on its first call.

```
*R_get_split_groups.sps.
DATA LIST FREE /salary (F6) jobcat (F2).
BEGIN DATA
21450 1
45000 1
30000 2
30750 2
103750 3
72500 3
57000 3
END DATA.

SORT CASES BY jobcat.
SPLIT FILE BY jobcat.
BEGIN PROGRAM R.
varnames <- spssdata.GetSplitVariableNames()
if(length(varnames) > 0)
{
   while (!spssdata.IsLastSplit()){
      data <- spssdata.GetSplitDataFromSPSS()
      cat("\n\nSplit variable values:")
      for (name in varnames) cat("\n",name,":",
                                 as.character(data[1,name]))
      cat("\nCases in Split: ",length(data[,1]))
   }
   spssdata.CloseDataConnection()
}
END PROGRAM.
```

Result

```
Split variable values:
 jobcat : 1
Cases in Split:  2

Split variable values:
 jobcat : 2
Cases in Split:  2

Split variable values:
 jobcat : 3
Cases in Split:  3
```

- The GetSplitVariableNames function returns the names of the split variables, if any, from the active dataset.

- The IsLastSplit function returns *TRUE* if the current split group is the last one in the active dataset.

- The GetSplitDataFromSPSS function retrieves the case data for the next split group from the active dataset and returns it as an R data frame of the same form as that returned by GetDataFromSPSS. GetSplitDataFromSPSS returns *NULL* if there are no more split groups in the active dataset.

- The CloseDataConnection function should be called when the necessary split groups have been read. In particular, GetSplitDataFromSPSS implicitly starts a data connection for reading from split files and this data connection must be closed with CloseDataConnection.

As with the `GetDataFromSPSS` function, you can include the *variables* argument in `GetSplitDataFromSPSS` to specify a subset of variables to retrieve, the *row.label* argument to specify the row labels of the returned data frame, and the *keepUserMissing* argument to specify how to handle user-missing values.

Working with Categorical Variables

If you're reading categorical data into R and need to retain the categorical nature for your R analysis, you'll want to convert the retrieved data to R factor variables.

```
*R_handle_catvars.sps.
DATA LIST FREE /id (F4) gender (A1) training (F1).
VARIABLE LABELS id 'Employee ID'
   /training 'Training Level'.
VARIABLE LEVEL id (SCALE)
   /gender (NOMINAL)
   /training (ORDINAL).
VALUE LABELS training 1 'Beginning' 2 'Intermediate' 3 'Advanced'
   /gender 'f' 'Female' 'm' 'Male'.
BEGIN DATA
18 m 1
37 f 2
10 f 3
22 m 2
END DATA.
BEGIN PROGRAM R.
casedata <- spssdata.GetDataFromSPSS(factorMode="labels")
casedata
END PROGRAM.
```

- The *factorMode* argument of the `GetDataFromSPSS` function converts categorical variables from IBM® SPSS® Statistics to R factors.

- The value `"labels"` for *factorMode*, used in this example, specifies that categorical variables are converted to factors whose levels are the value labels of the variables. Values for which value labels do not exist have a level equal to the value itself. The alternate value `"levels"` specifies that categorical variables are converted to factors whose levels are the values of the variables.

Result

```
  id gender      training
1 18   Male     Beginning
2 37 Female  Intermediate
3 10 Female      Advanced
4 22   Male  Intermediate
```

If you intend to write factors retrieved with `factorMode="labels"` to a new SPSS Statistics dataset, special handling is required. For details, see the example on "Writing Categorical Variables Back to SPSS Statistics" in "Writing Results to a New IBM SPSS Statistics Dataset".

Writing Results to a New IBM SPSS Statistics Dataset

The IBM® SPSS® Statistics - Integration Plug-In for R provides the ability to write results from R to a new IBM® SPSS® Statistics dataset. You can create a dataset from scratch, explicitly specifying the variables and case values, or you can build on a copy of an existing dataset, adding new variables or cases.

Creating a New Dataset

The steps to create a new dataset are:

▶ Create the dictionary for the new dataset. Dictionaries for IBM® SPSS® Statistics datasets are represented by a data frame. You can create the data frame representation of the dictionary from scratch using the `spssdictionary.CreateSPSSDictionary` function or you can build on the dictionary of an existing dataset using the `spssdictionary.GetDictionaryFromSPSS` function. The new dataset based on the specified dictionary is created with the `spssdictionary.SetDictionaryToSPSS` function.

▶ Populate the case data using the `spssdata.SetDataToSPSS` function.

Note: When setting values for a SPSS Statistics variable with a date or datetime format, specify the values as R POSIXt objects, which will then be correctly converted to the values appropriate for SPSS Statistics.

Example: Adding Variables to a Copy of the Active Dataset

This example shows how to create a new dataset that is a copy of the active dataset with the addition of a single new variable. Specifically, it adds the mean salary to a copy of *Employee data.sav*.

```
*R_copy_dataset_add_var.sps.
GET FILE='/examples/data/Employee data.sav'.
BEGIN PROGRAM R.
dict <- spssdictionary.GetDictionaryFromSPSS()
casedata <- spssdata.GetDataFromSPSS()
varSpec <- c("meansal","Mean Salary",0,"F8","scale")
dict <- data.frame(dict,varSpec)
spssdictionary.SetDictionaryToSPSS("results",dict)
casedata <- data.frame(casedata,mean(casedata$salary))
spssdata.SetDataToSPSS("results",casedata)
spssdictionary.EndDataStep()
END PROGRAM.
```

■ The `GetDictionaryFromSPSS` function returns an R data frame representation of the dictionary for the active dataset. The `GetDataFromSPSS` function returns an R data frame representation of the case data from the active dataset.

■ To create a variable for a new dataset, you first specify the basic properties of the variable in an R vector, as in *varSpec* in this example. All of the following components of the vector are required and must appear in the specified order: variable name, variable label (can be a blank string), variable type (0 for numeric, and an integer equal to the defined length, with a maximum of 32,767, for a string variable), the display format of the variable, and measurement level (`"nominal"`, `"ordinal"`, or `"scale"`). For information on display formats, see the help for the `spssdictionary.CreateSPSSDictionary` function, available in the SPSS Statistics Help system.

■ The code `data.frame(dict,varSpec)` uses the R `data.frame` function to create a data frame representation of the new dictionary, consisting of the original dictionary and the new variable. To add more than one variable, simply add the specification vectors for the variables to the `data.frame` function, as in `data.frame(dict,varSpec1,varSpec2)`. The position of the variables in the new dataset is given by their order in the data frame. In the current example, the new dataset consists of the original variables from *Employee data.sav*, followed by *meansal*.

■ The `SetDictionaryToSPSS` function creates the new dataset. The arguments to `SetDictionaryToSPSS` are the name of the new dataset and a data frame representation of the dictionary.

■ The code `data.frame(casedata,mean(casedata$salary))` creates a new data frame consisting of the data retrieved from the active dataset and the data for the new variable. In this example, the new variable is the mean of the variable *salary* from the active dataset. You can build data frames from existing ones, as done here, or from vectors representing each of the columns. For example, `data.frame(var1,var2,var2)` creates a data frame whose columns are specified by the vectors *var1*, *var2*, and *var3*, which must be of equal length. For data frame representations of case data, the order of the columns in the data frame should match the order of the associated variables in the new dataset. In particular, the data in the first column of the data frame populates the case data for the first variable in the dataset, data in the second column of the data frame populates the case data for the second variable in the dataset, and so on.

■ The `SetDataToSPSS` function populates the case data of the new dataset. Its arguments are the name of the dataset to populate and a data frame representation of the case data, where the rows of the data frame are the cases and the columns represent the variables in file order.

■ The `EndDataStep` function should be called after completing the steps for creating the new dataset. If an error causes the `EndDataStep` function not to be executed, you can run another `BEGIN PROGRAM R-END PROGRAM` block that calls only the `EndDataStep` function.

Example: Creating a Variable Dictionary from Scratch

This example shows how to create the variable dictionary for a new dataset without use of an existing dictionary.

```
*R_create_dataset.sps.
GET FILE='/examples/data/Employee data.sav'.
BEGIN PROGRAM R.
casedata <- spssdata.GetDataFromSPSS()
stats <- summary(casedata$salary)
min <- c("min","",0,"F8.2","scale")
q1 <- c("q1","1st Quartile",0,"F8.2","scale")
median <- c("median","",0,"F8.2","scale")
```

```
mean <- c("mean","",0,"F8.2","scale")
q3 <- c("q3","3rd Quartile",0,"F8.2","scale")
max <- c("max","",0,"F8.2","scale")
dict <- spssdictionary.CreateSPSSDictionary(min,q1,median,mean,q3,max)
spssdictionary.SetDictionaryToSPSS("summary",dict)
data <- data.frame(min=stats[1],q1=stats[2],median=stats[3],
                   mean=stats[4],q3=stats[5],max=stats[6])
spssdata.SetDataToSPSS("summary",data)
spssdictionary.EndDataStep()
END PROGRAM.
```

- The example uses data from *Employee data.sav*, so the GET command is called (prior to the R program block) to get the file. The case data are read into R and stored in the variable *casedata*.

- The R summary function is used to create summary statistics for the variable *salary* from *Employee data.sav*. casedata$salary is an R vector containing the case values of *salary*.

- Specifications for six variables are stored in the R vector variables *min*, *q1*, *median*, *mean*, *q3*, and *max*.

- The CreateSPSSDictionary function is used to create a data frame representation of a dictionary based on the six variable specifications. The order of the arguments to CreateSPSSDictionary is the file order of the associated variables in the new dataset.

- The SetDictionaryToSPSS function creates a new dataset named *summary*, based on the dictionary from CreateSPSSDictionary. The arguments to SetDictionaryToSPSS are the name of the new dataset and a data frame representation of the dictionary.

- The R data.frame function is used to create a data frame representation of the case data for the new dataset, which consists of a single case with the summary statistics. The labels (*min*, *q1*, etc.) used in the arguments to the data.frame function are optional and have no effect on the case data passed to SPSS Statistics. The SetDataToSPSS function is then used to populate the case data of the new dataset.

Example: Adding Cases to a Copy of the Active Dataset

This example shows how to create a new dataset that is a copy of the active dataset but with additional cases.

```
*R_copy_dataset_add_cases.sps.
DATA LIST FREE /numvar (F) strvar (A2).
BEGIN DATA
1 a
END DATA.
BEGIN PROGRAM R.
dict <- spssdictionary.GetDictionaryFromSPSS()
casedata <- spssdata.GetDataFromSPSS()
numvar <- c(casedata$numvar,2,3)
strvar <- c(as.vector(casedata$strvar),format("b",width=2),format("c",width=2))
spssdictionary.SetDictionaryToSPSS("results",dict)
casedata <- data.frame(numvar,strvar)
spssdata.SetDataToSPSS("results",casedata)
spssdictionary.EndDataStep()
END PROGRAM.
```

- To create a copy of the dataset, but with additional cases, you create vectors containing the desired case data for each of the variables and use those vectors to create a data frame representing the case data. In this example, the R variables *numvar* and *strvar* contain the case data from the active dataset with the addition of the values for two new cases. Case values of

string variables from SPSS Statistics are returned as a factor, so the R `as.vector` function is used to convert the factor to a vector in order to append values.

■ The vectors are used to populate a data frame representing the case data. The order of the vectors in the `data.frame` function is the same as the order in which their associated variables appear in the dataset.

Example: Writing Categorical Variables Back to IBM SPSS Statistics

When reading categorical variables from SPSS Statistics with `factorMode="labels"` and writing the associated R factors to a new SPSS Statistics dataset, special handling is required because labeled factors in R do not preserve the original values. In this example, we read data containing categorical variables from SPSS Statistics and create a new dataset containing the original data with the addition of a single new variable.

```
*R_read_write_catvars.sps.
DATA LIST FREE /id (F4) gender (A1) training (F1) salary (DOLLAR).
VARIABLE LABELS id 'Employee ID'
/training 'Training Level'.
VARIABLE LEVEL id (SCALE)
/gender (NOMINAL)
/training (ORDINAL)
/salary (SCALE).
VALUE LABELS training 1 'Beginning' 2 'Intermediate' 3 'Advanced'
/gender 'm' 'Male' 'f' 'Female'.
BEGIN DATA
18 m 3 57000
37 f 2 30750
10 f 1 22000
22 m 2 31950
END DATA.

BEGIN PROGRAM  R.
dict <- spssdictionary.GetDictionaryFromSPSS()
casedata <- spssdata.GetDataFromSPSS(factorMode="labels")
catdict <- spssdictionary.GetCategoricalDictionaryFromSPSS()
varSpec <- c("meansal","Mean Salary",0,"DOLLAR8","scale")
dict<-data.frame(dict,varSpec)
casedata<-data.frame(casedata,mean(casedata$salary))
spssdictionary.SetDictionaryToSPSS("results",dict,categoryDictionary=catdict)
spssdata.SetDataToSPSS("results",casedata,categoryDictionary=catdict)
spssdictionary.EndDataStep()
END PROGRAM.
```

■ The `GetCategoricalDictionaryFromSPSS` function returns a structure (referred to as a **category dictionary**) containing the values and value labels of the categorical variables from the active dataset.

■ The category dictionary stored in *catdict* is used when creating the new dataset with the `SetDictionaryToSPSS` function and when writing the data to the new dataset with the `SetDataToSPSS` function. The value labels of the categorical variables are automatically added to the new dataset and the case values of those variables (in the new dataset) are the values from the original dataset.

Note: If you rename categorical variables when writing them back to SPSS Statistics, you must use the `spssdictionary.EditCategoricalDictionary` function to change the name in the associated category dictionary.

Saving New Datasets

To save a new dataset created from within an R program block, you include command syntax—such as SAVE or SAVE TRANSLATE—following the program block that created the dataset. A dataset created in an R program block is not, however, set as the active dataset. To make a new dataset the active one, use the spssdictionary.SetActive function from within the program block, as shown in this example, or the DATASET ACTIVATE command outside of the program block.

```
*R_save_dataset.sps.
BEGIN PROGRAM R.
var1Spec <- c("id","",0,"F2","scale")
var2Spec <- c("qty","",0,"F2","scale")
var1 <- c(13,21,43)
var2 <- c(25,57,42)
dict <- spssdictionary.CreateSPSSDictionary(var1Spec,var2Spec)
spssdictionary.SetDictionaryToSPSS("newds",dict)
casedata <- data.frame(var1,var2)
spssdata.SetDataToSPSS("newds",casedata)
spssdictionary.SetActive("newds")
spssdictionary.EndDataStep()
END PROGRAM.

SAVE OUTFILE='/temp/file1.sav'.
```

■ A new dataset named *newds* is created. After populating the case data with the SetDataToSPSS function, the SetActive function is called to make it the active dataset.

■ The new dataset is saved to the file system with a SAVE command that follows the R program block.

Specifying Missing Values for New Datasets

User-missing values for new variables are specified with the spssdictionary.SetUserMissing function. The function must be called after spssdictionary.SetDictionaryToSPSS and before calling spssdictionary.EndDataStep.

```
*R_specify_missing_values.sps.
BEGIN PROGRAM R.
var1Spec <- c("var1","",0,"F8.2","scale")
var2Spec <- c("var2","",0,"F8.2","scale")
var3Spec <- c("var3","",0,"F8.2","scale")
var4Spec <- c("var4","",2,"A2","nominal")
dict <- spssdictionary.CreateSPSSDictionary(var1Spec,var2Spec,var3Spec,
                                            var4Spec)
spssdictionary.SetDictionaryToSPSS("newds",dict)
spssdictionary.SetUserMissing("newds","var1",missingFormat["Discrete"],
                    c(0,9,99))
spssdictionary.SetUserMissing("newds","var2",missingFormat["Range"],
                    c(9,99))
spssdictionary.SetUserMissing("newds","var3",missingFormat["Range Discrete"],
                    c(9,99,0))
spssdictionary.SetUserMissing("newds","var4",missingFormat["Discrete"],
                    c(" ","NA"))
spssdictionary.EndDataStep()
END PROGRAM.
```

- The vectors *var1Spec*, *var2Spec*, *var3Spec*, and *var4Spec* provide the specifications for four new variables, the first three of which are numeric and the last of which is a string. A new dataset named *newds* consisting of these four variables is created.

- The `SetUserMissing` function is called after `SetDictionaryToSPSS` to specify the missing values. The first argument to the function is the dataset name and the second argument is the name of the variable whose missing values are being set.

- The third argument to `SetUserMissing` specifies the missing value type: `missingFormat["Discrete"]` for discrete missing values of numeric or string variables, `missingFormat["Range"]` for a range of missing values, and `missingFormat["Range Discrete"]` for a range of missing values and a single discrete value. String variables can have only discrete missing values.

- The fourth argument to `SetUserMissing` is a vector specifying the missing values. For discrete missing values, the vector can contain up to three values, as shown for *var1* and *var4*. For a range of missing values, as for *var2*, provide a vector whose first element is the start of the range and whose second element is the end of the range. For a range of missing values and a single discrete value, as for *var3*, the first two elements of the vector specify the range and the third element specifies the discrete value.

Note: Missing values for string variables cannot exceed eight bytes. Although string variables can have a defined width of up to 32,767 bytes, defined missing values cannot exceed eight bytes.

Specifying Value Labels for New Datasets

Value labels are set with the `spssdictionary.SetValueLabel` function. The function must be called after `spssdictionary.SetDictionaryToSPSS` and before calling `spssdictionary.EndDataStep`.

```
*R_specify_value_labels.sps.
BEGIN PROGRAM R.
var1Spec <- c("var1","Minority",0,"F1","ordinal")
var2Spec <- c("var2","Gender",1,"A1","nominal")
dict <- spssdictionary.CreateSPSSDictionary(var1Spec,var2Spec)
spssdictionary.SetDictionaryToSPSS("newds",dict)
spssdictionary.SetValueLabel("newds","var1",c(0,1),c("No","Yes"))
spssdictionary.SetValueLabel("newds","var2",c("m","f"),c("male","female"))
spssdictionary.EndDataStep()
END PROGRAM.
```

- The vectors *var1Spec* and *var2Spec* provide the specifications for a new numeric variable and a new string variable. A new dataset named *newds* consisting of these two variables is created.

- The `SetValueLabel` function is called after `SetDictionaryToSPSS` to specify the value labels. The first argument to the function is the dataset name, and the second argument is the name of the variable whose value labels are being set. The third argument to `SetValueLabel` is a vector specifying the values that have associated labels, and the fourth argument is a vector with the associated labels.

Specifying Variable Attributes for New Datasets

Variable attributes are specified with the `spssdictionary.SetVariableAttributes` function. The function must be called after `spssdictionary.SetDictionaryToSPSS` and before calling `spssdictionary.EndDataStep`.

```
*R_specify_var_attr.sps.
BEGIN PROGRAM R.
var1Spec <- c("var1","Minority",0,"F1","ordinal")
dict <- spssdictionary.CreateSPSSDictionary(var1Spec)
spssdictionary.SetDictionaryToSPSS("newds",dict)
spssdictionary.SetVariableAttributes("newds","var1",DemographicVars="1",
                                     Binary="Yes")
spssdictionary.EndDataStep()
END PROGRAM.
```

- The vector *var1Spec* provides the specifications for a new numeric variable. A new dataset named *newds* and consisting of this variable is created.

- The `SetVariableAttributes` function is called after `SetDictionaryToSPSS` to specify the variable attributes. The first argument to the function is the dataset name, and the second argument is the name of the variable whose attributes are being specified. The remaining arguments specify the attributes and are of the form `attrName=attrValue`, where `attrName` is the name of the attribute and `attrValue` is either a single character value or a character vector. Specifying a vector results in an attribute array. An arbitrary number of attribute arguments can be specified.

Creating Pivot Table Output

The IBM® SPSS® Statistics - Integration Plug-In for R provides the ability to render tabular output from R as a pivot table that can be displayed in the IBM® SPSS® Statistics Viewer or written to an external file using the SPSS Statistics Output Management System (OMS). Although you can use the R `print` or `cat` functions to send output to a log item in the SPSS Statistics Viewer, rendering tabular output as a pivot table provides much nicer looking output.

Pivot tables are created with the `BasePivotTable` class or the `spsspivottable.Display` function. The `BasePivotTable` class allows you to create pivot tables with an arbitrary number of row, column, and layer dimensions, whereas the `spsspivottable.Display` function is limited to creating pivot tables with a single row dimension and a single column dimension. If you only need a pivot table with one row dimension and one column dimension then consider using the `spsspivottable.Display` function.

Using the spsspivottable.Display Function

Example

```
*R_ptable_demo.sps.
BEGIN PROGRAM R.
demo <- data.frame(A=c("1A","2A"),B=c("1B","2B"),row.names=c(1,2))
spsspkg.StartProcedure("Sample Procedure")
spsspivottable.Display(demo,
                       title="Sample Pivot Table",
                       rowdim="Row",
                       hiderowdimtitle=FALSE,
                       coldim="Column",
                       hidecoldimtitle=FALSE)
spsspkg.EndProcedure()
END PROGRAM.
```

Result

Figure 28-1
Viewer output of sample pivot table
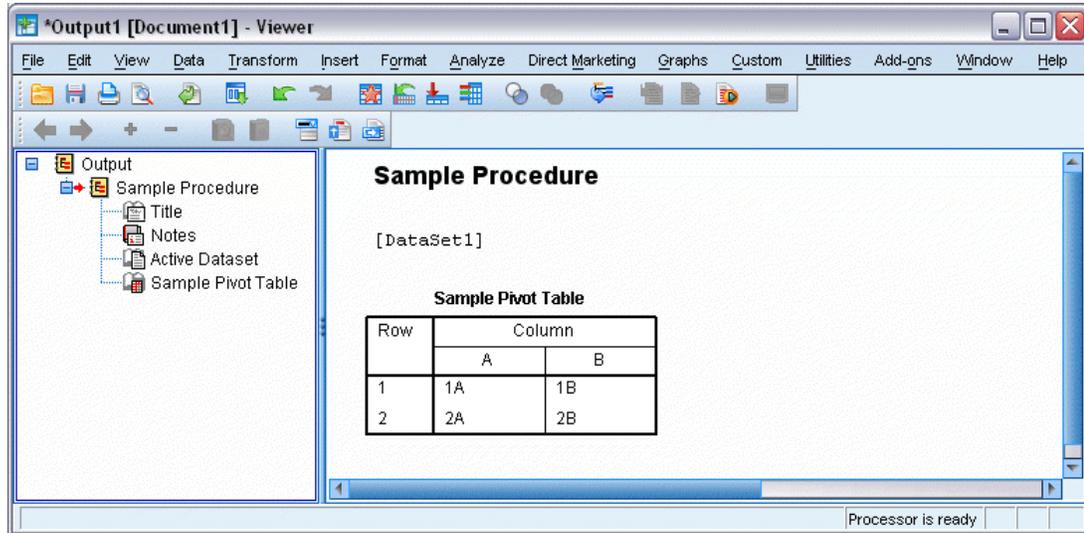
- The contents of a pivot table can be represented by an R data frame or any R object that can be converted to a data frame such as a matrix. In this example, the contents are provided as a data frame created with the R `data.frame` function. The rows of the data frame represent the rows of the pivot table, and the columns of the data frame—specified by the component vectors *A* and *B* in this example—represent the columns of the pivot table.

- By default, the row and column names of the provided data frame (or equivalent R object) are used as the row and column names for the resulting pivot table. In this example, the pivot table has columns named *A* and *B* and rows named *1* and *2*. You can specify the row and column names explicitly using the *rowlabels* and *collabels* arguments of the `Display` function, as described below.

- The `spsspkg.StartProcedure-spsspkg.EndProcedure` block groups output under a common heading, allowing you to display custom output in the same manner as built-in IBM® SPSS® Statistics procedures such as `DESCRIPTIVES` or `REGRESSION`. You can include multiple pivot tables and text blocks in a given `spsspkg.StartProcedure-spsspkg.EndProcedure` block.

  The argument to the `StartProcedure` function is a string and is the name that appears in the outline pane of the Viewer associated with the output. It is also the command name associated with this output when routing it with OMS (Output Management System), as used in the `COMMANDS` keyword of the `OMS` command. In order that names associated with output not conflict with names of existing SPSS Statistics commands (when working with OMS), consider using names of the form yourorganization.com.procedurename.

  Use of the `StartProcedure` function is optional when creating pivot tables with the `spsspivottable.Display` function. If omitted, the pivot table will appear under an item labeled *R* in the outline pane of the Viewer.

- The only required argument for the `spsspivottable.Display` function is the R object specifying the contents of the pivot table—in this example, the R data frame *demo*.

The full set of arguments to the `Display` function is as follows:

- **x.** The data to be displayed as a pivot table. It may be a data frame, matrix, table, or any R object that can be converted to a data frame.

- **title.** A character string that specifies the title that appears with the table. The default is *Rtable*.

- **templateName.** A character string that specifies the OMS (Output Management System) table subtype for this table. It must begin with a letter and have a maximum of 64 bytes. The default is *Rtable*. Unless you are routing this pivot table with OMS and need to distinguish subtypes, you do not need to specify a value.

  When routing pivot table output from R using OMS, the command name associated with this output is R by default, as in COMMANDS=['R'] for the COMMANDS keyword on the OMS command. If you wrap the pivot table output in a `StartProcedure-EndProcedure` block, then use the name specified in the `StartProcedure` call as the command name.

- **outline.** A character string that specifies a title (for the pivot table) that appears in the outline pane of the Viewer. The item for the table itself will be placed one level deeper than the item for the *outline* title. If omitted, the Viewer item for the table will be placed one level deeper than the root item for the output containing the table, as shown in the above figure.

- **caption.** A character string that specifies a table caption.

- **isSplit.** A logical value (*TRUE* or *FALSE*) specifying whether to enable split file processing for the table. The default is *TRUE*. Split file processing refers to whether results from different split groups are displayed in separate tables or in the same table but grouped by split, and is controlled by the SPLIT FILE command.

- **rowdim.** A character string specifying a title for the row dimension. The default is *row*.

- **coldim.** A character string specifying a title for the column dimension. The default is *column*.

- **hiderowdimtitle.** A logical value (*TRUE* or *FALSE*) specifying whether to hide the row dimension title. The default is *TRUE*.

- **hiderowdimlabel.** A logical value specifying whether to hide the row labels. The default is *FALSE*.

- **hidecoldimtitle.** A logical value specifying whether to hide the column dimension title. The default is *TRUE*.

- **hidecoldimlabel.** A logical value specifying whether to hide the column labels. The default is *FALSE*.

- **rowlabels.** A numeric or character vector specifying the row labels. If provided, the length of the vector must equal the number of rows in the argument *x*. If omitted, the row names of *x* will be used. If *x* does not have row names, the labels *row1*, *row2*, and so on, will be used. If a numeric vector is provided, the row labels will have the format specified by the argument *format*.

- **collabels.** A numeric or character vector specifying the column labels. If provided, the length of the vector must equal the number of columns in the argument *x*. If omitted, the column names of *x* will be used. If *x* does not have column names, the labels *col1*, *col2*, and so on,

will be used. If a numeric vector is provided, the column labels will have the format specified by the argument *format*.

■ **format.** Specifies the format to be used for displaying numeric values, including cell values, row labels, and column labels. The default format is *GeneralStat*, which is most appropriate for unbounded, scale-free statistics. The argument *format* is of the form `formatSpec.format` where `format` is the name of a supported format like *GeneralStat*, as in `formatSpec.GeneralStat`. The list of available formats is provided in the help for the `spsspivottable.Display` function, available in the SPSS Statistics Help system.

Displaying Output from R Functions

Typically, the output from an R analysis—such as a generalized linear model—is an object whose attributes contain the results of the analysis. You can extract the results of interest and render them as pivot tables in IBM® SPSS® Statistics.

Example

In this example, we read the case data from *demo.sav*, create a generalized linear model, and write summary results of the model coefficients back to the SPSS Statistics Viewer as a pivot table.

```
*R_ptable_glm.sps.
GET FILE='/examples/data/demo.sav'.
BEGIN PROGRAM R.
casedata <- spssdata.GetDataFromSPSS(variables=c("car","income","ed","marital"))
model <- glm(car~income+ed+marital,data=casedata)
res <- summary(model)
spsspivottable.Display(res$coefficients,
                       title="Model Coefficients")
END PROGRAM.
```

Result

Figure 28-2
Model coefficients

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	13.698	.449	30.477	.000
income	.220	.002	103.083	.000
ed	.475	.140	3.390	.001
marital	-.162	.335	-.484	.628

■ The R variable *model* contains the results of the generalized linear model analysis.

■ The R `summary` function takes the results of the GLM analysis and produces an R object with a number of attributes that summarize the model. In particular, the *coefficients* attribute contains a matrix of the model coefficients and associated statistics.

   Note: You can obtain a list of the attributes available for an object using `attributes(object)`. To display the attributes from within SPSS Statistics, use `print(attributes(object))`.

■ The `spsspivottable.Display` function creates the pivot table. In the present example, the content for the pivot table is provided by the *coefficients* matrix.

Displaying Graphical Output from R

The IBM® SPSS® Statistics - Integration Plug-In for R provides the ability to display graphical output from R in the IBM® SPSS® Statistics Viewer, allowing you to leverage the rich graphics capabilities available with R.

Example

By default, graphical output from R is rendered in the SPSS Statistics Viewer. In this example, we read the data for the categorical variables *inccat* and *ed* from *demo.sav* and create a mosaic plot displaying the relationship between the variables.

```
*R_graphic.sps.
GET FILE='/examples/data/demo.sav'.
BEGIN PROGRAM R.
dt <- spssdata.GetDataFromSPSS(variables=c("inccat","ed"))
mosaicplot(~ ed + inccat, xlab="Level of Education",
            ylab="Income Category", main="Income Level by Education",
            data = dt, cex.axis=1)
END PROGRAM.
```

Result

Figure 29-1
R graphic displayed in the Viewer
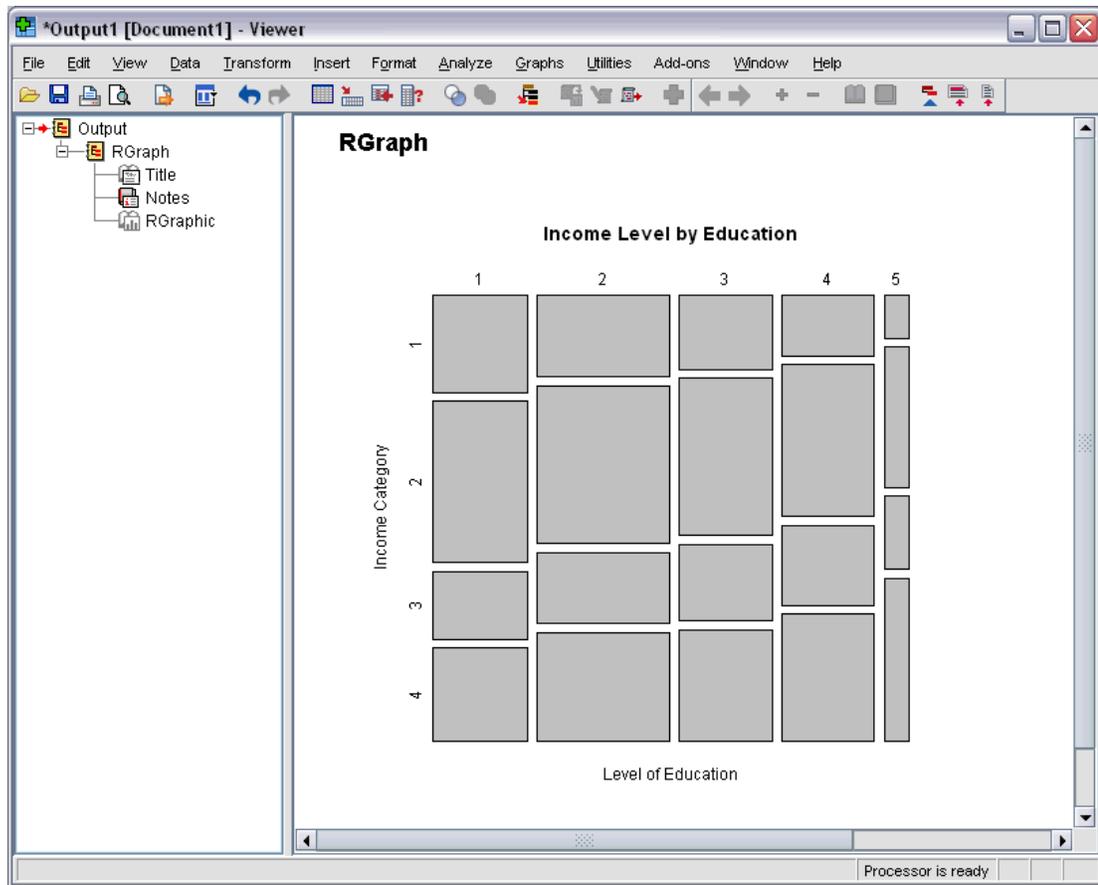
You can turn display of R graphics on or off using the spssRGraphics.SetOutput function. You can display an R graphic file (PNG, JPG, or BMP format) in the SPSS Statistics Viewer using the spssRGraphics.Submit function. For information about using these functions, see the topic on the Integration Plug-In for R in the SPSS Statistics Help system.

Note: R graphics displayed in the SPSS Statistics Viewer cannot be edited and do not use the graphics preference settings in SPSS Statistics.

Retrieving Output from Syntax Commands

The IBM® SPSS® Statistics - Integration Plug-In for R provides the means to retrieve the output produced by syntax commands, allowing you to access command output in a purely programmatic fashion. To retrieve command output, you first route it via the Output Management System (OMS) to an in-memory workspace, referred to as the **XML workspace**, or to a new dataset. Output routed to the XML workspace is accessed with XPath expressions, while output routed to a dataset is accessed by reading the case data from the dataset.

Using the XML Workspace

Output routed to the XML workspace is stored as an XPath DOM that conforms to the IBM® SPSS® Statistics Output XML Schema (*xml.spss.com/spss/oms*). Output is retrieved from the XML workspace with functions that employ XPath expressions.

Constructing the correct XPath expression (SPSS Statistics currently supports XPath 1.0) requires knowledge of the XPath language. If you're not familiar with XPath, this isn't the place to start. In a nutshell, XPath is a language for finding information in an XML document, and it requires a fair amount of practice. If you're interested in learning XPath, a good introduction is the XPath tutorial provided by W3Schools at *http://www.w3schools.com/xpath/*.

In addition to familiarity with XPath, constructing the correct XPath expression requires an understanding of the structure of XML output produced by OMS, which includes understanding the XML representation of a pivot table. You can find an introduction, along with example XML, in the "Output XML Schema" topic in the Help system.

Note: When constructing XPath expressions, it is best to work from a copy of the XML that you're trying to parse. You can route the XML (referred to as OXML) to a file using the OUTFILE keyword of the DESTINATION subcommand of OMS.

Example: Retrieving a Single Cell from a Table

In this example, we'll use output from the DESCRIPTIVES command to determine the percentage of valid cases for a specified variable.

```
*R_get_output_with_xpath.sps.
GET FILE='/examples/data/demo.sav'.
OMS SELECT TABLES
 /IF SUBTYPES=['Descriptive Statistics']
 /DESTINATION FORMAT=OXML XMLWORKSPACE='desc_table'
 /TAG='desc_out'.
DESCRIPTIVES VARIABLES=internet.
OMSEND TAG='desc_out'.
*Get output from the XML workspace using XPath.
BEGIN PROGRAM R.
handle <- "desc_table"
context <- "/outputTree"
xpath <- paste("//pivotTable[@subType='Descriptive Statistics']",
               "/dimension[@axis='row']",
               "/category[@varName='internet']",
               "/dimension[@axis='column']",
               "/category[@text='N']",
               "/cell/@number")
res <- spssxmlworkspace.EvaluateXPath(handle,context,xpath)
ncases <- spssdata.GetCaseCount()
cat("Percentage of valid cases for variable internet: ",
     round(100*as.integer(res)/ncases),"%")
spssxmlworkspace.DeleteXmlWorkspaceObject(handle)
END PROGRAM.
```

- The OMS command is used to direct output from a syntax command to the XML workspace. The XMLWORKSPACE keyword on the DESTINATION subcommand, along with FORMAT=OXML, specifies the XML workspace as the output destination. It is a good practice to use the TAG subcommand, as done here, so as not to interfere with any other OMS requests that may be operating. The identifiers available for use with the SUBTYPES keyword on the IF subcommand can be found in the OMS Identifiers dialog box, available from the Utilities menu in SPSS Statistics.

- The XMLWORKSPACE keyword is used to associate a name with this XPath DOM in the workspace. In the current example, output from the DESCRIPTIVES command will be identified with the name *desc_table*. You can have many XPath DOMs in the XML workspace, each with its own unique name. Note, however, that reusing an existing name will overwrite the contents associated with that name.

- The OMSEND command terminates active OMS commands, causing the output to be written to the specified destination—in this case, the XML workspace.

- You retrieve values from the XML workspace with the EvaluateXPath function. The function takes an explicit XPath expression, evaluates it against a specified XPath DOM in the XML workspace, and returns the result as a vector of character strings.

- The first argument to the EvaluateXPath function specifies the XPath DOM to which an XPath expression will be applied. This argument is referred to as the handle name for the XPath DOM and is simply the name given on the XMLWORKSPACE keyword on the associated OMS command. In this case the handle name is *desc_table*.

- The second argument to EvaluateXPath defines the XPath context for the expression and should be set to "/outputTree" for items routed to the XML workspace by the OMS command.

- The third argument to EvaluateXPath specifies the remainder of the XPath expression (the context is the first part) and must be quoted. Since XPath expressions almost always contain quoted strings, you'll need to use a different quote type from that used to enclose the expression. For users familiar with XSLT for OXML and accustomed to including a

namespace prefix, note that XPath expressions for the `EvaluateXPath` function should not contain the `oms:` namespace prefix.

■ The XPath expression in this example is specified by the variable *xpath*. It is not the minimal expression needed to select the value of interest but is used for illustration purposes and serves to highlight the structure of the XML output.

`//pivotTable[@subType='Descriptive Statistics']` selects the Descriptives Statistics table.

`/dimension[@axis='row']/category[@varName='internet']` selects the row for the variable *internet*.

`/dimension[@axis='column']/category[@text='N']` selects the column labeled *N* (the number of valid cases), thus specifying a single cell in the pivot table.

`/cell/@text` selects the textual representation of the cell contents.

■ When you have finished with a particular output item, it is a good idea to delete it from the XML workspace. This is done with the `DeleteXmlWorkspaceObject` function, whose single argument is the name of the handle associated with the item.

If you're familiar with XPath, you might want to convince yourself that the number of valid cases for *internet* can also be selected with the following simpler XPath expression:

`//category[@varName='internet']//category[@text='N']/cell/@text`

Note: To the extent possible, construct your XPath expressions using language-independent attributes, such as the variable name rather than the variable label. That will help reduce the translation effort if you need to deploy your code in multiple languages. Also, consider factoring out language-dependent identifiers, such as the name of a statistic, into constants. You can obtain the current language used for pivot table output with the `spsspkg.GetOutputLanguage()` function.

You may also consider using text_eng attributes in place of text attributes in XPath expressions. text_eng attributes are English versions of text attributes and have the same value regardless of the output language. The `OATTRS` subcommand of the `SET` command specifies whether text_eng attributes are included in OXML output.

Example: Retrieving a Column from a Table

In this example, we will retrieve a column from the iteration history table for the Quick Cluster procedure and check to see if the maximum number of iterations has been reached.

```
*R_get_table_column.sps.
GET FILE='/examples/data/telco_extra.sav'.
OMS SELECT TABLES
 /IF COMMANDS=['Quick Cluster'] SUBTYPES=['Iteration History']
 /DESTINATION FORMAT=OXML XMLWORKSPACE='iter_table'
 /TAG='iter_out'.
QUICK CLUSTER
  zlnlong zlntoll zlnequi zlncard zlnwire zmultlin zvoice
  zpager zinterne zcallid zcallwai zforward zconfer zebill
  /MISSING=PAIRWISE
  /CRITERIA= CLUSTER(3) MXITER(10) CONVERGE(0)
  /METHOD=KMEANS(NOUPDATE)
  /PRINT INITIAL.
OMSEND TAG='iter_out'.
*Get output from the XML workspace using XPath.
BEGIN PROGRAM R.
mxiter = 10
handle <- "iter_table"
context <- "/outputTree"
xpath <- paste("//pivotTable[@subType='Iteration History']",
               "//dimension[@axis='column']",
               "/category[@number='1']",
               "/cell/@text")
res <- spssxmlworkspace.EvaluateXPath(handle,context,xpath)
if (length(res)==10) cat("Maximum iterations reached for QUICK CLUSTER procedure")
spssxmlworkspace.DeleteXmlWorkspaceObject(handle)
END PROGRAM.
```

As an aid to understanding the code, the iteration history table produced by the QUICK CLUSTER command in this example is shown below.

Figure 30-1
Iteration history table

Iteration	Change in Cluster Centers		
	1	2	3
1	3.298	3.590	3.491
2	1.016	.427	.931
3	.577	.320	.420
4	.240	.180	.195
5	.119	.125	.108
6	.093	.083	.027
7	.069	.094	.032
8	.059	.051	.018
9	.035	.085	.063
10	.025	.359	.333

To further aid in constructing the XPath expression, the OXML representing the first row from the iteration history table is shown below.

```
<pivotTable subType="Iteration History" text="Iteration History">
 <dimension axis="row" text="Iteration">
  <category number="1" text="1">
   <dimension axis="column" text="Change in Cluster Centers">
    <category number="1" text="1">
     <cell decimals="3" format="g" number="3.2980427720769" text="3.298" />
    </category>
    <category number="2" text="2">
     <cell decimals="3" format="g" number="3.5899546987871" text="3.590" />
    </category>
    <category number="3" text="3">
     <cell decimals="3" format="g" number="3.4907178202949" text="3.491" />
    </category>
   </dimension>
  </category>
```

■ The XPath expression in this example selects the values in the column labeled *1*, under the *Change in Cluster Centers* heading, in the iteration history table.

`//pivotTable[@subType='Iteration History']` selects the iteration history table.

`//dimension[@axis='column']` selects all `dimension` elements that represent a column dimension. In the current example, there is one such element for each row in the table.

`/category[@number='1']` selects the `category` element, within the `dimension` element, corresponding to the column labeled *1*.

`/cell/@text` selects the textual representation of the cell contents.

■ The returned value from the `EvaluateXPath` function is an R vector, consisting of the values from column *1* in the iteration history table. Testing the length of the vector determines if the maximum number of iterations has been reached.

Using a Dataset to Retrieve Output

As an alternative to routing output to the XML workspace, you can route it to a new dataset. You can then retrieve values from the dataset using the `spssdata.GetDataFromSPSS` function.

Example

In this example, we'll route output from a `FREQUENCIES` command to a dataset. We'll then use the output to determine the three most frequent values for a specified variable—in this example, the variable *jobtime* from *Employee data.sav*.

```
*R_output_to_dataset.sps.
GET FILE='/examples/data/Employee data.sav'.
DATASET NAME employees.
DATASET DECLARE result.
OMS SELECT TABLES
 /IF COMMANDS=['Frequencies'] SUBTYPES=['Frequencies']
 /DESTINATION FORMAT=SAV OUTFILE='result'
 /TAG='freq_out'.
FREQUENCIES jobtime /FORMAT=DFREQ.
OMSEND TAG='freq_out'.
DATASET ACTIVATE result.
BEGIN PROGRAM R.
data <- spssdata.GetDataFromSPSS(cases=3,variables=c("Var2","Frequency"))
print(data)
END PROGRAM.
```

As a guide to understanding the code, a portion of the output dataset is shown here.

Figure 30-2
Dataset containing output from FREQUENCIES

- The `DATASET NAME` command is used to name the dataset containing *Employee data* in order to preserve it when the output dataset is created.

- The `DATASET DECLARE` command creates a dataset name for the new dataset.

- The `FORMAT=SAV` and `OUTFILE='result'` specifications on the `DESTINATION` subcommand specify that the output from the `OMS` command will be routed to a dataset named *result*.

- Using `/FORMAT=DFREQ` for the `FREQUENCIES` command produces output where categories are sorted in descending order of frequency. Obtaining the three most frequent values simply requires retrieving the first three cases from the output dataset.

- Before case data from *result* can be read into R, the dataset must be activated, as in `DATASET ACTIVATE result`.

- The `GetDataFromSPSS` function is used to read the data. The argument *cases* specifies the number of cases to read and the *variables* argument specifies the particular variables to retrieve. Referring to the portion of the output dataset shown in the previous figure, *Var2* contains the values for *jobtime* and *Frequency* contains the frequencies of these values.

Extension Commands

Extension commands provide the ability to wrap programs written in Python or R in IBM® SPSS® Statistics command syntax. Subcommands and keywords specified in the command syntax are first validated and then passed as argument parameters to the underlying Python or R program, which is then responsible for reading any data and generating any results. Extension commands allow users who are proficient in Python or R to share external functions with users of SPSS Statistics command syntax.

For example, you write a Python module that compares the case data and variable dictionaries of two SPSS Statistics datasets and reports any differences. You can then create an extension command—call it COMPDS—with subcommands and keywords that specify the parameters needed by your Python module. Users only need to know the syntax for the COMPDS command to use your Python module. You can even create a user interface for your extension command, using the Custom Dialog Builder introduced in version 17.0. For more information, see the topic "Creating and Deploying Custom Dialogs for Extension Commands" on p. 395.

Extension commands require the SPSS Statistics Integration Plug-In for the language in which the command is implemented. For instance, an extension command implemented in R requires the IBM® SPSS® Statistics - Integration Plug-In for R. Extension commands require SPSS Statistics release 16.0.1 or later.

Getting Started with Extension Commands

Extension commands require two basic components—an XML file that specifies the syntax of the command and code written in Python or R that implements the command. To illustrate the approach, we'll create a simple extension command called MY FREQUENCIES that executes the IBM® SPSS® Statistics FREQUENCIES command with preset specifications for customized output so that a user only needs to specify the variable list. The command has a single keyword, VARIABLES, that specifies the variable list and will be implemented in Python. For an example of an extension command implemented in R, see "Wrapping R Functions in Extension Commands" on p. 389.

Creating an extension command involves three steps:

▶ Create the XML file that describes the syntax.

▶ Write the implementation code for the command.

▶ Deploy the XML file and the implementation code.

Optionally, you can:

▶ Create a custom dialog that generates the command syntax for your extension command.

▶ Package the XML specification file, implementation code, and any associated custom dialog in an extension bundle so that your extension command can be easily installed by end users. Extension bundles require SPSS Statistics version 18 or higher.

Extension commands of general interest can be shared on Developer Central at *http://www.spss.com/devcentral/*. Users with SPSS Statistics version 18 or higher should package the extension command in an extension bundle. Users of earlier versions should package the extension command in a Zip file. See Developer Central for examples.

Although not required, it is recommended you create a syntax diagram for the extension command before creating the XML specification of the syntax.

Creating Syntax Diagrams

Syntax diagrams specify the syntax of IBM® SPSS® Statistics commands, such as the allowed subcommands, keywords, and keyword values. You'll certainly want to create a syntax diagram for end users of your extension command, and you may even want to make it available from the submitted syntax itself, perhaps with a HELP subcommand as shown for the SPSSINC COMPARE DATASETS command in this section.

A valid syntax diagram for the MY FREQUENCIES command, which only requires a variable list as input, is:

```
MY FREQUENCIES VARIABLES=varlist
```

Experienced users of SPSS Statistics command syntax may assume that VARIABLES is a subcommand. Subcommands for extension commands, however, always begin with a forward slash (/), so VARIABLES is actually a keyword. In addition, all keywords for extension commands must belong to a subcommand. The scenario of parameters that are not part of an explicit subcommand—as in this example—is handled with an unnamed subcommand that precedes any named subcommands and is referred to as the **anonymous** subcommand. In the present example, the VARIABLES keyword is associated with the anonymous subcommand. The specification of the anonymous subcommand is done in the accompanying XML file that describes the syntax diagram and is discussed in the next section.

As a general rule, it is best to use keywords whenever possible since the extension command mechanism provides for validation based on the type of keyword. For example, keywords that describe variable names and dataset names are checked for syntactic correctness (not for the existence of the variables or datasets), and keywords that describe input files are validated to ensure the files exist. The keyword type is specified in the accompanying XML file as discussed in the next section.

An example of a more complex extension command is the SPSSINC COMPARE DATASETS command that compares the data (cases and/or dictionaries) for two datasets and is available for download from Developer Central. The syntax diagram is:

```
SPSSINC COMPARE DATASETS DS1=primary datasetname DS2=secondary datasetname
 [VARIABLES=variable list]

[/HELP]

[/DATA ID=id var [DIFFCOUNT=varname] [ROOTNAME=root name]]

[/DICTIONARY [NONE | [MEASLEVEL TYPE VARLABEL VALUELABELS MISSINGVALUES
ATTRIBUTES FORMAT ALIGNMENT COLUMNWIDTH INDEX]]
```

- The `DS1` and `DS2` keywords are associated with the anonymous subcommand and would be specified as dataset parameters in the associated XML specification of the syntax diagram. The `VARIABLES` keyword would be specified as a variable name list parameter.

- The named subcommands `HELP`, `DATA`, and `DICTIONARY` follow the anonymous subcommand. Notice that all of the parameters associated with these subcommands are keywords. Although you can specify that a subcommand has an explicit value, such as `/VARIABLES = varlist`, this is not recommended since the value cannot be validated by SPSS Statistics (only values associated with keywords can be validated).

- Equals signs (=) are always required for keywords in extension commands that have associated values, as in the `VARIABLES` keyword of the anonymous subcommand for the `SPSSINC COMPARE DATASETS` command.

- The `HELP` subcommand provides an approach for displaying the syntax diagram from the Syntax Editor. Submitted syntax containing the `HELP` subcommand would simply display the syntax diagram and then exit without actually executing the command. For an example of implementing such a `HELP` subcommand, see the `SPSSINC_COMPARE_DATASETS` module packaged with the `SPSSINC COMPARE DATASETS` command and available from Developer Central.

XML Specification of the Syntax Diagram

Once you have created a syntax diagram for your extension command, you translate the diagram into an XML specification of the syntax. Consider the syntax diagram for the `MY FREQUENCIES` command discussed in the previous section.

```
MY FREQUENCIES VARIABLES=varlist
```

The XML specification for a Python implementation of the syntax diagram is:

```
<Command xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="extension.xsd"
  Name="MY FREQUENCIES" Language="Python">
    <Subcommand Name="" Occurrence="Required" IsArbitrary="False">
        <Parameter Name="VARIABLES" ParameterType="VariableNameList"/>
    </Subcommand>
</Command>
```

- The top-level element, Command, names the command. Subcommands are children of this element. The Name attribute is required and specifies the command name. For release 16.0.1, the name must be a single word in upper case with a maximum of eight bytes. For release 17.0 and higher, the name can consist of up to three words separated by spaces, as in `MY EXTENSION COMMAND`, and is not case sensitive. Command names are restricted to 7-bit ascii characters. The Language attribute is optional and specifies the implementation language. The default is the Python programming language. The choices for Language are *Python* or *R*.

- Subcommands are specified with the Subcommand element. All parameters must be associated with a subcommand. In the present example, the `VARIABLES` keyword is associated with the anonymous subcommand, which is specified by setting the Name attribute to the empty string.

  The Occurrence attribute of the Subcommand element is optional and specifies whether the subcommand is required. By default, subcommands are optional.

The IsArbitrary attribute of the Subcommand element is optional and specifies whether arbitrary tokens are allowed on the subcommand. For example, you would use arbitrary tokens to describe syntax, such as `<dependent variable> BY <factor list> WITH <covariate list>`. By default, arbitrary tokens are not allowed.

■ Parameter elements describe the keywords associated with a subcommand. There are many types of parameters, such as VariableNameList for specifying a list of IBM® SPSS® Statistics variable names and InputFile for specifying a filename. Values specified for parameters are checked to ensure they are valid for the particular parameter type. For instance, values specified for a VariableNameList parameter will be checked to be sure they represent syntactically valid SPSS Statistics variable names (the existence of the variables is not checked), and a value specified for an InputFile parameter is validated to ensure it corresponds to an existing file.

An XML file named *MY_FREQUENCIES.xml* and containing the specification for the MY FREQUENCIES command is included in the */examples/extensions* folder of the accompanying examples. To learn where to copy this file in order to use the MY FREQUENCIES command, see "Deploying an Extension Command" on p. 386.

For more examples of XML specifications of extension commands as well as documentation for the underlying XML schema, which includes a listing of all available keyword types, see the article on writing extension commands, available from Developer Central. The documentation for the XML schema is also available from the Extension Schema topic in the SPSS Statistics Help system. For those interested, a copy of the XML schema—*extension-1.0.xsd*—for specifying extension commands can be found in the SPSS Statistics installation directory.

Naming Conventions and Name Conflicts

■ Extension commands take priority over built-in command names. For example, if you create an extension command named MEANS, the built-in MEANS command will be replaced by your extension. Likewise, if an abbreviation is used for a built-in command and the abbreviation matches the name of an extension command, the extension command will be used (abbreviations are not supported for extension commands).

■ To reduce the risk of creating extension command names that conflict with built-in commands or commands created by other users, you should use two- or three-word command names, using the first word to specify your organization.

■ There are no naming requirements for the file containing the XML specification of the syntax. For example, the XML specification for the PLS extension command could be contained in the file *plscommand.xml*. As with choosing the name of the extension command, take care when choosing a name to avoid conflicting XML file names. A useful convention is to use the same name as the Python module, R source file, or R package that implements the command.

Implementation Code

The extension command mechanism requires that the implementation code (whether written in Python or R) reside in a function named Run, which is then contained in a Python module, R source code file, or R package. IBM® SPSS® Statistics parses the command syntax entered by the user and passes the specified values to the Run function in a single argument. Continuing

with the example of the `MY FREQUENCIES` command, consider the following command syntax entered by a user:

```
MY FREQUENCIES VARIABLES = gender educ jobcat.
```

The argument passed to the Python `Run` function, when rendered with the Python `print` command from within the `Run` function (the `MY FREQUENCIES` command is implemented in Python), is as follows:

```
{'MY FREQUENCIES': {'': [{'VARIABLES': ['gender', 'educ', 'jobcat']}]}}
```

The argument is a Python dictionary whose single key is the command name. The value associated with this key is a dictionary that contains all of the specifications provided by the user in the submitted command syntax. Each subcommand is represented by a key in this inner dictionary. The key is the name of the subcommand. In the case of the anonymous subcommand, the key is the empty string as shown above.

The code for the Python module that implements the `MY FREQUENCIES` command and contains the `Run` function is:

```
import spss

def Run(args):
    varlist = args['MY FREQUENCIES'][''][0]['VARIABLES']
    varlist = " ".join(varlist)
    spss.Submit("FREQUENCIES /VARIABLES=%s /BARCHART /FORMAT=NOTABLE." %(varlist))
```

- The module must contain `import` statements for any other modules used by the code. In this case, the code uses the `Submit` function from the `spss` module.

- The `Run` function always takes a single argument containing all of the specifications provided by the user in the submitted syntax.

- The code `args['MY FREQUENCIES'][''][0]['VARIABLES']` extracts the value specified for the `VARIABLES` keyword. In the current example, this is the Python list `['gender', 'educ', 'jobcat']`. Briefly, `args['MY FREQUENCIES']['']` selects the specifications for the anonymous subcommand, resulting in a list. The `[0]` element of that list gives you the innermost dictionary whose single key is `'VARIABLES'`. You then select the value of that key, resulting in a Python list of the specified variable names.

As this trivial example illustrates, the argument passed to the implementation code has a nontrivial structure. As an alternative to manually parsing the argument, consider using the `extension` module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python that greatly simplifies the task of argument parsing for extension commands implemented in Python. For more information, see the topic "Using the Python extension Module" on p. 387.

The Python module file *MY_FREQUENCIES.py* containing the implementation code described here is included in the */examples/extensions* folder of the accompanying examples. To learn where to copy this file to in order to use the `MY FREQUENCIES` command, see "Deploying an Extension Command" on p. 386.

Naming Conventions

The Python module, R source file, or R package containing the `Run` function that implements an extension command must adhere to the following naming conventions:

- **Python.** The `Run` function must reside in a Python module file with the same name as the command—for instance, in the Python module file *MYCOMMAND.py* for an extension command named `MYCOMMAND`. The name of the Python module file must be in upper case, although the command name itself is case insensitive. For multi-word command names, replace the spaces between words with underscores. For example, for an extension command with the name `MY COMMAND`, the associated Python module would be *MY_COMMAND.py*.

- **R.** The `Run` function must reside in an R source file or R package with the same name as the command—for instance, in a source file named *MYRFUNC.R* for an extension command named `MYRFUNC`. The name of the R source file or package must be in upper case, although the command name itself is case insensitive. For multi-word command names, replace the spaces between words with underscores for R source files and periods for R packages. For example, for an extension command with the name `MY RFUNC`, the associated R source file would be named *MY_RFUNC.R*, whereas an R package that implements the command would be named *MY.RFUNC.R*. The source file or package should include any `library` function calls required to load R functions used by the code. *Note*: Use of multi-word command names for R extension commands requires SPSS Statistics release 17.0.1 or later.

Deploying an Extension Command

Using an extension command requires that IBM® SPSS® Statistics can access both the XML syntax specification file and the implementation code (Python module, R source file, or R package). If the extension command is distributed in an extension bundle (.spe) file, then you can simply install the bundle from Utilities>Extension Bundles>Install Extension Bundle within SPSS Statistics (extension bundles require SPSS Statistics version 18 or higher). Otherwise, you will need to manually install the XML syntax specification file and the implementation code. Both should be placed in the *extensions* directory, located at the root of the SPSS Statistics installation directory. For Mac, the installation directory refers to the *Contents* directory in the SPSS Statistics application bundle.

Note: For version 18 on Mac, the files can also be placed in */Library/Application Support/SPSSInc/PASWStatistics/18/extensions*. For version 19 and higher on Mac, the files can also be placed in */Library/Application Support/IBM/SPSS/Statistics/<version>/extensions*, where <version> is the two digit SPSS Statistics version—for example, 19.

- If you do not have write permissions to the SPSS Statistics installation directory or would like to store the XML file and the implementation code elsewhere, you can specify one or more alternate locations by defining the *SPSS_EXTENSIONS_PATH* environment variable. When present, the paths specified in *SPSS_EXTENSIONS_PATH* take precedence over the *extensions* subdirectory of the SPSS Statistics installation directory. The *extensions* subdirectory is always searched after any locations specified in the environment variable. Note that Mac users may also utilize the *SPSS_EXTENSIONS_PATH* environment variable. For multiple locations, separate each with a semicolon on Windows and a colon on Linux and Mac.

- For an extension command implemented in Python, you can always store the associated Python module to a location on the Python search path (such as the Python *site-packages* directory), independent of where you store the XML specification file. The *extensions* subdirectory and any other directories specified in *SPSS_EXTENSIONS_PATH* are automatically added to the Python search path when SPSS Statistics starts.

- For an extension command implemented in R, the R source file or R package containing the implementation code should be installed to the directory containing the XML syntax specification file. R packages can alternatively be installed to the default location for the associated platform—for example, *R_Home/library* on Windows, where *R_Home* is the installation location of R and *library* is a subdirectory under that location. For help with installing R packages, consult the *R Installation and Administration* guide, distributed with R.

At startup, SPSS Statistics reads the *extensions* directory and any directories specified in *SPSS_EXTENSIONS_PATH*, and registers the extension commands found in those locations. If you want to load a new extension command without restarting SPSS Statistics you will need to use the EXTENSION command (see the SPSS Statistics Help system or the *Command Syntax Reference* for more information).

Note: If you or your end users will be running an extension command while in distributed mode, be sure that the extension command files (XML specification and implementation code) and the relevant SPSS Statistics Integration Plug-In(s) (Python and/or R) are installed to both the client and server machines.

Enabling Color Coding and Auto-Completion in the Syntax Editor

The XML syntax specification file contains all of the information needed to provide color coding and auto-completion for your extension command in the Syntax Editor. For SPSS Statistics release 18 and later these features are automatically enabled. To enable these features for release 17, place a copy of the XML file in the *syntax_xml* directory—located at the root of the SPSS Statistics installation directory for Windows, and under the *bin* subdirectory of the installation directory for Linux and Mac. The contents of the *syntax_xml* directory are read when SPSS Statistics starts up.

Using the Python extension Module

The Python extension module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python, greatly simplifies the task of parsing the argument passed to the Run function for extension commands implemented in the Python programming language. To illustrate the approach, consider rewriting the Python module that implements the MY FREQUENCIES command (from "Implementation Code" on p. 384) using the extension module.

The syntax diagram for the MY FREQUENCIES command is:

```
MY FREQUENCIES VARIABLES=varlist
```

The code for the Python module `MY_FREQUENCIES` that implements the command using the `extension` module, including all necessary `import` statements, is:

```
from extension import Syntax, Template, processcmd
import spssaux, spss

def Run(args):
    synObj = Syntax([Template(kwd="VARIABLES", subc="", var="varlist",
                              islist = True, ktype="existingvarlist")])
    processcmd(synObj,args['MY FREQUENCIES'],myfreq,vardict=spssaux.VariableDict())

def myfreq(varlist):
    varlist = " ".join(varlist)
    spss.Submit("FREQUENCIES /VARIABLES=%s /BARCHART /FORMAT=NOTABLE." %(varlist))
```

■ The module consists of the `Run` function that parses the values passed from IBM® SPSS® Statistics and the `myfreq` function that implements the customized version of the `FREQUENCIES` command. In more complex cases, you will probably want to separate the code that does the parsing from the code that implements the actual functionality. For instance, you might split off the `myfreq` function into a separate Python module and import that module in the module that contains the `Run` function. For an example of this approach, see the `SPSSINC MODIFY OUTPUT` command, available from Developer Central.

■ The `Template` class from the `extension` module is used to specify a keyword. Each keyword of each subcommand should have an associated instance of the `Template` class. In this example, `VARIABLES` is the only keyword and it belongs to the anonymous subcommand.

The argument *kwd* to the `Template` class specifies the name of the keyword.

The argument *subc* to the `Template` class specifies the name of the subcommand that contains the keyword. If the keyword belongs to the anonymous subcommand, the argument *subc* can be omitted or set to the empty string as shown here.

The argument *var* specifies the name of the Python variable that receives the value specified for the keyword. In this case, the Python variable *varlist* will contain the variable list specified for the `VARIABLES` keyword. If *var* is omitted, the lowercase value of *kwd* is used.

The argument *islist* specifies whether the value of the keyword is a list. In this case, *islist* is set to *True* since the keyword value is a variable list. By default, *islist* is *False*.

The argument *ktype* specifies the type of keyword, such as whether the keyword specifies a variable name, a string, or a floating point number. In this example, the keyword defines a variable list and is specified as the type *existingvarlist*. The *existingvarlist* type validates the existence of the specified variables and expands any `TO` and `ALL` constructs used in the specification. In that regard, the `extension` module supports `TO` and `ALL` in variable lists.

■ The `Syntax` class from the `extension` module validates the syntax specified by the `Template` objects. You instantiate the `Syntax` class with a sequence of one or more `Template` objects. In this example, there is only one `Template` object so the argument to the `Syntax` class is a list with a single element.

■ The `processcmd` function from the `extension` module parses the values passed to the `Run` function and executes the implementation function.

The first argument to the `processcmd` function is the Syntax object for the command, created from the `Syntax` class.

The argument passed to the `Run` function consists of a dictionary with a single key whose name is the command name and whose value contains the specified syntax (see "Implementation Code" on p. 384). The second argument to the `processcmd` function is this value. In this example, it is given by `args['MY FREQUENCIES']`. It can also be expressed more generally as `args[args.keys()[0]]`.

The third argument to `processcmd` is the name of the implementation function—in this case, `myfreq`. The values of the keywords specified by the `Template` objects are passed to the implementation function as a set of keyword arguments. In the present example, the function `myfreq` will be called with the following signature: `myfreq(varlist=<value of VARIABLES keyword>)`. The `processcmd` function checks for required parameters by scanning the signature of the implementation function for parameters that do not have default values.

Note: If a Python exception is raised in the implementation function, the Python traceback is suppressed, but the error message is displayed.

The *vardict* argument to the `processcmd` function is used when a keyword of type *existingvarlist* is included in one of the `Template` objects. It is used to expand and validate the variable names (the `extension` module supports `TO` and `ALL` in variable lists). Its value should be set to an instance of the `VariableDict` class for the active dataset, as in `spssaux.VariableDict()`.

■ The `myfreq` function takes the variable list specified by the user and calls the `FREQUENCIES` command with this list.

You can obtain additional help for the `extension` module by including the statement `help(extension)` in a program block, once you've imported the module, or by reading the detailed comments in the module.

For a more complex example of using the `extension` module, see the `SPSSINC_COMPARE_DATASETS` module, packaged with the `SPSSINC COMPARE DATASETS` command and available from Developer Central. It presents a more complex syntax diagram and shows how to handle optional keywords that take a Boolean value.

Wrapping R Functions in Extension Commands

The ability to implement extension commands in R opens up the vast libraries of R functions to users of IBM® SPSS® Statistics command syntax. Wrapping an R function in an extension command is relatively straightforward, requiring the same steps described in "Getting Started with Extension Commands" on p. 381. As an example, we'll wrap the `polychor` function from the `polycor` package (available from any CRAN mirror site) in an extension command named `RPOLYCHOR`. In its simplest usage, the function computes the correlation between two ordinal variables. The function has the following signature:

```
polychor(x,y,ML=FALSE,control=list(),std.err=FALSE,maxcor=.9999)
```

To simplify the associated syntax, we'll omit all parameters other than the two variables *x* and *y* and the *maxcor* parameter and consider the case where *x* and *y* are numeric variables. For a general treatment, see the `SPSSINC HETCOR` extension command, available from Developer Central.

XML Specification of the Command Syntax

The first step is to create a syntax diagram for the command and then translate the diagram into an XML specification of the syntax. A syntax diagram for the RPOLYCHOR command is:

```
RPOLYCHOR VARIABLES=varlist
   /OPTIONS MAXCOR = {.9999**}
                     {value  }
```

The corresponding XML specification, based on the extension schema, would then be:

```
<Command xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="extension.xsd" Name="RPOLYCHOR" Language="R">
    <Subcommand Name="" IsArbitrary="False" Occurrence="Required">
        <Parameter Name="VARIABLES" ParameterType="VariableNameList"/>
    </Subcommand>
    <Subcommand Name="OPTIONS">
        <Parameter Name="MAXCOR" ParameterType="Number"/>
    </Subcommand>
</Command>
```

- The Command element names the command RPOLYCHOR. The Language attribute specifies R as the implementation language.

- The VARIABLES keyword, associated with the anonymous subcommand, is used to specify the input variables. It has a parameter type of VariableNameList. Values specified for VariableNameList parameters are checked to be sure they represent syntactically valid SPSS Statistics variable names (the existence of the variables is not checked).

- The OPTIONS Subcommand element contains a Parameter element for the value of *maxcor*. The parameter type is specified as Number, which means that the value can be a number, possibly in scientific notation using e or E.

The */examples/extensions* folder of the accompanying examples contains the files *RPOLYCHOR1.xml* and *RPOLYCHOR2.xml* that specify the RPOLYCHOR command shown here. The files are identical except *RPOLYCHOR1.xml* specifies R as the implementation language and *RPOLYCHOR2.xml* specifies Python as the implementation language. To learn where to copy these file in order to use the RPOLYCHOR command, see "Deploying an Extension Command" on p. 386.

Implementation Code

When wrapping an R function in an extension command, different architectures for the implementation code are available and depend on your version of SPSS Statistics. For SPSS Statistics version 18 and higher it is recommended to use the R source file approach.

- **R source file.** The implementation code is contained in an R source file. This approach requires that you and your end users have R and the IBM® SPSS® Statistics - Integration Plug-In for R installed on machines that will run the extension command, and is only available for SPSS Statistics version 18 and higher. This is by far the simplest approach and is the recommended method for users who have SPSS Statistics version 18 or higher. An example of this approach is described in "R Source File" on p. 391.

- **Wrapping in python.** You can wrap the code that implements the R function in Python. This is the recommended approach for users who do not have SPSS Statistics version 18 or higher or who need to customize the output with Python scripts. This approach requires that you and your end users have R, Python, the Integration Plug-In for R, and the IBM® SPSS® Statistics

- Integration Plug-In for Python installed on machines that will run the extension command. An example of this approach is described in "Wrapping R Code in Python" on p. 393. *Note*: Full support for this approach requires SPSS Statistics version 17.0.1 or higher.

■ **R package.** The implementation code is contained in an R package. This approach is the most involved because it requires creating and installing R packages, but it allows you to potentially distribute your package through the Comprehensive R Archive Network (CRAN). This approach requires that you and your end users have R and the Integration Plug-In for R installed on machines that will run the extension command. For users with SPSS Statistics version 18 or higher, the approach for creating the implementation code is the same as for the R source file approach but requires the further step of creating an R package containing the implementation code. If you are interested in this approach but are not familiar with creating R packages, you may consider creating a skeleton package using the R `package.skeleton` function (distributed with R). If you and any end users do not have SPSS Statistics version 18 or higher, then you will have to manually parse the argument passed to the `Run` function.

It is also possible to generate an R program directly from a custom dialog, bypassing the extension method entirely. However, the entire program will then appear in the log file, and extra care must be taken with long lines of code. For an example of this approach, see the Rboxplot example, available from the Developer Central.

R Source File

To wrap an R function, you create an R source file containing a `Run` function that parses and validates the syntax specified by the end user, and another function—called by `Run`—that actually implements the command. Following the example of the `RPOLYCHOR` extension command, the associated Run function is:

```
Run<-function(args){
    args <- args[[2]]
    oobj<-spsspkg.Syntax(templ=list(
            spsspkg.Template(kwd="VARIABLES",subc="",ktype="existingvarlist",
                            var="vars",islist=TRUE),
            spsspkg.Template(kwd="MAXCOR",subc="OPTIONS",ktype="float",var="maxcor")
            ))
    res <- spsspkg.processcmd(oobj,args,"rpolycor")
}
```

■ IBM® SPSS® Statistics parses the command syntax entered by the user and passes the specified values to the `Run` function in a single argument—*args* in this example. The argument is a list structure whose first element is the command name and whose second element is a set of nested lists containing the values specified by the user.

■ The `Run` function contains calls to the `spsspkg.Syntax`, `spsspkg.Template`, and `spsspkg.processcmd` functions, which are designed to work together.

   `spsspkg.Template` specifies the details needed to process a specified keyword in the syntax for an extension command.

   `spsspkg.Syntax` validates the values passed to the `Run` function according to the templates specified for the keywords.

   `spsspkg.processcmd` parses the values passed to the `Run` function and calls the function that will actually implement the command—in this example, the function *rpolycor* (discussed below) which resides in the same source file as the `Run` function.

Note: Complete documentation for these functions is available from the SPSS Statistics Help system.

- You call `spsspkg.Template` once for each keyword supported by the extension command. In this example, the extension command contains the two keywords VARIABLES and MAXCOR, so `spsspkg.Template` is called twice. The function returns an object, that we'll refer to as a template object, for use with the `spsspkg.Syntax` function.

- The argument *kwd* to `spsspkg.Template` specifies the name of the keyword (in uppercase) for which the template is being defined.

- The argument *subc* to `spsspkg.Template` specifies the name of the subcommand (in uppercase) that contains the keyword. If the keyword belongs to the anonymous subcommand, the argument *subc* can be omitted or set to the empty string as shown here.

- The argument *ktype* to `spsspkg.Template` specifies the type of keyword, such as whether the keyword specifies a variable name, a string, or a floating point number.

- The value *existingvarlist* for *ktype* specifies a list of variable names that are validated against the variables in the active dataset. It is used for the VARIABLES keyword that specifies the variables for the analysis.

- The value *float* for *ktype* specifies a real number. It is used for the MAXCOR keyword.

- The argument *var* to `spsspkg.Template` specifies the name of an R variable that will be set to the value specified for the keyword. This variable will be passed to the implementation function by the `spsspkg.processcmd` function.

- The optional argument *islist* to `spsspkg.Template` is a boolean (*TRUE* or *FALSE*) specifying whether the keyword contains a list of values. The default is *FALSE*. The keyword VARIABLES in this example is a list of variable names, so it should be specified with *islist=TRUE*.

- Once you have specified the templates for each of the keywords, you call `spsspkg.Syntax` with a list of the associated template objects. The returned value from `spsspkg.Syntax` is passed to the `spsspkg.processcmd` function, which has the following required arguments:

  The first argument is the value returned from the `spsspkg.Syntax` function.

  The second argument is the list structure containing the values specified by the user in the submitted syntax.

  The third argument is the name of the function that will actually implement the extension command—in this example, the function `rpolycor`.

```
rpolycor<- function(vars,maxcor=0.9999){
library(polycor)
x <- vars[[1]]
y <- vars[[2]]

# Get the data from the active dataset and run the analysis
data <- spssdata.GetDataFromSPSS(variables=c(x,y),missingValueToNA=TRUE)
result <- polychor(data[[x]],data[[y]],maxcor=maxcor)

# Create output to display the result in the Viewer
spsspkg.StartProcedure("Polychoric Correlation")
spsspivottable.Display(result,title="Correlation",
                        rowlabels=c(x),
                        collabels=c(y),
                        format=formatSpec.Correlation)
spsspkg.EndProcedure()
}
```

- The `spsspkg.processcmd` function calls the specified implementation function—in this case, *rpolycor*—with a set of named arguments, one for each template object. The names of the arguments are the values of the *var* arguments specified for the associated template objects and the values of the arguments are the values of the associated keywords from the syntax specified by the end user.

- The implementation function contains the `library` statement for the R `polycor` package.

- The `spssdata.GetDataFromSPSS` function reads the case data for the two specified variables from the active dataset. `missingValueToNA=TRUE` specifies that missing values of numeric variables are converted to the R *NA* value (by default, they are converted to the R *NaN* value). The data are then passed to the R `polychor` function to compute the correlation.

- You group output under a common heading using a `spsspkg.StartProcedure`-`spsspkg.EndProcedure` block. The argument to `spsspkg.StartProcedure` specifies the name that appears in the outline pane of the Viewer associated with the output.

- The `spsspivottable.Display` function creates a pivot table that is displayed in the SPSS Statistics Viewer. The first argument is the data to be displayed as a pivot table. It can be a data frame, matrix, table, or any R object that can be converted to a data frame.

An R source code file named *RPOLYCHOR.R* containing the implementation code described here is included in the */examples/extensions* folder of the accompanying examples. This implementation code should be used with the XML specification file *RPOLYCHOR1.xml*, also available in the */examples/extensions* folder.

The data file *polychor.sav*, included with the accompanying examples in the */examples/data* folder, contains sample data for testing the RPOLYCHOR extension command.

Wrapping R Code in Python

To wrap the implementation code for an R function in Python, you create a Python function that generates the necessary R code and submits it to IBM® SPSS® Statistics in a BEGIN PROGRAM R–END PROGRAM block. Following the example of the RPOLYCHOR extension command, the code for the Python module—which must be named RPOLYCHOR—that implements the command, including all necessary import statements, is:

```
import spss, spssaux
```

```
from extension import Template, Syntax, processcmd

def Run(args):

    args = args[args.keys()[0]]

    oobj = Syntax([
      Template("VARIABLES",subc="",ktype="existingvarlist",var="vars",islist=True),
      Template("MAXCOR",subc="OPTIONS",ktype="float",var="maxcor")])

    processcmd(oobj, args, rpolychor, vardict=spssaux.VariableDict())

def rpolychor(vars, maxcor=0.9999):

        varX = vars[0]
        varY = vars[1]
        pgm = r"""BEGIN PROGRAM R.
library(polycor)
data <- spssdata.GetDataFromSPSS(variables=c("%(varX)s","%(varY)s"),missingValueToNA=TRUE)
result <- polychor(data[["%(varX)s"]],data[["%(varY)s"]],maxcor=%(maxcor)s)

# Create output to display the result in the Viewer
spsspkg.StartProcedure("Polychoric Correlation")
spsspivottable.Display(result,title="Correlation",
                        rowlabels=c("%(varX)s"),
                        collabels=c("%(varY)s"),
                        format=formatSpec.Correlation)
spsspkg.EndProcedure()
END PROGRAM.
""" % locals()

        spss.Submit(pgm)
```

- The module consists of the `Run` function that parses the values passed from SPSS Statistics and a Python function named `rpolychor` that generates the BEGIN PROGRAM R–END PROGRAM block that calls the R `polychor` function.

- The `Run` function uses the Python `extension` module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python, to parse the arguments passed from SPSS Statistics and to pass those arguments to the `rpolychor` function. For more information, see the topic "Using the Python extension Module" on p. 387.

- The `rpolychor` function generates a BEGIN PROGRAM R–END PROGRAM block of command syntax containing all of the R code needed to get the data from SPSS Statistics, call the R `polychor` function, and display the results in a pivot table in the SPSS Statistics Viewer. The block is submitted to SPSS Statistics with the `spss.Submit` function.

As an alternative to directly submitting the BEGIN PROGRAM R–END PROGRAM block, you can write the block to an external file and use the INSERT command to run the syntax. Writing the block to an external file has the benefit of saving the generated R code for future use. The following code—which replaces `spss.Submit(pgm)` in the previous example—shows how to do this in the case that the block is written to a file in the directory currently designated as the temporary directory. To use this code, you will also have to import the `tempfile`, `os`, and `codecs` modules.

```
cmdfile = (tempfile.gettempdir() + os.sep + "pgm.R").replace("\\", "/")
f = codecs.open(cmdfile, "wb", encoding="utf_8_sig")
f.write(pgm)
f.close()
spss.Submit("INSERT FILE='%s'" % cmdfile)
```

- Setting `encoding="utf_8_sig"` means that the file is written in UTF-8 with a byte order mark (BOM). This ensures that SPSS Statistics will properly handle any extended ascii characters in the file when the file is read with the INSERT command.

As a useful convention, you may want to consider adding a `SAVE` subcommand with a `PROGRAMFILE` keyword to your extension command to let the user decide whether to save the generated R code. For an example of this approach, see the `SPSSINC HETCOR` command, installed with IBM® SPSS® Statistics - Essentials for R.

A Python module named *RPOLYCHOR.py* containing the implementation code described here is included with the accompanying examples in the */examples/extensions* folder. This implementation code should be used with the XML specification file *RPOLYCHOR2.xml*, also available in the */examples/extensions* folder.

The SPSS Statistics data file *polychor.sav*, included with the accompanying examples in the */examples/data* folder, contains sample data for testing the `RPOLYCHOR` extension command.

Creating and Deploying Custom Dialogs for Extension Commands

Using the Custom Dialog Builder, available with IBM® SPSS® Statistics, you can create a custom dialog that generates the syntax for your extension command. The dialog can be deployed along with the command, allowing end users to access the dialog from within SPSS Statistics. Once you have created an extension command, the basic steps involved in building a custom dialog are:

▶ Specify the properties of the dialog itself, such as a new menu item for the dialog within the SPSS Statistics menus.

▶ Specify the controls, such as source and target variable lists, that make up the dialog and any sub-dialogs.

▶ Create the syntax template that specifies the command syntax to be generated by the dialog.

Once you have completed the specifications for the dialog, you can install it and begin using it. You can also package the dialog along with the files for your extension command in an extension bundle so that your custom solution can be easily installed by end users. Extension bundles require SPSS Statistics version 18 or higher.

As an example of the approach, we will create a custom dialog for the `RPOLYCHOR` extension command described in "Wrapping R Functions in Extension Commands" on p. 389.

Figure 31-1
Polychoric Correlation dialog
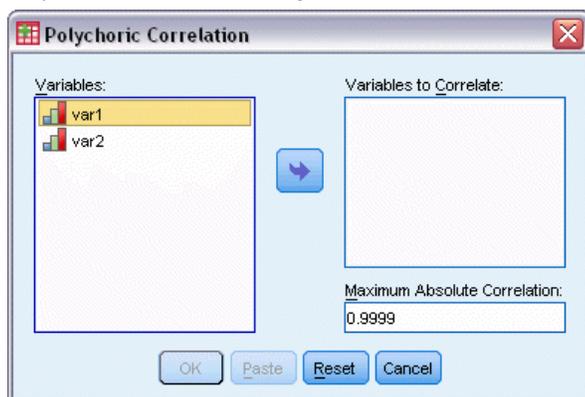
The dialog will generate command syntax like that shown here.

```
RPOLYCHOR VARIABLES=var1 var2
    /OPTIONS MAXCOR=0.9999.
```

Detailed help on creating and managing custom dialogs is available in the SPSS Statistics Help system. You may also want to consult the Tutorial, "Creating Custom Dialogs", available from Help>Tutorial>Customizing SPSS Statistics. Additional examples of custom dialogs can be found in the tutorials available from Help>Working with R.

Creating the Dialog and Adding Controls

To create a new custom dialog:

▶ From the menus in IBM® SPSS® Statistics choose:
Utilities > Custom Dialogs > Custom Dialog Builder

This will launch the Custom Dialog Builder, which is an interactive window that allows you to design and preview a custom dialog.

Figure 31-2
Custom Dialog Builder
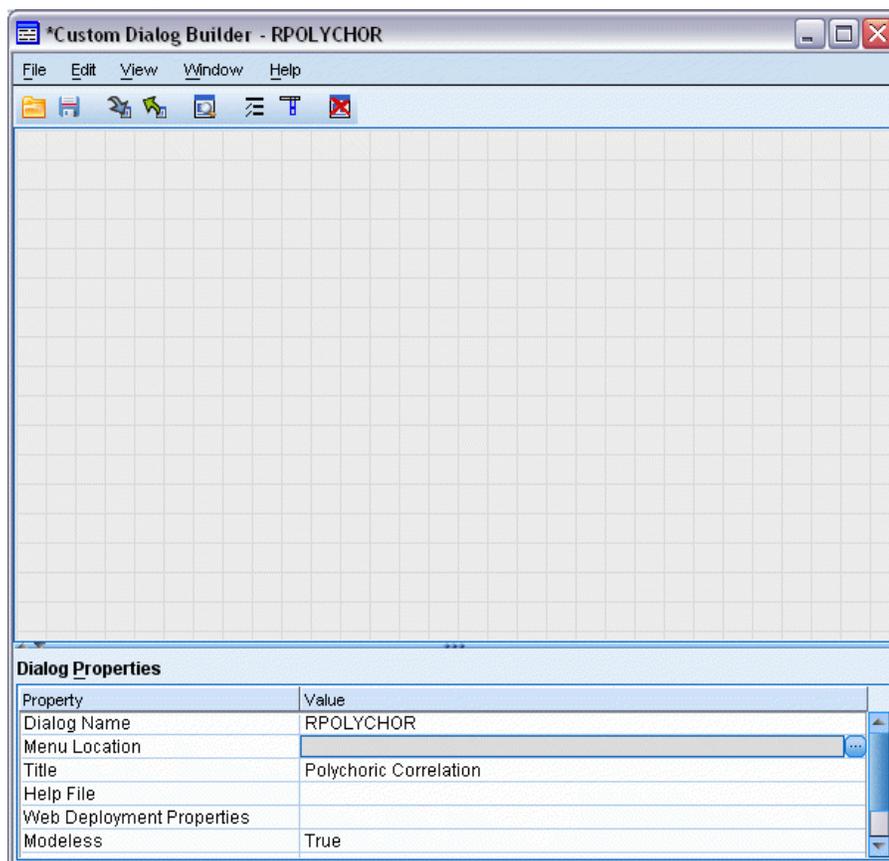
▶ Enter RPOLYCHOR for the Dialog Name. The Dialog Name is the identifier for the dialog and is a required property.

▶ Enter Polychoric Correlation for the Title. The Title property specifies the text to be displayed in the title bar of the dialog box.

▶ Click the ellipsis (...) button in the *Value* column for the Menu Location property to specify the location of a new menu item for your dialog.

Figure 31-3
Menu Location dialog box
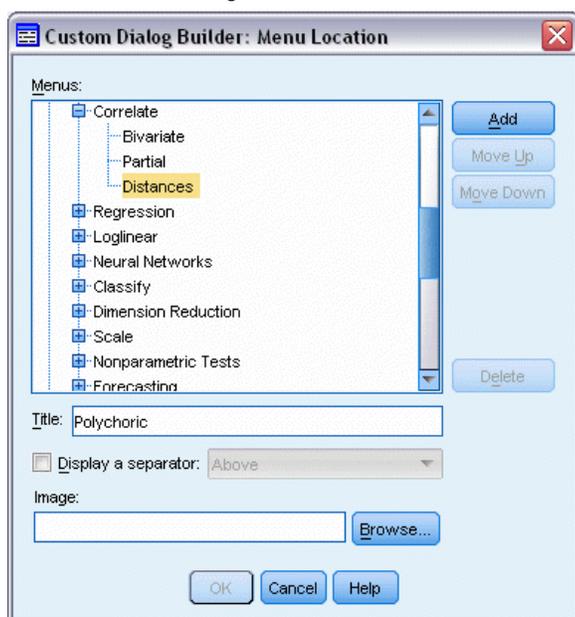
The Menu Location dialog box allows you to specify the name and location of a new menu item for your dialog. Since we're creating a procedure for calculating correlations, we'll add a new menu item to the Correlate sub-menu of the Analyze menu.

▶ Expand the items for the Analyze menu and the Correlate sub-menu.

▶ Choose Distances on the Correlate sub-menu.

▶ In the Title text box, enter Polychoric.

▶ Click Add to add the new item and then click OK.

The RPOLYCHOR command requires the names of the two variables whose correlation is being calculated, as well as an optional specification of the maximum absolute correlation.

▶ From the Tools Palette, drag a Source List control, a Target List control, and a Number control into the Custom Dialog Builder window, then click on the source list control to display the Source Variable List Properties pane.

Figure 31-4
Source Variable List Properties
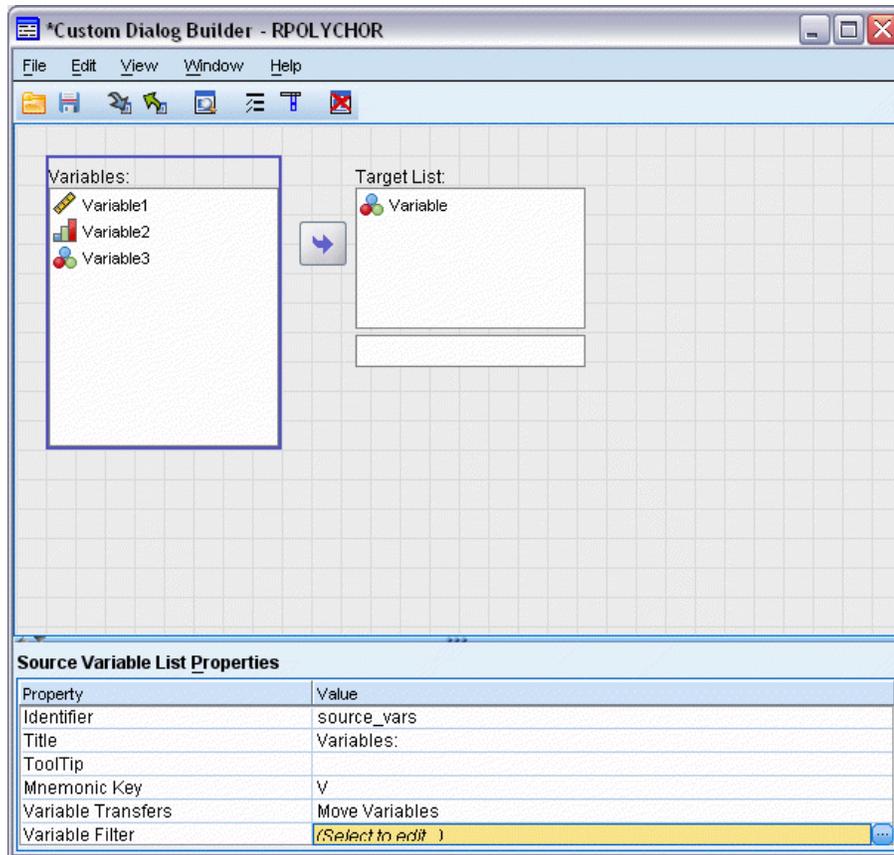
▶ Enter source_vars for the value of the Identifier property. Each control must have a unique identifier.

▶ Leave the default value of Variables: for the Title property.

▶ Enter V for the value of the Mnemonic Key property to specify a character in the title to use as a keyboard shortcut to the control. The character appears underlined in the title. *Note*: The Mnemonic Key property is not supported on Mac.

▶ Leave the default value of Move Variables for the Variable Transfers property. This specifies that variables transferred from the source list to a target list are removed from the source list.

▶ Click the ellipsis (...) button in the *Value* column for the Variable Filter property to filter the types of variables contained in the source list.

Figure 31-5
Filter dialog box
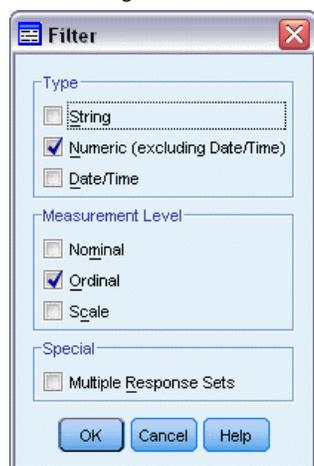
► Deselect (uncheck) String and Date/Time in the Type group.

► Deselect (uncheck) Nominal and Scale in the Measurement Level group.

► Click OK.

► Click on the target list control to display the Target Variable List Properties pane.

Figure 31-6
Target Variable List Properties
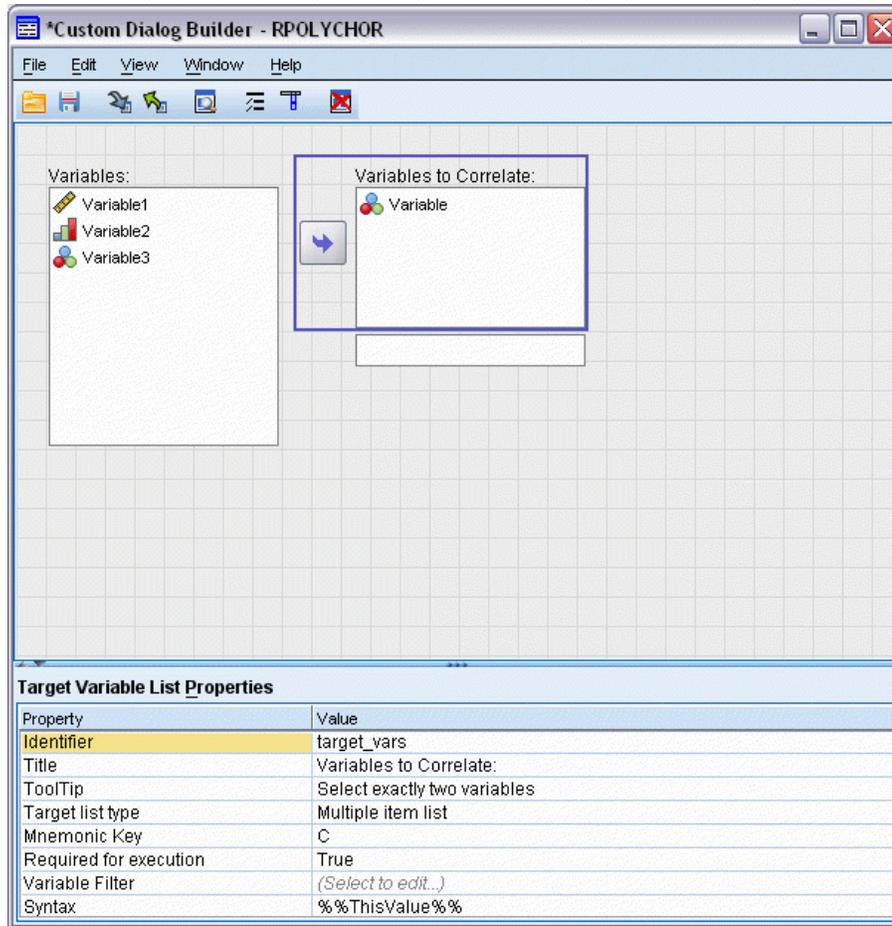
► Enter target_vars for the value of the Identifier property.

► Enter Variables to Correlate: for the value of the Title property.

► Enter Select exactly two variables for the value of the ToolTip property. The specified text will appear when the user hovers over the title area of the target list control.

► Enter C for the value of the Mnemonic Key property.

► Leave the default value of True for the Required for execution property. This specifies that the OK and Paste buttons will remain disabled until a value is specified for this control.

The Syntax property specifies the command syntax to be generated by this control at run time. Notice that it has a default value of %%ThisValue%%. This specifies that the syntax generated by the control will consist of the run-time value of the control, which is the list of variables transferred to the control. Leave the default value.

► Click on the Number control to display its properties pane.

Figure 31-7
Number Control Properties
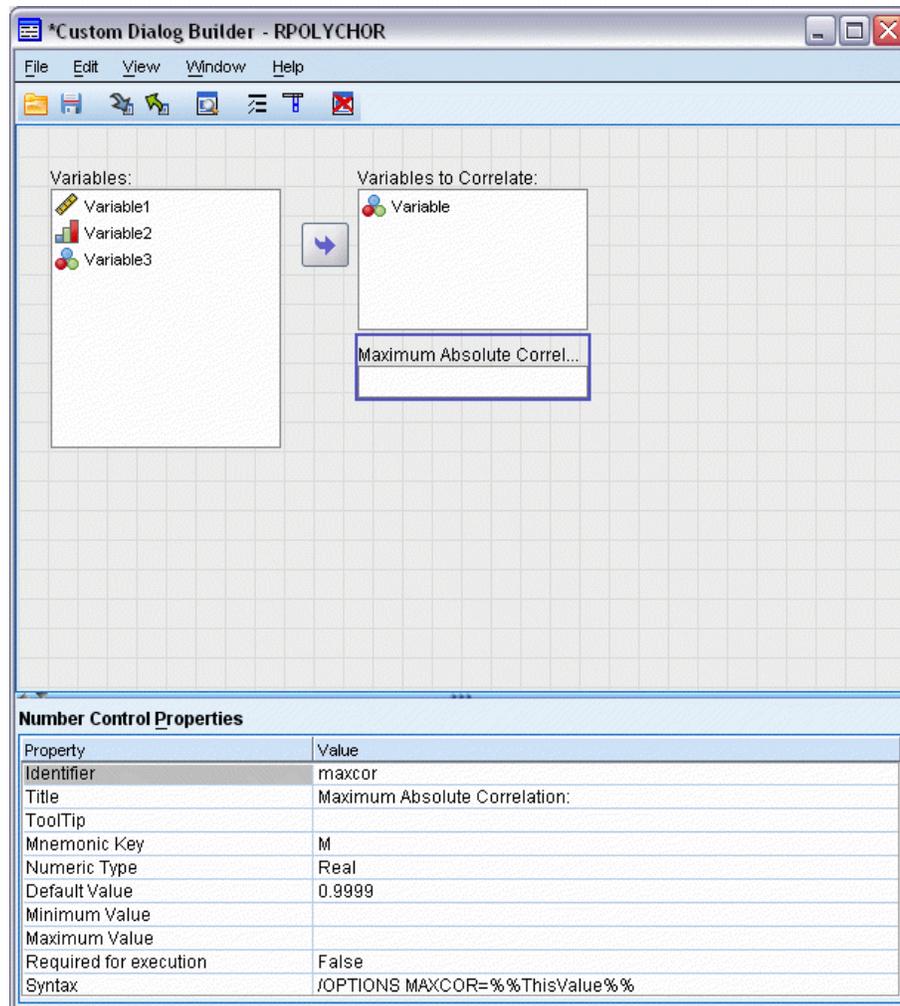
▶ Enter maxcor for the value of the Identifier property.

▶ Enter Maximum Absolute Correlation: for the value of the Title property.

▶ Enter M for the value of the Mnemonic Key property.

▶ Leave the default value of Real for the Numeric Type property.

▶ Enter 0.9999 for the value of the Default Value property.

▶ Leave the default value of False for the Required for execution property.

▶ Modify the Syntax property so that it has the value:

```
/OPTIONS MAXCOR=%%ThisValue%%
```

The value `%%ThisValue%%` specifies the run-time value of the control, which is the specified numeric value, if any. If the run-time value is empty, then the number control does not generate any command syntax, meaning that the `OPTIONS` subcommand is not included in the generated syntax.

Creating the Syntax Template

The syntax template specifies the command syntax to be generated by the custom dialog—in this case, the syntax for the `RPOLYCHOR` extension command. It may consist of static text that is always generated and control identifiers that are replaced at run-time with the values of the associated custom dialog controls. The syntax template for the Polychoric Correlation dialog is very simple since the only inputs from the dialog are the two variables to correlate and the optional value of the maximum absolute correlation parameter.

To access the syntax template:

▶ From the menus in the Custom Dialog Builder choose:
Edit > Syntax Template

Figure 31-8
Syntax Template dialog box
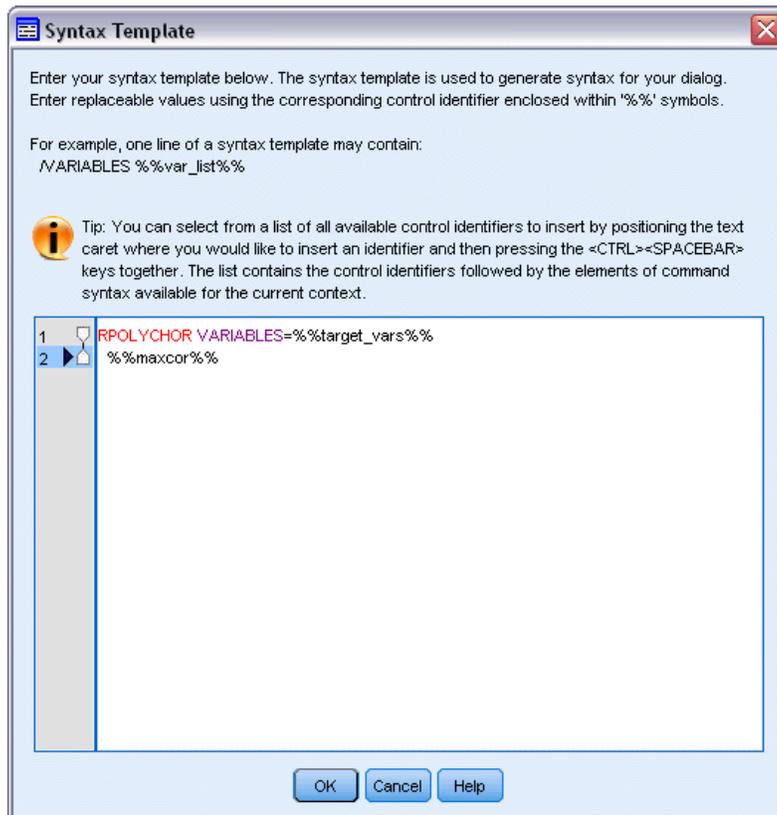
▶ Enter RPOLYCHOR VARIABLES= and then press CTRL+SPACEBAR. This displays a list of identifiers of the controls that generate command syntax.

▶ Select target_vars to insert the identifier for the Target Variable List control into the syntax template.

At run time, the control identifier `%%target_vars%%` will be replaced with the syntax generated by the Target Variable List control, which was specified to be the list of variables transferred to the control. *Note*: You can always enter control identifiers manually, but remember to include the double percent signs (`%%`) before and after the identifier.

▶ On a new line, insert the identifier `%%maxcor%%` for the *maxcor* parameter.

At run time, the identifier `%%maxcor%%` will be replaced by:

```
/OPTIONS MAXCOR=<value of Number control>
```

However, if the value of the Number control is empty, then `%%maxcor%%` will be replaced with the empty string.

For example, the following user choices generate the displayed syntax:

■ The variables *var1* and *var2* are transferred to the Target Variable List control.

■ The value of the *maxcor* control is specified as 0.999.

```
RPOLYCHOR VARIABLES=var1 var2
   /OPTIONS MAXCOR=0.999.
```

▶ Click OK in the Syntax Template dialog box to save the template. You can always modify it later.

Deploying a Custom Dialog

To use a custom dialog for an extension command, the specifications for the dialog and the files defining the extension command (the XML specification file and the file(s) containing the implementation code) must be installed on each machine that will use the dialog.

To install the custom dialog on your machine:

▶ From the menus in the Custom Dialog Builder choose:
File > Install

Installing a dialog for the first time requires a restart of IBM® SPSS® Statistics in order to add the menu item for the dialog. Once the application is restarted, you should see the new menu item for your dialog.

To share your custom dialog with other users of your extension command you first save the dialog specifications to a custom dialog package (*.spd*) file.

▶ From the menus in the Custom Dialog Builder choose:
File > Save

▶ In the Save a Dialog Specification dialog box, specify a location for the custom dialog package file. By default, the name of the file is the value of the Dialog Name property.

For users with PASW Statistics 17:

► Distribute the custom dialog package file, along with the files for the extension command, to the desired end users.

End users can easily install the dialog from the custom dialog package file:

► From the menus, they will choose:
Utilities > Install Custom Dialog

► In the Open a Dialog Specification dialog box, they will then select the custom dialog package file you distributed.

► Instruct your end users to manually deploy the files associated with the extension command. For more information, see the topic "Deploying an Extension Command" on p. 386.

A custom dialog package file named *RPOLYCHOR.spd* containing the specifications for the Polychoric Correlation dialog described here is included with the accompanying examples in the */examples/extensions* folder.

For users with PASW Statistics 18 or higher :

► Create an extension bundle containing the specifications for the custom dialog and the files associated with the extension command.

End users simply need to install the extension bundle to install both the custom dialog and the extension command files.

► From the menus, they will choose:
Utilities > Extension Bundles > Install Extension Bundle

► In the Open an Extension Bundle dialog box, they will then select the extension bundle file you distributed.

Creating an Extension Bundle

To create an extension bundle for the RPOLYCHOR extension command and its custom dialog:

► From the menus choose:
Utilities > Extension Bundles > Create Extension Bundle

Figure 31-9
Create Extension Bundle dialog box, Required tab
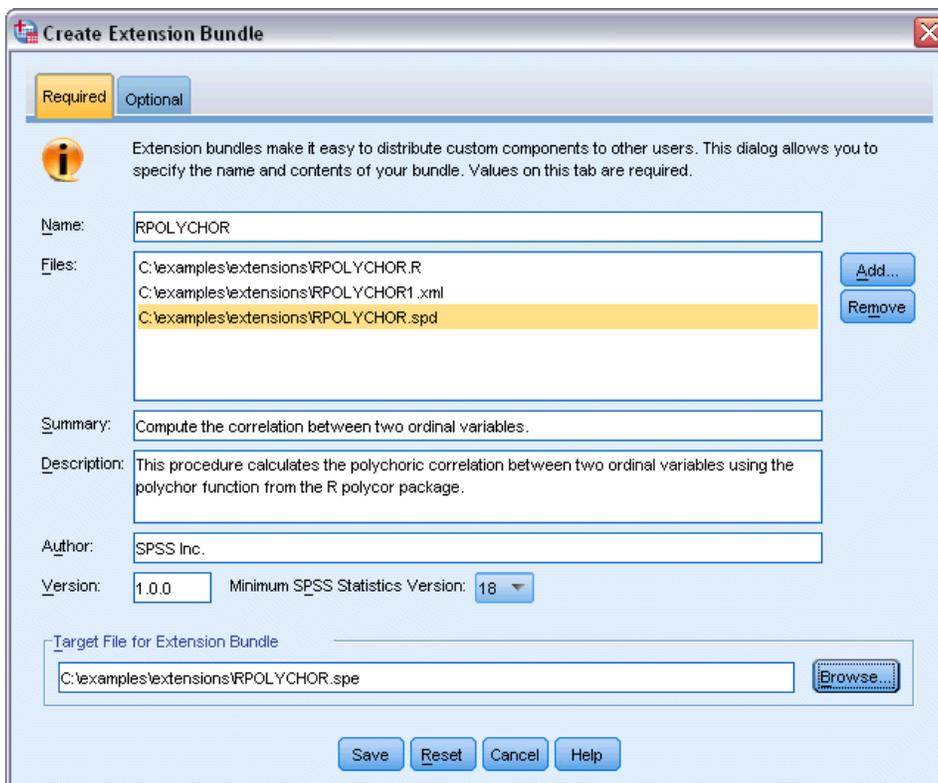
The Required tab of the Create Extension Bundle dialog box contains the fields that are required for an extension bundle.

▶ Enter RPOLYCHOR in the Name field. The recommended convention is to use the same name as the extension command or custom dialog you're packaging.

▶ Click Add to add the files *RPOLYCHOR.R*, *RPOLYCHOR1.xml*, and *RPOLYCHOR.spd*, from the */examples/extensions* folder of the accompanying examples.

▶ Enter a value in the Summary field. This should be a short description, intended to be displayed as a single line.

▶ Enter a value in the Description field. This should provide a more detailed description than the summary. If the extension bundle provides a wrapper for a function from an R package, then that should be mentioned here.

▶ Enter a value in the Author field. For your own extension bundles, you may wish to provide an email address here.

▶ Enter a value in the Version field. The version should have the form x.x.x, where each component is an integer. Zeros are implied for missing components if less than three are specified.

► Click Browse, navigate to the location where you want to save the extension bundle, and enter a file name.

The recommended convention for the name of the file containing an extension bundle is the name of the associated extension command or custom dialog, with any spaces replaced by underscores. Extension bundles have a file type of *spe*.

► Click on the Optional tab.

Figure 31-10
Create Extension Bundle dialog box, Optional tab

The Optional tab of the Create Extension Bundle dialog box contains the optional fields for an extension bundle.

The Links field allows you to specify a set of URL's to associate with the extension bundle—for example, your home page.

The Categories field allows you to specify a set of terms to associate with the extension bundle when authoring an extension bundle for posting to Developer Central. Enter one or more of the values.

► Check the Integration Plug-In for R required box. The RPOLYCHOR command requires the IBM® SPSS® Statistics - Integration Plug-In for R since it is implemented in R. Users will be alerted at install time if they don't have a required Plug-in.

▶ Click anywhere in the Required R Packages control to highlight the entry field and enter polycor.

When the extension bundle is installed, IBM® SPSS® Statistics will check if the required R packages exist on the end user's machine and attempt to download and install any that are missing.

▶ Click Save

This saves the extension bundle to the specified file and closes the Create Extension Bundle dialog box. You can then distribute the extension bundle (*spe*) file to your end users. An extension bundle file named *RPOLYCHOR.spe* is included with the accompanying examples in the */examples/extensions* folder.

Detailed help on creating and installing extension bundles is available in the SPSS Statistics Help system. You may also want to consult the tutorials available from Help>Working with R.

IBM SPSS Statistics for SAS Programmers

This chapter shows the IBM® SPSS® Statistics code and SAS equivalents for a number of basic data management tasks. It is not a comprehensive comparison of the two applications. The purpose of this chapter is to provide a point of reference for users familiar with SAS who are making the transition to SPSS Statistics; it is not intended to demonstrate how one application is better or worse than the other.

Reading Data

Both IBM® SPSS® Statistics and SAS can read data stored in a wide variety of formats, including numerous database formats, Excel spreadsheets, and text files. All of the SPSS Statistics examples presented in this section are discussed in greater detail in Chapter 3.

Reading Database Tables

Both SAS and IBM® SPSS® Statistics rely on Open Database Connectivity (ODBC) to read data from relational databases. Both applications read data from databases by reading database tables. You can read information from a single table or merge data from multiple tables in the same database.

Reading a Single Database Table

The structure of a database table is very similar to the structure of a data file in IBM® SPSS® Statistics format or an SAS dataset: records (rows) are cases, and fields (columns) are variables.

```
access1.sps.
GET DATA /TYPE=ODBC /CONNECT=
 'DSN=MS Access Database;DBQ=/examples/data/dm_demo.mdb;'+
 'DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
 /SQL = 'SELECT * FROM CombinedTable'.
EXECUTE.
```

```
proc sql;
connect to odbc(dsn=dm_demo uid=admin pwd=admin);
create table sasdata1 as
  select *
  from connection to odbc(
  select *
  from CombinedTable
  );
quit;
```

- The SPSS Statistics code allows you to input the parameters for the name of the database and the path directly into the code. SAS assumes that you have used the Windows Administrative Tools to set up the ODBC path. For this example, SAS assumes that the ODBC DSN for the database */examples/data/dm_demo.mdb* is defined as *dm_demo*.

- Another difference you will notice is that SPSS Statistics does not use a dataset name. This is because once the data is read, it is immediately the active dataset in SPSS Statistics. For this example, the SAS dataset is given the name *sasdata1*.

- In SPSS Statistics, the CONNECT string and all SQL statements must be enclosed in quotes.

- SAS converts the spaces in field names to underscores in variable names, while SPSS Statistics removes the spaces without substituting any characters. Where SAS uses all of the original variable names as labels, SPSS Statistics provides labels for only the variables not conforming to SPSS Statistics standards. So, in this example, the variable *ID* will be named *ID* in SPSS Statistics with no label and will be named *ID* in SAS with a label of ID. The variable *Marital Status* will be named *Marital_Status* in SAS and *MaritalStatus* in SPSS Statistics, with a label of *Marital Status* in both SPSS Statistics and SAS.

Reading Multiple Tables

Both IBM® SPSS® Statistics and SAS support reading and merging multiple database tables, and the code in both languages is very similar.

```
*access_multtables1.sps.
GET DATA /TYPE=ODBC /CONNECT=
 'DSN=MS Access Database;DBQ=/examples/data/dm_demo.mdb;'+
  'DriverId=25;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;'
 /SQL =
 'SELECT  * FROM DemographicInformation, SurveyResponses'
 ' WHERE DemographicInformation.ID=SurveyResponses.ID'.
EXECUTE.


proc sql;
connect to odbc(dsn=dm_demo uid=admin pwd=admin);
create table sasdata2 as
  select *
  from connection to odbc(
  select *
  from DemographicInformation, SurveyResponses
  where DemographicInformation.ID=SurveyResponses.ID
  );
quit;
```

Both languages also support left and right outer joins and one-to-many record matching between database tables.

```
*sqlserver_outer_join.sps.
GET DATA /TYPE=ODBC
 /CONNECT= 'DSN=SQLServer;UID=;APP=SPSS For Windows;'
  'WSID=ROLIVERLAP;Network=DBMSSOCN;Trusted_Connection=Yes'
 /SQL =
 'SELECT  SurveyResponses.ID,  SurveyResponses.Internet,'
 ' [Value Labels].[Internet Label]'
 ' FROM SurveyResponses LEFT OUTER JOIN [Value Labels]'
 ' ON SurveyResponses.Internet'
 ' = [Value Labels].[Internet Value]'.
```

```
proc sql;
connect to odbc(dsn=sql_survey uid=admin pwd=admin);
create table sasdata3 as
  select *
  from connection to odbc(
  select SurveyResponses.ID,
      SurveyResponses.Internet,
      "Value Labels"."Internet Label"
  from SurveyReponses left join "Value Labels"
      on SurveyReponses.Internet =
      "Value Labels"."Internet Value"
      );
quit;
```

The left outer join works similarly for both languages.

- The resulting dataset will contain all of the records from the *SurveyResponses* table, even if there is not a matching record in the *Value Labels* table.

- SPSS Statistics requires the syntax LEFT OUTER JOIN and SAS requires the syntax left join to perform the join.

- Both languages support the use of either quotes or square brackets to delimit table and/or variable names that contain spaces. Since SPSS Statistics requires that each line of SQL be quoted, square brackets are used here for clarity.

Reading Excel Files

IBM® SPSS® Statistics and SAS can read individual Excel worksheets and multiple worksheets in the same Excel workbook.

Reading a Single Worksheet

As with reading a single database table, the basic mechanics of reading a single worksheet are fairly simple: rows are read as cases, and columns are read as variables.

```
*readexcel.sps.
GET DATA
   /TYPE=XLS
   /FILE='/examples/data/sales.xls'
   /SHEET=NAME 'Gross Revenue'
   /CELLRANGE=RANGE 'A2:I15'
   /READNAMES=on .

proc import datafile='/examples/data/sales.xls'
  dbms=excel2000 replace out=SASdata4;
  sheet="Gross Revenue";
  range="A2:I15";
  getnames=yes;
  run;
```

- Both languages require the Excel filename, worksheet name, and cell range.

- Both provide the choice of reading the top row of the range as variable names. IBM® SPSS® Statistics accomplishes this with the READNAMES subcommand, and SAS accomplishes this with the getnames option.

- SAS requires an output dataset name. The dataset name *SASdata4* has been used in this example. SPSS Statistics has no corresponding requirement.

- Both languages convert spaces in variable names to underscores. SAS uses all of the original variable names as labels, and SPSS Statistics provides labels for the variable names not conforming to SPSS Statistics variable naming rules. In this example, both languages convert *Store Number* to *Store_Number* with a label of *Store Number*.

- The two languages use different rules for assigning the variable type (for example, numeric, string, or date). SPSS Statistics searches the entire column to determine each variable type. SAS searches to the first nonmissing value of each variable to determine the type. In this example, the *Toys* variable contains dollar-formatted data with the exception of one record containing a value of "NA." SPSS Statistics assigns this variable the string data type, preserving the "NA" in record five, whereas SAS assigns it a numeric dollar format and sets the value for *Toys* in record five to missing.

Reading Multiple Worksheets

Both IBM® SPSS® Statistics and SAS rely on ODBC to read multiple worksheets from a workbook.

```
*readexcel2.sps.
GET DATA
  /TYPE=ODBC
  /CONNECT=
    'DSN=Excel Files;DBQ=c:\examples\data\sales.xls;' +
    'DriverId=790;MaxBufferSize=2048;PageTimeout=5;'
    /SQL =
    'SELECT  Location$.[Store Number], State, Region, City,'
    ' Power, Hand, Accessories,'
    ' Tires, Batteries, Gizmos, Dohickeys'
    ' FROM [Location$], [Tools$], [Auto$]'
    ' WHERE [Tools$].[Store Number]=[Location$].[Store Number]'
    ' AND [Auto$].[Store Number]=[Location$].[Store Number]'.

proc sql;
connect to odbc(dsn=salesxls uid=admin pwd=admin);
create table sasdata5 as
  select *
  from connection to odbc(
  select Location$."Store Number", State, Region, City,
    Power, Hand, Accessories, Tires, Batteries, Gizmos,
    Dohickeys
  from "Location$", "Tools$", "Auto$"
  where "Tools$"."Store Number"="Location$"."Store Number"
  and "Auto$"."Store Number"="Location$"."Store Number"
  );
quit;;
```

- For this example, both SPSS Statistics and SAS treat the worksheet names as table names in the `From` statement.

- Both require the inclusion of a "$" after the worksheet name.

- As in the previous ODBC examples, quotes could be substituted for the square brackets in the SPSS Statistics code and vice versa for the SAS code.

Reading Text Data

Both IBM® SPSS® Statistics and SAS can read a wide variety of text-format data files. This example shows how the two applications read comma-separated value (CSV) files. A CSV file uses commas to separate data values and encloses values that include commas in quotation marks. Many applications export text data in this format.

```
ID,Name,Gender,Date Hired,Department
1,"Foster, Chantal",f,10/29/1998,1
2,"Healy, Jonathan",m,3/1/1992,3
3,"Walter, Wendy",f,1/23/1995,2
```

```
*delimited_csv.sps.
GET DATA  /TYPE = TXT
  /FILE = '/examples/data/CSV_file.csv'
  /DELIMITERS = ","
  /QUALIFIER = '"'
  /ARRANGEMENT = DELIMITED
  /FIRSTCASE = 2
  /VARIABLES = ID F3 Name A15 Gender A1
   Date_Hired ADATE10 Department F1.
```

```
data csvnew;
  infile "/examples/data/csv_file.csv" DLM=',' Firstobs=2 DSD;
  informat name $char15. gender $1. date_hired mmddyy10.;
  input id name gender date_hired department;
  run;
```

- The SPSS Statistics `DELIMITERS` and SAS `DLM` values identify the comma as the delimiter.
- SAS uses the `DSD` option on the `infile` statement to handle the commas within quoted values, and SPSS Statistics uses the `QUALIFIER` subcommand.
- SPSS Statistics uses the format `ADATE10`, and SAS uses the format `mmddyy10` to properly read the date variable.
- The SPSS Statistics `FIRSTCASE` subcommand is equivalent to the SAS `Firstobs` specification, indicating that the data to be read start on the second line, or record.

Merging Data Files

Both IBM® SPSS® Statistics and SAS can merge two or more datasets together.

Merging Files with the Same Cases but Different Variables

One of the types of merges supported by both applications is a **match merge**: two or more datasets that contain the same cases but different variables are merged together. Records from each dataset are matched based on the values of one or more key variables. For example, demographic data for survey respondents might be contained in one dataset, and survey responses for surveys taken at different times might be contained in multiple additional datasets. The cases are the same (respondents), but the variables are different (demographic information and survey responses).

```
GET FILE='/examples/data/match_response1.sav'.
SORT CASES BY id.
DATASET NAME response1
```

```
GET FILE='/examples/data/match_response2.sav'.
SORT CASES BY id.
DATASET NAME response2.
GET FILE='/examples/data/match_demographics.sav'.
SORT CASES BY id.
MATCH FILES /FILE=*
  /FILE='response1'
  /FILE='response2'
  /RENAME  opinion1=opinion1_2 opinion2=opinion2_2
   opinion3=opinion3_2 opinion4=opinion4_2
  /BY id.
EXECUTE.


libname in "/examples/data";
proc sort data=in.match_response1;
  by id;
  run;
proc sort data=in.match_response2;
  by id;
  run;
proc sort data=in.match_demographics;
  by id;
  run;
data match_new;
  merge match_demographics
        match_response1
        match_response2 (rename=(opinion1=opinion1_2
          opinion2=opinion2_2 opinion3=opinion3_2
          opinion4=opinion4_2));
  by id;
  run;
```

- IBM® SPSS® Statistics uses the GET FILE command to open each data file prior to sorting. SAS uses libname to assign a working directory for each dataset that needs sorting.

- Both require that each dataset be sorted by values of the BY variable used to match cases.

- In SPSS Statistics, the last data file opened with the GET FILE command is the active data file. So, in the MATCH FILES command, FILE=* refers to the data file *match_demographics.sav*, and the merged working data file retains that filename. If you do not explicitly save the file with the same filename, the original file is not overwritten. SAS requires a dataset name for the data step. In this example, the merged dataset is given the name *match_new*.

- Both SPSS Statistics and SAS allow you to rename variables when merging. This is necessary because *match_response1* and *match_response2* contain variables with the same names. If the variables were not renamed for the second dataset, then the variables merged from the first dataset would be overwritten.

The SPSS Statistics example presented in this section is discussed in greater detail in "Merging Files with the Same Cases but Different Variables" on p. 58.

Merging Files with the Same Variables but Different Cases

You can also merge two or more datasets that contain the same variables but different cases, appending cases from each dataset. For example, regional revenue for two different company divisions might be stored in two separate datasets. Both files have the same variables (region indicator and revenue) but different cases (each region for each division is a case).

```
*add_files1.sps.
```

```
ADD FILES
  /FILE = '/examples/data/catalog.sav'
  /FILE ='/examples/data/retail.sav'
    /IN = Division.
EXECUTE.
VALUE LABELS Division 0 'Catalog' 1 'Retail Store'.

libname in "/examples/data";
proc format;
  value divfmt
       0='Catalog'
       1='Retail Store' ;
  run;
data append_new;
  set in.catalog (in=a) in.retail (in=b);
  format division divfmt.;
  if a then division=0;
     else if b then division=1;
  run;
```

■ In the IBM® SPSS® Statistics code, the `IN` subcommand after the second `FILE` subcommand creates a new variable, *Division*, with a value of 1 for cases from *retail.sav* and a value of 0 for cases from *catalog.sav*. To achieve this same result, SAS requires the `format` procedure to create a user-defined format where 0 represents the catalog file and 1 represents the retail file.

■ In SAS, the `set` statement is required to append the files so that the system variable *in* can be used in the data step to assist with identifying which dataset contains each observation.

■ The SPSS Statistics `VALUE LABELS` command assigns descriptive labels to the values 0 and 1 for the variable *Division*, making it easier to interpret the values of the variable that identifies the source file for each case. In SAS, this would require a separate formats file.

The SPSS Statistics example presented in this section is discussed in greater detail in "Merging Files with the Same Variables but Different Cases" on p. 61.

Performing General Match Merging

In addition to the simple match merge discussed in "Merging Files with the Same Cases but Different Variables" on p. 413, both applications can handle more complex examples of match merging. For instance, you want to merge two datasets, keeping all records from the first one and only those from the second one that match on the key variable. However, the key variable in the second dataset presents the following complexities: its values are a transformation of the values of the key variable in the first dataset; it contains duplicate values and you only want to merge values from one of the duplicate records.

As an example, consider data from a study on new relaxation therapies for reducing blood pressure. Blood pressure readings are taken after each of several treatments and recorded in a master file that includes all readings for all participants in the study. A number of patients from a particular medical group are enrolled in the study, and the medical group would like to merge the final readings from the study with their patient's records. This requires merging only those records from the master file that correspond to patients from the medical group and keeping only the most recent record for each such patient. For privacy purposes, patients in the study are identified by the last five digits of their social security number, whereas the records maintained by the medical group use the full social security number as the patient identifier.

```
*python_dataset_mergeds2.sps.
GET FILE='/examples/data/merge_study.sav'.
SORT CASES BY id date (D).
DATASET NAME study.
GET FILE='/examples/data/merge_patients.sav'.
DATASET NAME patients.
BEGIN PROGRAM.
import spss
spss.StartDataStep()
ds1 = spss.Dataset(name='patients')
ds1.varlist.append('bps_study')
ds1.varlist.append('bpd_study')
ds2 = spss.Dataset(name='study')
id2vals = [item[0] for item in ds2.cases[0:len(ds2.cases),
                                          ds2.varlist['id'].index]]
for i in range(len(ds1.cases)):
   try:
      id1 = ds1.cases[i,ds1.varlist['id'].index][0]
      rownum=id2vals.index(id1[-ds2.varlist['id'].type:])
      ds1.cases[i,-2]=ds2.cases[rownum,ds2.varlist['sys'].index][0]
      ds1.cases[i,-1]=ds2.cases[rownum,ds2.varlist['dia'].index][0]
   except:
      pass
spss.EndDataStep()
END PROGRAM.

libname in "/examples/data";
data _null_;
  set in.merge_study;
  call symput('id_len',length(id));
  run;
data temp;
  set in.merge_patients;
  tempid = substr(id,length(id)+1-symget('id_len'),symget('id_len'));
  run;
proc sort data=temp;
  by tempid;
  run;
proc sort data=in.merge_study;
  by id date;
  run;
data merge_new;
  merge temp(in=c)
        in.merge_study(drop=date
           rename=(id=tempid sys=bps_study dia=bpd_study));
  by tempid;
  if c & last.tempid;
  drop tempid;
  run;
```

- To perform more general match merging in IBM® SPSS® Statistics than is possible with the MATCH FILES command, you initiate a data step. Data steps in SPSS Statistics offer similar functionality to what is available with the SAS DATA step. They are initiated with the spss.StartDataStep() function from within a BEGIN PROGRAM–END PROGRAM block and require the SPSS Statistics-Python Integration Plug-In. Statements within a BEGIN PROGRAM–END PROGRAM block are written in the Python programming language.

- Once a data step has been initiated in SPSS Statistics, you can access any open dataset and create new datasets using the Dataset class—a Python class provided with the plug-in. Each instance of the class provides access to the cases and variables in a particular dataset. In this example, an instance of the Dataset class is created for the datasets *patients* and *study*, as in spss.Dataset(name='patients') and spss.Dataset(name='study').

- The `Dataset` class does not require that the data are sorted. In this example, for SPSS Statistics, it is convenient to sort the data from the study by the key variable *id* and in descending order by *date*. This simplifies the task of extracting the most recent record for a given patient. For SAS, both datasets are sorted before performing the merge with the `MERGE` statement.

- As with the `MATCH FILES` command, merging data with the `Dataset` class does not require the creation of a new dataset. In this example, data from the study will be merged to the existing dataset *patients*.

- In a SAS DATA step, you use specific syntax statements such as `INPUT`, `SET`, and `MERGE` to accomplish your goals. There are no equivalents to these statements for a data step in SPSS Statistics. SPSS Statistics data steps are written in the Python programming language and utilize a set of SPSS Statistics-specific classes—such as the `Dataset` class—and functions to accomplish data management tasks. For more information, see the topic "Creating and Accessing Multiple Datasets" in Chapter 16 on p. 261.

- For a data step in SPSS Statistics, there is no equivalent to the SAS `RUN` statement. A data step in SPSS Statistics is executed along with the `BEGIN PROGRAM-END PROGRAM` block that contains it.

Aggregating Data

IBM® SPSS® Statistics and SAS can both aggregate groups of cases, creating a new dataset in which the groups are the cases. In this example, information was collected for every person living in a selected sample of households. In addition to information for each individual, each case contains a variable that identifies the household. You can change the unit of analysis from individuals to households by aggregating the data based on the value of the household ID variable.

```
*aggregate2.sps.
DATA LIST FREE (" ")
  /ID_household (F3) ID_person (F2) Income (F8).
BEGIN DATA
101 1 12345 101 2 47321 101 3 500 101 4 0
102 1 77233 102 2 0
103 1 19010 103 2 98277 103 3 0
104 1 101244
END DATA.
AGGREGATE
  /OUTFILE = * MODE = ADDVARIABLES
  /BREAK = ID_household
  /per_capita_Income = MEAN(Income)
  /Household_Size = N.

data tempdata;
  informat id_household 3. id_person 2. income 8.;
  input ID_household ID_person Income @@;
cards;
101 1 12345 101 2 47321 101 3 500 101 4 0
102 1 77233 102 2 0
103 1 19010 103 2 98277 103 3 0
104 1 101244
;
run;
proc sort data=tempdata;
  by ID_household;
  run;
proc summary data=tempdata;
```

```
      var Income;
    by ID_household;
    output out=aggdata
      mean=per_capita_Income
      n=Household_Size;
    run;
  data new;
    merge tempdata aggdata (drop=_type_ _freq_);
    by ID_Household;
    run;
```

■ SAS uses the summary procedure for aggregating, whereas SPSS Statistics has a specific command for aggregating data: AGGREGATE.

■ The SPSS Statistics BREAK subcommand is equivalent to the SAS By Variable command.

■ In SPSS Statistics, you specify the aggregate summary function and the variable to aggregate in a single step, as in per_capita_Income = MEAN(Income). In SAS, this requires two separate statements: var Income and mean=per_capita_Income.

■ To append the aggregated values to the original data file, SPSS Statistics uses the subcommand /OUTFILE = * MODE = ADDVARIABLES. With SAS, you need to merge the original and aggregated datasets, and the aggregated dataset contains two automatically generated variables that you probably don't want to include in the merged results. The SAS merge command contains a specification to delete these extraneous variables.

Assigning Variable Properties

In addition to the basic data type (numeric, string, date, and so on), you can assign other properties that describe the variables and their associated values. In a sense, these properties can be considered **metadata**—data that describe the data. The IBM® SPSS® Statistics examples provided here are discussed in greater detail in "Variable Properties" on p. 77.

Variable Labels

Both IBM® SPSS® Statistics and SAS provide the ability to assign descriptive variable labels that have less restrictive rules than variable naming rules. For example, variable labels can contain spaces and special characters not allowed in variable names.

```
VARIABLE LABELS
  Interview_date "Interview date"
  Income_category "Income category"
  opinion1 "Would buy this product"
  opinion2 "Would recommend this product to others"
  opinion3 "Price is reasonable"
  opinion4 "Better than a poke in the eye with a sharp stick".


label Interview_date = "Interview date"
  Income_category = "Income category"
  opinion1="Would buy this product"
  opinion2="Would recommend this product to others"
  opinion3="Price is reasonable"
  opinion4="Better than a poke in the eye with a sharp stick";
```

- In SPSS Statistics, you define variable labels with the VARIABLE LABELS command. In SAS, you use the label statement.

- In SPSS Statistics, VARIABLE LABELS commands can appear anywhere in the command stream, and the labels are attached to the variables at that point in the command processing. So you can assign labels to newly created variables and/or change labels for existing variables at any time. In SAS, label statements can be contained in a data step or a proc step. When used in a data step, the labels are permanently associated with the specified variables. When used in a proc step, the association is temporary.

Value Labels

You can also assign descriptive labels for each value of a variable. This is particularly useful if your data file uses numeric codes to represent non-numeric categories. For example, *income_category* uses the codes 1 through 4 to represent different income ranges, and the four opinion variables use the codes 1 through 5 to represent levels of agreement/disagreement.

```
VALUE LABELS
  Gender "m" "Male" "f" "Female"
  /Income_category 1 "Under 25K" 2 "25K to 49K"
   3 "50K to 74K" 4 "75K+" 7 "Refused to answer"
   8 "Don't know" 9 "No answer"
  /Religion 1 "Catholic" 2 "Protestant" 3 "Jewish"
   4 "Other" 9 "No answer"
  /opinion1 TO opinion4 1 "Strongly Disagree" 2 "Disagree"
   3 "Ambivalent" 4 "Agree" 5 "Strongly Agree" 9 "No answer".
```

```
proc format;
  value $genfmt
    'm'='Male'
         'f'='Female'
         ;
  value incfmt
    1='Under 25K'
    2='25K to 49K'
    4='75K+'     3='50K to 74K'
    7='Refused to answer'
    8='Don''t know'
    9='No answer'
    ;
  value relfmt
    1='Catholic'
    2='Protestant'
    3='Jewish'
    4='Other'
    9='No answer'
    ;
  value opnfmt
    1='Strongly Disagree'
    2='Disagree'
    3='Ambivalent'
    4='Agree'
    5='Strongly Agree'
    9='No answer'
    ;
  run;
data new;
  format Gender $genfmt.
    Income_category incfmt.
    Religion relfmt.
```

```
     opinion1 opinion2 opinion3 opinion4 opnfmt.;
  input Gender $ Income_category Religion opinion1-opinion4;
cards;
m 3 4 5 1 3 1
f 3 0 2 3 4 3
;
run;
```

- In IBM® SPSS® Statistics, assigning value labels is relatively straightforward. You can insert `VALUE LABELS` commands (and `ADD VALUE LABELS` commands to append additional value labels) at any point in the command stream; those value labels, like variable labels, become metadata that is part of the data file and saved with the data file.

- In SAS, you can define a format and then apply the format to specified variables within a data step.

Cleaning and Validating Data

Real data frequently contain real errors, and IBM® SPSS® Statistics and SAS both have features that can help identify invalid or suspicious values. All of the SPSS Statistics examples provided in this section are discussed in detail.

Finding and Displaying Invalid Values

All of the variables in a file may have values that appear to be valid when examined individually, but certain combinations of values for different variables may indicate that at least one of the variables has either an invalid value or at least one that is suspect. For example, a pregnant male clearly indicates an error in one of the values, whereas a pregnant female older than 55 may not be invalid but should probably be double-checked.

```
*invalid_data3.sps.
DATA LIST FREE /age gender pregnant.
BEGIN DATA
25 0 0
12 1 0
80 1 1
47 0 0
34 0 1
9 1 1
19 0 0
27 0 1
END DATA.
VALUE LABELS gender 0 'Male' 1 'Female'
  /pregnant 0 'No' 1 'Yes'.
DO IF pregnant = 1.
- DO IF gender = 0.
-   COMPUTE valueCheck = 1.
- ELSE IF gender = 1.
-   DO IF age > 55.
-     COMPUTE valueCheck = 2.
-   ELSE IF age < 12.
-     COMPUTE valueCheck = 3.
-   END IF.
- END IF.
ELSE.
- COMPUTE valueCheck=0.
END IF.
VALUE LABELS valueCheck
```

```
    0 'No problems detected'
    1 'Male and pregnant'
    2 'Age > 55 and pregnant'
    3 'Age < 12 and pregnant'.
FREQUENCIES VARIABLES = valueCheck.

proc format;
  value genfmt
    0='Male'
    1='Female'
    ;
  value pregfmt
    0='No'
    1='Yes'
    ;
  value vchkfmt
    0='No problems detected'
    1='Male and pregnant'
    2='Age > 55 and pregnant'
    3='Age < 12 and pregnant'
    ;
  run;
data new;
  format gender genfmt.
    pregnant pregfmt.
    valueCheck vchkfmt.
    ;
  input age gender pregnant;
  valueCheck=0;
  if pregnant then do;
    if gender=0 then valueCheck=1;
    else if gender then do;
      if age > 55 then valueCheck=2;
      else if age < 12 then valueCheck=3;
    end;
  end;
cards;
25 0 0
12 1 0
80 1 1
47 0 0
34 0 1
9 1 1
19 0 0
27 0 1
;
run;
proc freq data=new;
  tables valueCheck;
  run;
```

- `DO IF pregnant = 1` in IBM® SPSS® Statistics is equivalent to `if pregnant then do` in SAS. As in the SAS example, you could simplify the SPSS Statistics code to `DO IF pregnant`, since this resolves to Boolean *true* if the value of *pregnant* is 1.

- `END IF` in SPSS Statistics is equivalent to `end` in SAS in this example.

- To display a frequency table of *valueCheck*, SPSS Statistics uses a simple `FREQUENCIES` command, whereas in SAS you need to call a procedure separate from the data processing step.

Finding and Filtering Duplicates

In this example, each case is identified by two ID variables: *ID_house*, which identifies each household, and *ID_person*, which identifies each person within the household. If multiple cases have the same value for both variables, then they represent the same case. In this example, that is not necessarily a coding error, since the same person may have been interviewed on more than one occasion. The interview date is recorded in the variable *int_date*, and for cases that match on both ID variables, we want to ignore all but the most recent interview.

The IBM® SPSS® Statistics code used in this example was generated by pasting and editing command syntax generated by the Identify Duplicate Cases dialog box (Data menu > Identify Duplicate Cases).

```
* duplicates_filter.sps.
GET FILE='/examples/data/duplicates.sav'.
SORT CASES BY ID_house(A) ID_person(A) int_date(A) .
MATCH FILES  /FILE = *
  /BY ID_house ID_person /LAST = MostRecent .
FILTER BY MostRecent .
EXECUTE.

libname in "/examples/data";
proc sort data=in.duplicates;
  by ID_house ID_person int_date;
  run;
data new;
  set in.duplicates;
  by ID_house ID_person;
  if last.ID_person;
  run;
```

■ Like SAS, SPSS Statistics is able to identify the last record within each sorted group. In this example, both assign a value of 1 to the last record in each group and a value of 0 to all other records.

■ SAS uses the temporary variable *last.* to identify the last record in each group. This variable is available for each variable in the `by` statement following the `set` statement within the data step, but it is not saved to the dataset.

■ SPSS Statistics uses a `MATCH FILES` command with a `LAST` subcommand to create a new variable, *MostRecent*, that identifies the last case in each group. This is not a temporary variable, so it is available for future processing.

■ Where SAS uses an `if` statement to select the last case in each group, SPSS Statistics uses a `FILTER` command to filter out all but the last case in each group. The new SAS data step does not contain the duplicate records. SPSS Statistics retains the duplicates but does not include them in reports or analyses unless you turn off filtering (but you can use `SELECT IF` to delete rather than filter unselected cases). SPSS Statistics displays these records in the Data Editor with a slash through the row number.

Transforming Data Values

In both IBM® SPSS® Statistics and SAS, you can perform data transformations ranging from simple tasks, such as collapsing categories for reports, to more advanced tasks, such as creating new variables based on complex equations and conditional statements. All of the SPSS Statistics examples provided here are discussed in greater detail in "Data Transformations" on p. 85.

Recoding Data

There are many reasons why you might need or want to recode data. For example, questionnaires often use a combination of high-low and low-high rankings. For reporting and analysis purposes, however, you probably want these all coded in a consistent manner.

```
*recode.sps.
DATA LIST FREE /opinion1 opinion2.
BEGIN DATA
1 5
2 4
3 3
4 2
5 1
END DATA.
RECODE opinion2
 (1 = 5) (2 = 4) (4 = 2) (5 = 1)
 (ELSE = COPY)
 INTO opinion2_new.
EXECUTE.
VALUE LABELS opinion1 opinion2_new
  1 'Really bad' 2 'Bad' 3 'Blah'
  4 'Good' 5 'Terrific!'.

proc format;
  value opfmt
        1='Really bad'
        2='Bad'
        3='Blah'
        4='Good'
        5='Terrific!'
        ;
  value conv
        5='1'
        4='2'
        3='3'
        2='4'
        1='5';
  run;
data recode;
  input opinion1 opinion2;
  length opinion2_new 8;
  opinion2_new=input(put(opinion2,conv.),1.);

  format opinion1 opinion2_new opfmt.;
datalines;
1 5
2 4
3 3
4 2
5 1
;
run;
```

- IBM® SPSS® Statistics uses a single RECODE command to create a new variable, *opinion2_new*, with the recoded values of the original variable, *opinion2*.

- The recoding can be accomplished in SAS by defining a format—*conv* in the example—that specifies the recoding scheme and applying the format with the put function.

- ELSE = COPY in the SPSS Statistics RECODE command covers any values not explicitly specified and copies the original values to the new variable.

Binning Data

Creating a small number of discrete categories from a continuous scale variable is sometimes referred to as **binning** or **banding**. For example, you can bin salary data into a few salary range categories.

Although it is not difficult to write code in IBM® SPSS® Statistics or SAS to bin a scale variable into range categories, in SPSS Statistics we recommend that you use the Visual Binning dialog box, available on the Transform menu, because it can help you make the best recoding choices by showing the actual distribution of values and where your selected category boundaries occur in the distribution. It also provides a number of different binning methods and can automatically generate descriptive labels for the binned categories. The SPSS Statistics command syntax in this example was generated by the Visual Binning dialog box.

```
*visual_binning.sps.
****commands generated by visual binning dialog***.
RECODE salary
     ( MISSING = COPY )
     ( LO THRU 25000 =1 )
     ( LO THRU 50000 =2 )
     ( LO THRU 75000 =3 )
     ( LO THRU HI = 4 )
     ( ELSE = SYSMIS ) INTO salary_category.
VARIABLE LABELS salary_category 'Current Salary (Binned)'.
FORMAT salary_category (F5.0).
VALUE LABELS salary_category
        1   '<= $25,000'
        2   '$25,001 - $50,000'
        3   '$50,001 - $75,000'
        4   '$75,001+'
        0   'missing'.
MISSING VALUES salary_category ( 0 ).
VARIABLE LEVEL salary_category ( ORDINAL ).
EXECUTE.

libname in "/examples/data";
proc format;
  value salfmt
        1='<= $25,000'
        2='$25,001 - $50,000'
        3='$50,001 - $75,000'
        4='$75,001+'
        0='missing'
        ;
  run;
data recode;
  set in.employee_data;
  format salary_category salfmt.;
  label salary_category = "Current Salary (Binned)";
  select;
        when (0<salary<=25000) salary_category=1;
        when (25000<salary<=50000) salary_category=2;
        when (50000<salary<=75000) salary_category=3;
        when (salary>75000) salary_category=4;
        otherwise salary_category=salary;
  end;
run;
```

- The SPSS Statistics Visual Binning dialog box generates RECODE command syntax similar to the code in the previous recoding example. It can also automatically generate appropriate descriptive value labels (as in this example) for each binned category.

- The recoding is accomplished in SAS with a series of `when` statements in a `select` group.

- The SPSS Statistics RECODE command supports the keywords LO and HI to ensure that no values are left out of the binning scheme. In SAS, you can obtain similar functionality with the standard <, <=, >, and >= operators.

Numeric Functions

In addition to simple arithmetic operators (for example, +, -, /, *), you can transform data values in both IBM® SPSS® Statistics and SAS with a wide variety of functions, including arithmetic and statistical functions.

```
*numeric_functions.sps.
DATA LIST LIST (",") /var1 var2 var3 var4.
BEGIN DATA
1, , 3, 4
5, 6, 7, 8
9, , , 12
END DATA.
COMPUTE Square_Root = SQRT(var4).
COMPUTE Remainder = MOD(var4, 3).
COMPUTE Average = MEAN.3(var1, var2, var3, var4).
COMPUTE Valid_Values = NVALID(var1 TO var4).
COMPUTE Trunc_Mean = TRUNC(MEAN(var1 TO var4)).
EXECUTE.

data new;
  input var1-var4;
  Square_Root=sqrt(var4);
  Remainder=mod(var4,3);
  x=n(of var1-var4);
  if x >= 3 then Average=mean(of var1-var4);
  Valid_Values=x;
  Trunc_Mean=int(mean(of var1-var4));
cards;
1 . 3 4
5 6 7 8
9 . . 12
;
run;
```

- SPSS Statistics and SAS use the same function names for the square root (SQRT) and remainder (MOD) functions.

- SPSS Statistics allows you to specify the minimum number of nonmissing values required to calculate any numeric function. For example, MEAN.3 specifies that at least three of the variables (or other function arguments) must contain nonmissing values.

- In SAS, if you want to specify the minimum number of nonmissing arguments for a function calculation, you need to calculate the number of nonmissing values using the function n and then use this information in an if statement prior to calculating the function.

- The SPSS Statistics NVALID function returns the number of nonmissing values in an argument list.

- The SAS int function is the analogue of the SPSS Statistics TRUNC function.

Random Number Functions

Random value and distribution functions generate random values based on various distributions.

```
*random_functons.sps.
NEW FILE.
SET SEED 987987987.
*create 1,000 cases with random values.
INPUT PROGRAM.
- LOOP #I=1 TO 1000.
-    COMPUTE Uniform_Distribution = UNIFORM(100).
-    COMPUTE Normal_Distribution = RV.NORMAL(50,25).
-    COMPUTE Poisson_Distribution = RV.POISSON(50).
-    END CASE.
- END LOOP.
- END FILE.
END INPUT PROGRAM.
FREQUENCIES VARIABLES = ALL
  /HISTOGRAM /FORMAT = NOTABLE.

data new;
  seed=987987987;
  do i=1 to 1000;
     Uniform_Distribution=100*ranuni(seed);
     Normal_Distribution=50+25*rannor(seed);
     Poisson_Distribution=ranpoi(seed,50);
     output;
     end;
  run;
```

■ Both SAS and IBM® SPSS® Statistics allow you to set the seed to start the random number generation process.

■ Both languages allow you to generate random numbers using a wide variety of statistical distributions. This example generates 1,000 observations using the uniform distribution with a mean of 100, the normal distribution with a mean of 50 and standard deviation of 25, and the Poisson distribution with a mean of 50.

■ SPSS Statistics allows you to provide parameters for the distribution functions, such as the mean and standard deviation for the RV.NORMAL function.

■ SAS functions are generic and require that you use equations to modify the distributions.

■ SPSS Statistics does not require the seed as a parameter in the random number functions as does SAS.

String Concatenation

You can combine multiple string and/or numeric values to create new string values. For example, you can combine three numeric variables for area code, exchange, and number into one string variable for telephone number with dashes between the values.

```
*concat_string.sps.
DATA LIST FREE /tel1 tel2 tel3 (3F4).
BEGIN DATA
111 222 3333
222 333 4444
333 444 5555
555 666  707
END DATA.
STRING telephone (A12).
```

```
COMPUTE telephone =
  CONCAT((STRING(tel1, N3)), "-",
         (STRING(tel2, N3)), "-",
         (STRING(tel3, N4))).
EXECUTE.

data new;
   input tel1 4. tel2 4. tel3 4.;
   telephone=
       cat(put(tel1,z3.),"-",put(tel2,z3.),"-",put(tel3,z4.));
cards;
111 222 3333
222 333 4444
333 444 5555
555 666  707
;
run;
```

- IBM® SPSS® Statistics uses the CONCAT function to concatenate strings and SAS uses the cat function for concatenation.

- The SPSS Statistics STRING function converts a numeric value to a character value, like the SAS put function.

- The SPSS Statistics N format converts spaces to zeroes. The SAS z format adds leading zeroes to fill the specified width.

String Parsing

In addition to being able to combine strings, you can take them apart. For example, you can take apart a 12-character telephone number, recorded as a string (because of the embedded dashes), and create three new numeric variables for area code, exchange, and number.

```
DATA LIST FREE (",") /telephone (A16).
BEGIN DATA
111-222-3333
222 - 333 - 4444
 333-444-5555
444 - 555-6666
555-666-0707
END DATA.
COMPUTE tel1 =
  NUMBER(SUBSTR(telephone, 1, CHAR.INDEX(telephone, "-")-1), F5).
COMPUTE tel2 =
  NUMBER(SUBSTR(telephone, CHAR.INDEX(telephone, "-")+1,
  CHAR.RINDEX(telephone, "-")-(CHAR.INDEX(telephone, "-")+1)), F5).
COMPUTE tel3 =
  NUMBER(SUBSTR(telephone, CHAR.RINDEX(telephone, "-")+1), F5).
EXECUTE.
FORMATS tel1 tel2 (N3) tel3 (N4).


data new;
   input telephone $16.;
   format tel1 tel2 3. tel3 z4.;
   tel1=input(substr(compress(telephone,'- '),1,3),3.);
   tel2=input(substr(compress(telephone,'- '),4,3),3.);
   tel3=input(substr(compress(telephone,'- '),7,4),4.);

cards;
111-222-3333
222 - 333 - 4444
```

```
333-444-5555
444 - 555-6666
555-666-0707
;
run;
```

- IBM® SPSS® Statistics uses substring (`SUBSTR`) and index (`CHAR.INDEX`, `CHAR.RINDEX`) functions to search the string for specified characters and to extract the appropriate values.

- SAS allows you to name the characters to exclude from a variable using the `compress` function and then take a substring (`substr`) of the resulting value.

- The SPSS Statistics `N` format is comparable to the SAS `z` format. Both formats write leading zeros.

Working with Dates and Times

Dates and times come in a wide variety of formats, ranging from different display formats (for example, 10/28/1986 versus 28-OCT-1986) to separate entries for each component of a date or time (for example, a day variable, a month variable, and a year variable). Both IBM® SPSS® Statistics and SAS can handle date and times in a variety of formats, and both applications provide features for performing date/time calculations.

Calculating and Converting Date and Time Intervals

A common date calculation is the elapsed time between two dates and/or times. Assuming you have assigned the appropriate date, time, or date/time format to the variables, IBM® SPSS® Statistics and SAS can both perform this type of calculation.

```
*date_functions.sps.
DATA LIST FREE (",")
  /StartDate (ADATE12) EndDate (ADATE12)
   StartDateTime(DATETIME20) EndDateTime(DATETIME20)
   StartTime (TIME10) EndTime (TIME10).
BEGIN DATA
3/01/2003, 4/10/2003
01-MAR-2003 12:00, 02-MAR-2003 12:00
09:30, 10:15
END DATA.
COMPUTE days = CTIME.DAYS(EndDate-StartDate).
COMPUTE hours = CTIME.HOURS(EndDateTime-StartDateTime).
COMPUTE minutes = CTIME.MINUTES(EndTime-StartTime).
EXECUTE.

data new;
    infile cards dlm=',' n=3;
    input StartDate : MMDDYY10. EndDate : MMDDYY10.
      #2 StartDateTime : DATETIME17. EndDateTime : DATETIME17.
      #3 StartTime : TIME5. EndTime : TIME5.
      ;
      days=EndDate-StartDate;
      hours=intck("hour",StartDateTime,EndDateTime);
      minutes=intck("minute",StartTime,EndTime);
cards;
3/01/2003, 4/10/2003
01-MAR-2003 12:00, 02-MAR-2003 12:00
09:30, 10:15
;
```

```
run;
```

- SPSS Statistics stores all date and time values as a number of seconds, and subtracting one date or time value returns the difference in seconds. You can use CTIME functions to return the difference as number of days, hours, or minutes.

- In SAS, simple dates are stored as a number of days, but times and dates with a time component are stored as a number of seconds. Subtracting one simple date from another will return the difference as a number of days. Subtracting one date/time from another, however, will return the difference as a number of seconds. You can obtain the difference in some other time measurement unit by using the intck function.

Adding to or Subtracting from One Date to Find Another Date

Another common date/time calculation is adding or subtracting days (or hours, minutes, and so forth) from one date to obtain another date. For example, let's say prospective customers can use your product on a trial basis for 30 days, and you need to know when the trial period ends—just to make it interesting, if the trial period ends on a Saturday or Sunday, you want to extend it to the following Monday.

```
*date_functions2.sps.
DATA LIST FREE (" ") /StartDate (ADATE10).
BEGIN DATA
10/29/2003 10/30/2003
10/31/2003 11/1/2003
11/2/2003 11/4/2003
11/5/2003 11/6/2003
END DATA.
COMPUTE expdate = StartDate + TIME.DAYS(30).
FORMATS expdate (ADATE10).
***if expdate is Saturday or Sunday, make it Monday***.
DO IF (XDATE.WKDAY(expdate) = 1).
-  COMPUTE expdate = expdate + TIME.DAYS(1).
ELSE IF (XDATE.WKDAY(expdate) = 7).
-  COMPUTE expdate = expdate + TIME.DAYS(2).
END IF.
EXECUTE.

data new;
  format expdate date10.;
    input StartDate : MMDDYY10. @@ ;
    expdate=StartDate+30;;
  if weekday(expdate)=1 then expdate+1;
  else if weekday(expdate)=7 then expdate+2;
cards;
10/29/2003 10/30/2003
10/31/2003 11/1/2003
11/2/2003 11/4/2003
11/5/2003 11/6/2003
;
run;
```

- Since all IBM® SPSS® Statistics date values are stored as a number of seconds, you need to use the TIME.DAYS function to add or subtract days from a date value. In SAS, simple dates are stored as a number of days, so you do not need a special function to add or subtract days.

- The SPSS Statistics XDATE.WKDAY function is equivalent to the SAS weekday function, and both return a value of 1 for Sunday and 7 for Saturday.

Extracting Date and Time Information

A great deal of information can be extracted from date and time variables. For example, in addition to the day, month, and year, a date is associated with a specific day of the week, week of the year, and quarter.

```
*date_functions3.sps.
DATA LIST FREE (",")
  /StartDateTime (datetime25).
BEGIN DATA
29-OCT-2003 11:23:02
1 January 1998 1:45:01
21/6/2000 2:55:13
END DATA.
COMPUTE dateonly=XDATE.DATE(StartDateTime).
FORMATS dateonly(ADATE10).
COMPUTE hour=XDATE.HOUR(StartDateTime).
COMPUTE DayofWeek=XDATE.WKDAY(StartDateTime).
COMPUTE WeekofYear=XDATE.WEEK(StartDateTime).
COMPUTE quarter=XDATE.QUARTER(StartDateTime).
EXECUTE.

data new;
  format dateonly mmddyy10.;
     input StartDateTime & : DATETIME25. ;
     dateonly=datepart(StartDateTime);
     hour=hour(StartDateTime);

     DayofWeek=weekday(dateonly);
     quarter=qtr(dateonly);
cards;
29-OCT-2003 11:23:02
;
run;
```

- IBM® SPSS® Statistics uses one main function, XDATE, to extract the date, hour, weekday, week, and quarter from a datetime value.

- SAS uses separate functions to extract the date, hour, weekday, and quarter from a datetime value.

- The SPSS Statistics XDATE.DATE function is equivalent to the SAS datepart function. The SPSS Statistics XDATE.HOUR function is equivalent to the SAS hour function.

- SAS requires a simple date value (with no time component) to obtain weekday and quarter information, requiring an extra calculation, whereas SPSS Statistics can extract weekday and quarter directly from a datetime value.

Custom Functions, Job Flow Control, and Global Macro Variables

The purpose of this section is to introduce users familiar with SAS to capabilities available with the IBM® SPSS® Statistics - Integration Plug-In for Python that allow you to:

- Write custom functions as you would with %macro.

- Control job flow as you would with call execute.

- Create global macro variables as you would with symput.

- Pass values to programs as you would with sysparm.

The Integration Plug-In for Python works with release 14.0.1 or later and requires only the Core system. The IBM® SPSS® Statistics examples in this section assume some familiarity with Python and the way it can be used with command syntax. For more information, see the topic "Getting Started with Python Programming in IBM SPSS Statistics" in Chapter 12 on p. 175.

Creating Custom Functions

Both IBM® SPSS® Statistics and SAS allow you to encapsulate a set of commands in a named piece of code that is callable and accepts parameters that can be used to complete the command specifications. In SAS, this is done with `%macro`, and in SPSS Statistics, this is best done with a Python user-defined function. To demonstrate this functionality, consider creating a function that runs a `DESCRIPTIVES` command in SPSS Statistics or the `means` procedure in SAS on a single variable. The function has two arguments: the variable name and the dataset containing the variable.

```
def prodstats(dataset,product):
    spss.Submit(r"""
    GET FILE='%(dataset)s'.
    DESCRIPTIVES %(product)s.
    """ %locals())


libname mydata '/data';
%macro prodstats(dataset=, product=);
   proc means data=&dataset;
      var &product;
   run;
%mend prodstats;

%prodstats(dataset=mydata.sales, product=milk)
```

- The `def` statement signals the beginning of a Python user-defined function (the colon at the end of the `def` statement is required). From within a Python function, you can execute syntax commands using the `Submit` function from the `spss` module. The function accepts a quoted string representing a syntax command and submits the command text to SPSS Statistics for processing. In SAS, you simply include the desired commands in the macro definition.

- The argument *product* is used to specify the variable for the `DESCRIPTIVES` command in SPSS Statistics or the `means` procedure in SAS, and *dataset* specifies the dataset. The expressions `%(product)s` and `%(dataset)s` in the SPSS Statistics code specify to substitute a string representation of the value of *product* and the value of *dataset*, respectively. For more information, see the topic "Dynamically Specifying Command Syntax Using String Substitution" in Chapter 13 on p. 200.

- In SPSS Statistics, the `GET` command is used to retrieve the desired dataset. If you omit this command, the function will attempt to run a `DESCRIPTIVES` command on the active dataset.

- To run the SAS macro, you simply call it. In the case of SPSS Statistics, once you've created a Python user-defined function, you typically include it in a Python module on the Python search path. Let's say you include the `prodstats` function in a module named `myfuncs`. You would then call the function with code such as,

  ```
  myfuncs.prodstats("/data/sales.sav","milk")
  ```

assuming that you had first imported `myfuncs`. Note that since the Python function `prodstats` makes use of a function from the `spss` module, the module `myfuncs` would need to include the statement `import spss` prior to the function definition.

For more information on creating Python functions for use with SPSS Statistics, see "Creating User-Defined Functions in Python" on p. 203.

Job Flow Control

Both IBM® SPSS® Statistics and SAS allow you to control the flow of a job, conditionally executing selected commands. In SAS, you can conditionally execute commands with `call execute`. The equivalent in SPSS Statistics is to drive command syntax from Python using the `Submit` function from the `spss` module. Information needed to determine the flow is retrieved from SPSS Statistics into Python. As an example, consider the task of conditionally generating a report of bank customers with low balances only if there are such customers at the time the report is to be generated.

```
BEGIN PROGRAM.
import spss, spssdata
spss.Submit("GET FILE='/data/custbal.sav'.")
dataObj=spssdata.Spssdata(indexes=['acctbal'])
report=False
for row in dataObj:
   if row.acctbal<200:
      report=True
      break
dataObj.close()
if report:
   spss.Submit("""
   TEMPORARY.
   SELECT IF acctbal<200.
   SUMMARIZE
   /TABLES=custid custname acctbal
   /FORMAT=VALIDLIST NOCASENUM NOTOTAL
   /TITLE='Customers with Low Balances'.
   """)
END PROGRAM.
```

```
libname mydata '/data';
data lowbal;
   set mydata.custbal end=final;
   if acctbal<200 then
      do;
         n+1;
         output;
      end;
   if final and n then call execute
      ("
       proc print data=lowbal;
         var custid custname acctbal;
         title 'Customers with Low Balances';
       run;
      ");
run;
```

■ Both SPSS Statistics and SAS use a conditional expression to determine whether to generate the report. In the case of SPSS Statistics, this is a Python `if` statement, since the execution is being controlled from Python. In SPSS Statistics, the command syntax to run the report is

passed as an argument to the `Submit` function in the `spss` module. In SAS, the command to run the report is passed as an argument to the `call execute` function.

■ The SPSS Statistics code makes use of functions in the `spss` and `spssdata` modules, so an import statement is included for them. The `spssdata` module is a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-In for Python. It builds on the functionality available in the `spss` module to provide a number of features that simplify the task of working with case data. For more information, see the topic "Using the spssdata Module" in Chapter 15 on p. 245.

■ The SAS job reads through all records in *custbal* and writes those records that represent customers with a balance of less than 200 to the dataset *lowbal*. In contrast, the SPSS Statistics code does not create a separate dataset but simply filters the original dataset for customers with a balance less than 200. The filter is executed only if there is at least one such customer when the report needs to be run. To determine if any customers have a low balance, data for the single variable *acctbal* (from *custbal*) is read into Python one case at a time, using the `Spssdata` class from the `spssdata` module. If a case with a low balance is detected, the indicator variable *report* is set to *true*, the `break` statement terminates the loop used to read the data, and the job proceeds to generating the report.

Creating Global Macro Variables

Both IBM® SPSS® Statistics and SAS have the ability to create global macro variables. In SAS, this is done with `symput`, whereas in SPSS Statistics, this is done from Python using the `SetMacroValue` function in the `spss` module. As an example, consider sales data that has been pre-aggregated into a dataset—let's call it *regionsales*—that contains sales totals by region. We're interested in using these totals in a set of analyses and find it convenient to store them in a set of global variables whose names are the regions with a prefix of *region_*.

```
BEGIN PROGRAM.
import spss, spssdata
spss.Submit("GET FILE='/data/regionsales.sav'.")
dataObj=spssdata.Spssdata()
data=dataObj.fetchall()
dataObj.close()
for row in data:
   macroValue=row.total
   macroName="!region_" + row.region
   spss.SetMacroValue(macroName, macroValue)
END PROGRAM.

libname mydata '/data';
data _null_;
   set mydata.regionsales;
   call symput('region_'||region,trim(left(total)));
run;
```

■ The `SetMacroValue` function from the `spss` module takes a name and a value (string or numeric) and creates a macro of that name that expands to the specified value (a numeric value provided as an argument is converted to a string). The availability of this function from Python means that you have great flexibility in specifying the value of the macro. Although the `SetMacroValue` function is called from Python, it creates a macro that is then available to command syntax outside of a `BEGIN PROGRAM` block. The convention in SPSS

Statistics—followed in this example—is to prefix the name of a macro with the ! character, although this is not required.

- Both `SetMacroValue` and `symput` create a macro variable that resolves to a string value, even if the value passed to the function was numeric. In SAS, the string is right-aligned and may require trimming to remove excess blanks. This is provided by the combination of the `left` and `trim` functions. SPSS Statistics does not require this step.

- The SAS code utilizes a data step to read the *regionsales* dataset, but there is no need to create a resulting dataset, so `_null_` is used. Likewise, the SPSS Statistics version doesn't need to create a dataset. It uses the `spssdata` module to read the data in *regionsales* and create a separate SPSS Statistics macro for each case read. For more information on the `spssdata` module, see "Using the spssdata Module" on p. 245.

Setting Global Macro Variables to Values from the Environment

IBM® SPSS® Statistics and SAS both support obtaining values from the operating environment and storing them to global macro variables. In SAS, this is accomplished by using the `sysparm` option on the command line to pass a value to a program. The value is then available as the global macro variable `&sysparm`. In SPSS Statistics, you first set an operating system environment variable that you can then retrieve using the Python `os` module—a built-in module that is always available in Python. Values obtained from the environment can be, but need not be, typical ones, such as a user name. For example, you may have a financial analysis program that uses the current interest rate as an input to the analysis, and you'd like to pass the value of the rate to the program. In this example, we're imagining passing a rate that we've set to a value of 4.5.

```
BEGIN PROGRAM.
import spss,os
val = os.environ['rate']
spss.SetMacroValue("!rate",val)
END PROGRAM.

sas /Work/SAS/prog1.sas -sysparm 4.5
```

- In the SPSS Statistics version, you first include an `import` statement for the Python `os` module. To retrieve the value of a particular environment variable, simply specify its name in quotes, as in: `os.environ['rate']`.

- With SPSS Statistics, once you've retrieved the value of an environment variable, you can set it to a Python variable and use it like any other variable in a Python program. This allows you to control the flow of a command syntax job using values retrieved from the environment. And you can use the `SetMacroValue` function (discussed in the previous example) to create a macro that resolves to the retrieved value and can be used outside of a `BEGIN PROGRAM` block. In the current example, a macro named `!rate` is created from the value of an environment variable named *rate*.

Notices

Licensed Materials – Property of SPSS Inc., an IBM Company. © Copyright SPSS Inc. 1989, 2010.

Patent No. 7,023,453

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: SPSS INC., AN IBM COMPANY, PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. SPSS Inc. may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-SPSS and non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this SPSS Inc. product and use of those Web sites is at your own risk.

When you send information to IBM or SPSS, you grant IBM and SPSS a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Information concerning non-SPSS products was obtained from the suppliers of those products, their published announcements or other publicly available sources. SPSS has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-SPSS products. Questions on the capabilities of non-SPSS products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to SPSS Inc., for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. SPSS Inc., therefore, cannot guarantee or

imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. SPSS Inc. shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks of IBM Corporation, registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at *http://www.ibm.com/legal/copytrade.hmtl*.

SPSS is a trademark of SPSS Inc., an IBM Company, registered in many jurisdictions worldwide.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

This product uses WinWrap Basic, Copyright 1993-2007, Polar Engineering and Consulting, *http://www.winwrap.com*.

Other product and service names might be trademarks of IBM, SPSS, or other companies.

Adobe product screenshot(s) reprinted with permission from Adobe Systems Incorporated.

Microsoft product screenshot(s) reprinted with permission from Microsoft Corporation.

Index