SPSS Programming and Data Management, 4th Edition

A Guide for SPSS and SAS® Users

Raynald Levesque and SPSS Inc.

For more information about SPSS® software products, please visit our Web site at *http://www.spss.com* or contact:

SPSS Inc.
233 South Wacker Drive, 11th Floor
Chicago, IL 60606-6412
Tel: (312) 651-3000
Fax: (312) 651-3668

SPSS Programming and Data Management, 4th Edition: A Guide for SPSS and SAS Users

Printed in the United States of America.

1 2 3 4 5 6 7 8 9 0   10 09 08 07

ISBN-13: 978-1-56827-390-7
ISBN-10: 1-56827-390-8

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. SPSS 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 SPSS command syntax by itself and in combination with the Python® programming language.

Using This Book

The contents of this book and the accompanying CD are discussed in Chapter 1. In particular, see the section "Using This Book" if you plan to run the examples on the CD. The CD also contains additional command files, macros, and scripts that are mentioned but not discussed in the book and that can be useful for solving specific problems.

This edition has been updated to include numerous enhanced data management features introduced in SPSS 15.0. Many examples will work with earlier versions, but some examples rely on features not available prior to SPSS 15.0. Some of the Python examples require SPSS 15.0.1 or later.

For SAS Users

If you have more experience with SAS than with SPSS for data management, see Chapter 22 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 staff who have contributed examples here and in SPSS 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 "SPSS 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 SPSS 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

1 Overview 1

Using This Book 1
Documentation Resources 2

Part I: Data Management

2 Best Practices and Efficiency Tips 4

Working with Command Syntax 4
- Creating Command Syntax Files 4
- Running SPSS Commands 5
- Syntax Rules 6

Customizing the Programming Environment 7
- Displaying Commands in the Log 7
- Displaying the Status Bar in Command Syntax Windows 8

Protecting the Original Data 9
- Do Not Overwrite Original Variables 10
- Using Temporary Transformations 10
- Using Temporary Variables 11

Use EXECUTE Sparingly 12
- Lag Functions 13
- Using $CASENUM to Select Cases 15
- MISSING VALUES Command 16
- WRITE and XSAVE Commands 16

Using Comments 16

Using SET SEED to Reproduce Random Samples or Values 17

Divide and Conquer 18
Using INSERT with a Master Command Syntax File 19
Defining Global Settings. 19

3 Getting Data into SPSS 22

Getting Data from Databases 22
Installing Database Drivers 22
Database Wizard 24
Reading a Single Database Table. 24
Reading Multiple Tables. 26
Reading Excel Files. 29
Reading a “Typical” Worksheet 30
Reading Multiple Worksheets 32
Reading Text Data Files. 35
Simple Text Data Files 36
Delimited Text Data 37
Fixed-Width Text Data 41
Text Data Files with Very Wide Records 45
Reading Different Types of Text Data 46
Reading Complex Text Data Files. 48
Mixed Files 48
Grouped Files 49
Nested (Hierarchical) Files 52
Repeating Data 58
Reading SAS Data Files 59
Reading Stata Data Files. 61

4 File Operations 62

Working with Multiple Data Sources 62
Merging Data Files 66
Merging Files with the Same Cases but Different Variables 66
Merging Files with the Same Variables but Different Cases 70
Updating Data Files by Merging New Values from Transaction Files 74
Aggregating Data 76
Aggregate Summary Functions 78
Weighting Data 79
Changing File Structure 81
Transposing Cases and Variables 82
Cases to Variables 83
Variables to Cases 86

5 Variable and File Properties 91

Variable Properties 91
Variable Labels 94
Value Labels 94
Missing Values 95
Measurement Level 95
Custom Variable Properties 96
Using Variable Properties as Templates 98
File Properties 99

6 Data Transformations 101

Recoding Categorical Variables 101

Binning Scale Variables 102
Simple Numeric Transformations 105
Arithmetic and Statistical Functions 106
Random Value and Distribution Functions 107
String Manipulation 108
    Changing the Case of String Values 109
    Combining String Values 109
    Taking Strings Apart 110
Working with Dates and Times 114
    Date Input and Display Formats 115
    Date and Time Functions 117

7 Cleaning and Validating Data 123

Finding and Displaying Invalid Values 123
Excluding Invalid Data from Analysis 126
Finding and Filtering Duplicates 127
Data Validation Option 130

8 Conditional Processing, Looping, and Repeating 133

Indenting Commands in Programming Structures 133
Conditional Processing 134
    Conditional Transformations 134
    Conditional Case Selection 137
Simplifying Repetitive Tasks with DO REPEAT 138
    ALL Keyword and Error Handling 141
Vectors 141

Creating Variables with VECTOR 143
Disappearing Vectors 143
Loop Structures 145
Indexing Clauses 146
Nested Loops 147
Conditional Loops 149
Using XSAVE in a Loop to Build a Data File 150
Calculations Affected by Low Default MXLOOPS Setting 152

9 Exporting Data and Results 155

Output Management System. 155
Using Output as Input with OMS 156
Adding Group Percentile Values to a Data File 156
Bootstrapping with OMS 160
Transforming OXML with XSLT 165
“Pushing” Content from an XML File 166
“Pulling” Content from an XML File 169
Positional Arguments versus Localized Text Attributes. 180
Layered Split-File Processing. 181
Exporting Data to Other Applications and Formats 182
Saving Data in SAS Format 182
Saving Data in Stata Format. 183
Saving Data in Excel Format. 184
Writing Data Back to a Database 184
Saving Data in Text Format. 188
Exporting Results to PDF, Word, Excel, and PowerPoint. 188
Controlling and Saving Output Files. 189

10 Scoring Data with Predictive Models 191

Introduction 191
Basics of Scoring Data 192
Transforming Your Data 192
Merging Transformations and Model Specifications 193
Command Syntax for Scoring. 193
Mapping Model Variables to SPSS Variables 195
Missing Values in Scoring 195
Using Predictive Modeling to Identify Potential Customers 196
Building and Saving Predictive Models 196
Commands for Scoring Your Data. 204
Including Post-Scoring Transformations 206
Getting Data and Saving Results 206
Running Your Scoring Job Using the SPSS Batch Facility 208

Part II: Programming with SPSS and Python

11 Introduction 210

12 Getting Started with Python Programming in SPSS 213

The spss Python Module. 214
Submitting Commands to SPSS. 215
Dynamically Creating SPSS Command Syntax. 217
Capturing and Accessing Output. 218
Python Syntax Rules 220

Mixing Command Syntax and Program Blocks 223
Handling Errors. 225
Using a Python IDE 226
Working with Multiple SPSS Versions 229
Creating a Graphical User Interface 229
Supplementary Python Modules for Use with SPSS 235
Getting Help 236

13 Best Practices 237

Creating Blocks of Command Syntax within Program Blocks. 237
Dynamically Specifying Command Syntax Using String Substitution 238
Using Raw Strings in Python 241
Displaying Command Syntax Generated by Program Blocks 242
Handling Wide Output in the Viewer 242
Creating User-Defined Functions in Python 243
Creating a File Handle to the SPSS Install Directory 245
Choosing the Best Programming Technology 246
Using Exception Handling in Python 247
Debugging Your Python Code 250

14 Working with Variable Dictionary Information 254

Summarizing Variables by Measurement Level 256
Listing Variables of a Specified Format 257
Checking If a Variable Exists 259
Creating Separate Lists of Numeric and String Variables. 260
Retrieving Definitions of User-Missing Values 261

Using Object-Oriented Methods for Retrieving Dictionary Information. 262
Getting Started with the VariableDict Class 263
Defining a List of Variables between Two Variables 266
Identifying Variables without Value Labels 267
Retrieving Variable or Datafile Attributes 271
Using Regular Expressions to Select Variables. 273

15 Working with Case Data in the Active Dataset 275

Using the Cursor Class 275
Reading Case Data with the Cursor Class. 276
Creating New SPSS Variables with the Cursor Class 282
Appending New Cases with the Cursor Class. 284
Example: Reducing a String to Minimum Length. 286
Example: Adding Group Percentile Values to a Dataset 288
Using the spssdata Module. 291
Reading Case Data with the Spssdata Class. 292
Creating New SPSS Variables with the Spssdata Class 300
Appending New Cases with the Spssdata Class. 306
Creating a New Dataset with the Spssdata Class 307
Example: Adding Group Percentile Values to a Dataset with the Spssdata Class 308
Example: Generating Simulated Data 311

16 Retrieving Output from SPSS Commands 314

Getting Started with the XML Workspace 314
Writing XML Workspace Contents to a File 317
Using the spssaux Module 318

17 Creating Procedures 327

Getting Started with Procedures 327
Procedures with Multiple Data Passes 332
Creating Pivot Table Output 335
Treating Categories or Cells as Variable Names or Values 340
Specifying Formatting for Numeric Cell Values 342

18 Data Transformations 344

Getting Started with the trans Module 344
Using Functions from the extendedTransforms Module 349
The search and subs Functions 350
The templatesub Function 354
The levenshteindistance Function 357
The soundex and nysiis Functions 357
The strtodatetime Function 360
The datetimetostr Function 360
The lookup Function 361

19 Modifying and Exporting Viewer Contents 363

Getting Started with the viewer Module 364
Persistence of Objects 365
Modifying Pivot Tables 366
Using the viewer Module from a Python IDE 369

20 Tips on Migrating Command Syntax, Macro, and Scripting Jobs to Python 371

Migrating Command Syntax Jobs to Python 371
Migrating Macros to Python 375
Migrating Sax Basic Scripts to Python 379

21 Special Topics 386

Using Regular Expressions 386
Locale Issues 390

22 SPSS for SAS Programmers 392

Reading Data 392
- Reading Database Tables 392
- Reading Excel Files 395
- Reading Text Data 397

Merging Data Files 397
- Merging Files with the Same Cases but Different Variables 398
- Merging Files with the Same Variables but Different Cases 399

Aggregating Data 400
Assigning Variable Properties. 401
- Variable Labels 402
- Value Labels 402

Cleaning and Validating Data 404
- Finding and Displaying Invalid Values. 404
- Finding and Filtering Duplicates 406

Transforming Data Values . . . 407
Recoding Data . . . 407
Banding Data . . . 408
Numeric Functions . . . 410
Random Number Functions . . . 411
String Concatenation . . . 412
String Parsing . . . 413
Working with Dates and Times . . . 414
Calculating and Converting Date and Time Intervals . . . 414
Adding to or Subtracting from One Date to Find Another Date . . . 415
Extracting Date and Time Information . . . 416
Custom Functions, Job Flow Control, and Global Macro Variables . . . 417
Creating Custom Functions . . . 418
Job Flow Control . . . 419
Creating Global Macro Variables . . . 421
Setting Global Macro Variables to Values from the Environment . . . 422

Appendix

A Python Functions and Classes 424

spss.BasePivotTable Class . . . 425
Creating Pivot Tables with the SimplePivotTable Method . . . 427
General Approach to Creating Pivot Tables . . . 429
spss.BasePivotTable Methods . . . 437
Auxiliary Classes for Use with spss.BasePivotTable . . . 450
spss.BaseProcedure Class . . . 456
spss.CreateXPathDictionary Function . . . 459
spss.Cursor Class . . . 460
Read Mode (accessType='r') . . . 460

Write Mode (accessType='w') 462
Append Mode (accessType='a'). 465
spss.Cursor Methods. 467
spss.DeleteXPathHandle Function 488
spss.EndProcedure Function 488
spss.EvaluateXPath Function 489
spss.GetCaseCount Function 490
spss.GetDefaultPlugInVersion Function 490
spss.GetHandleList Function. 491
spss.GetLastErrorLevel and spss.GetLastErrorMessage Functions 491
spss.GetSPSSLowHigh Function. 492
spss.GetVarAttributeNames Function 493
spss.GetVarAttributes Function. 493
spss.GetVariableCount Function 494
spss.GetVariableFormat Function 494
spss.GetVariableLabel Function 497
spss.GetVariableMeasurementLevel Function. 497
spss.GetVariableName Function 498
spss.GetVariableType Function 498
spss.GetVarMissingValues Function 499
spss.GetWeightVar Function. 499
spss.GetXmlUtf16 Function 500
spss.HasCursor Function 500
spss.IsOutputOn Function 500
spss.PyInvokeSpss.IsXDriven Function. 501
spss.SetDefaultPlugInVersion Function. 501
spss.SetMacroValue Function 502
spss.SetOutput Function 502
spss.ShowInstalledPlugInVersions Function 503
spss.SplitChange Function 503
spss.StartProcedure Function. 506

spss.StartSPSS Function 510
spss.StopSPSS Function 510
spss.Submit Function 511
spss.TextBlock Class 512
append Method 514

Index _515_

Chapter 1

Overview

This book is divided into two main sections:

- **Data management using the SPSS 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 and Python.** The SPSS Python plug-in provides the ability to integrate the capabilities of the Python programming language with SPSS. One of the major benefits of Python is the ability to add **jobwise** flow control to the SPSS command stream. SPSS 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 SPSS Python plug-in makes jobwise flow control much easier to accomplish.

For readers who may be more familiar with the commands in the SAS system, Chapter 22 provides examples that demonstrate how some common data management and programming tasks are handled in both SAS and SPSS.

Using This Book

This book is intended for use with SPSS release 15.0. or later. Many examples will work with earlier versions, but some commands and features are not available in earlier releases. Some of the Python examples require SPSS 15.0.1.

Most of the examples shown in this book are designed as hands-on exercises that you can perform yourself. The CD that comes with the book contains the command files and data files used in the examples. All of the sample files are contained in the *examples* folder.

- *\examples\commands* contains SPSS command syntax files.

- *\examples\data* contains data files in a variety of formats.
- *\examples\python* contains sample Python files.

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

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

Many examples, such as the one above, also assume that you have a *C:\temp* folder for writing temporary files. You can access command and data files from the accompanying CD, substituting the drive location for *C:* in file access commands. For commands that write files, however, you need to specify a valid folder location on a device for which you have write access.

Documentation Resources

The *SPSS Base 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 Command Syntax Reference*, which is installed as a PDF file with the SPSS system, is a complete guide to the specifications for each SPSS 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 Command Syntax Reference* 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

Chapter

2 Best Practices and Efficiency Tips

If you haven't worked with SPSS 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 SPSS command syntax, but there are a few basic things you should know. A detailed introduction to SPSS command syntax is available in the "Universals" section in the *SPSS Command Syntax Reference*.

Creating Command Syntax Files

An SPSS command file is a simple text file. You can use any text editor to create a command syntax file, but SPSS 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.

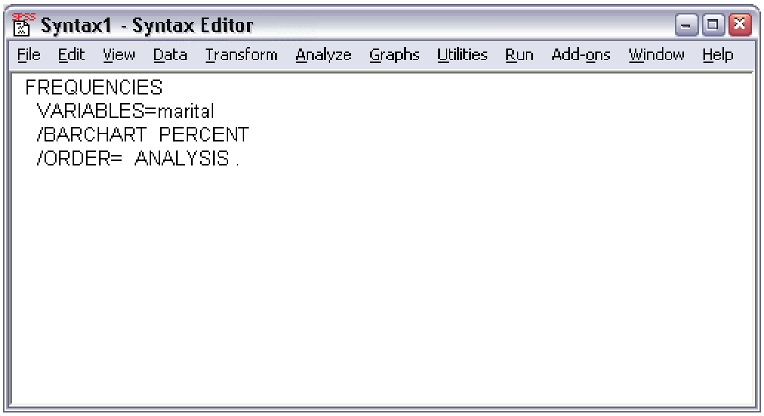
- **Retrieve commands from the journal file.** Most actions that you perform in the graphical user interface (and all commands that you run from a command syntax window) are automatically recorded in the journal file in the form of command syntax. The default name of the journal file is *spss.jnl*. The 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 menu, Options).

Running SPSS 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 Using INSERT with a Master Command Syntax File on p. 19.
- 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 SPSSB (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 SPSSB documentation supplied with the SPSS server software for more information.

Figure 2-1
Command syntax pasted from a dialog box



Syntax Rules

- Commands run from a command syntax window during a typical SPSS session must follow the **interactive** command syntax rules.
- Commands files run via SPSSB 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 or production 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.

Customizing the Programming Environment

There are a few global settings and customization features that may make working with command syntax a little easier.

Displaying Commands in the Log

By default, commands that have been run are not displayed in the log, which can make it difficult to interpret error messages. To display commands in the log, use the command:

```
SET PRINTBACK = ON.
```

Or, using the graphical user interface:

▶ From the menus, choose:

Edit
  Options...

▶ Click the Viewer tab.

▶ Select (check) Display commands in the log.

Figure 2-2
Log with and without commands displayed

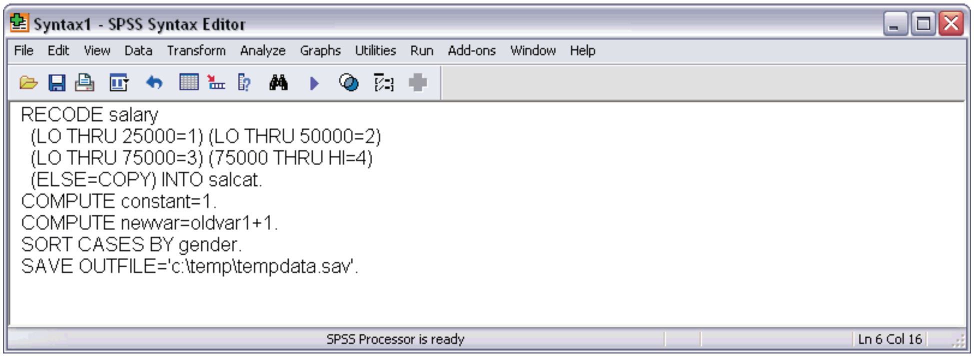
```
Log without commands displayed

>Error # 4285 in column 16.  Text: oldvar1
>Incorrect variable name: either the name is more than 64 characters, or it
>is not defined by a previous command.
>This command not executed.


Log with commands displayed

RECODE salary
  (LO THRU 25000=1) (LO THRU 50000=2)
  (LO THRU 75000=3) (75000 THRU HI=4)
  (ELSE=COPY) INTO salcat.
COMPUTE constant=1.
COMPUTE newvar=oldvar1+1.

>Error # 4285 in column 16.  Text: oldvar1
>Incorrect variable name: either the name is more than 64 characters, or it
>is not defined by a previous command.
>This command not executed.

SORT CASES BY gender.
SAVE OUTFILE='c:\temp\tempdata.sav'.
```

Displaying the Status Bar in Command Syntax Windows

In addition to various status messages, the status bar at the bottom of a command syntax window displays the current line number and character position within the line. Since error messages typically contain information about the column position where an error was encountered, the column position information in the status bar can help you to pinpoint errors. (*Note*: You may have to increase the width of the command syntax window to see this information.)

The status bar is displayed by default. If it is currently not displayed, choose Status Bar from the View menu in the command syntax window.

Figure 2-3
Status bar in command syntax window with current line number and column position displayed



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 SPSS. However, if the original data are in SPSS-format 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 an SPSS-format data file. 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 and do *all* data transformations, recoding, and calculations using SPSS. 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 SPSS 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

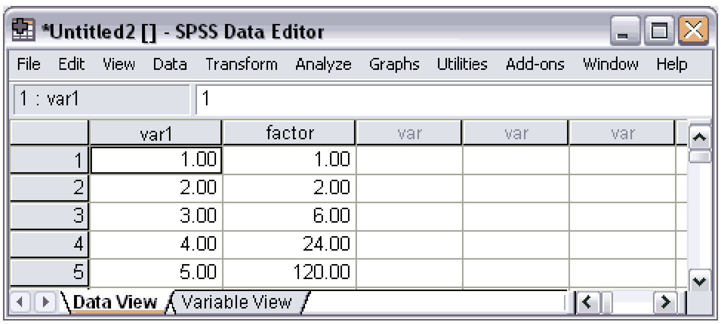
For transformations that require intermediate variables, use scratch (temporary) variables for the intermediate values. Any variable name that begins with a pound sign (#) is treated as a scratch variable that is discarded at the end of the series of transformation commands when SPSS encounters an `EXECUTE` 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 2-4
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

SPSS is designed to work with large data files (the current version can accommodate 2.15 billion cases). 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:

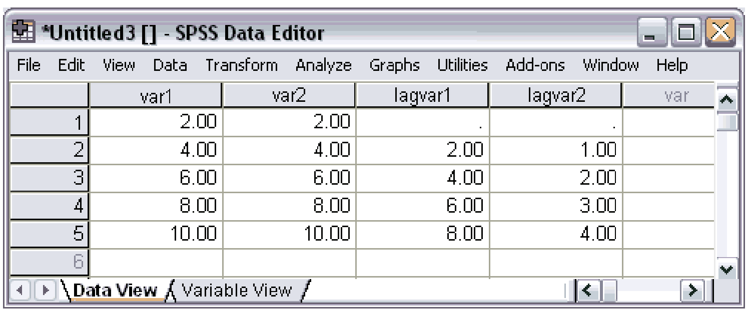
- The lag variable is also used in any of the other transformation commands.
- One of the transformations selects a subset of cases and deletes the unselected cases, such as `SELECT IF` or `SAMPLE`.

Example

```
*lagfunction.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 2-5
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.

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 Using XSAVE in a Loop to Build a Data File in Chapter 8 on p. 150.

Using Comments

It is always a good practice to include explanatory comments in your code. In SPSS, 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 = 'c:\examples\data\onevar.sav'.
SET SEED = 123456789.
SAMPLE .1.
LIST.
GET FILE = 'c:\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

SPSS provides 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 = "c:\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 SPSS 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='c:\examples\data'.
FILE HANDLE commands /NAME='c:\examples\commands'.
FILE HANDLE spssdir /NAME='c:\program files\spss'.
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 SPSS folder. This path can be useful if you use any of the command syntax or script files that are installed with SPSS.
- 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 = "c:\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 `"c:\examples\data"` and `"c:\examples\commands"`, respectively.

Note: Using absolute paths or file handles that represent those paths is the most reliable way to make sure that SPSS 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.

Chapter

3

Getting Data into SPSS

Before you can work with data in SPSS, 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 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 an SPSS-format data file 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

SPSS 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, 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 for a wide variety of database formats are included on the SPSS installation CD, including:

- Access

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

Most of these drivers can be installed by installing the SPSS Data Access Pack. You can install the SPSS Data Access Pack from the AutoPlay menu on the SPSS installation CD.

If you need a Microsoft Access driver, you will need to install the Microsoft Data Access Pack. An installable version is located in the *Microsoft Data Access Pack* folder on the SPSS installation CD.

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

OLE DB

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

To access OLE DB data sources, you must have the following items installed on the computer that is running SPSS:

- .NET framework
- Dimensions Data Model and OLE DB Access

Versions of these components that are compatible with this release of SPSS can be installed from the SPSS installation CD and are available on the AutoPlay menu.

- 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 SPSS Server), OLE DB data sources are available only on Windows servers, and both .NET and the Dimensions Data Model and OLE DB Access must be installed on the server.

Database Wizard

It's probably a good idea to use the Database Wizard (File menu, 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

SPSS 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 an SPSS data file: records are cases and fields are variables. So, reading a single table can be very simple.

Example

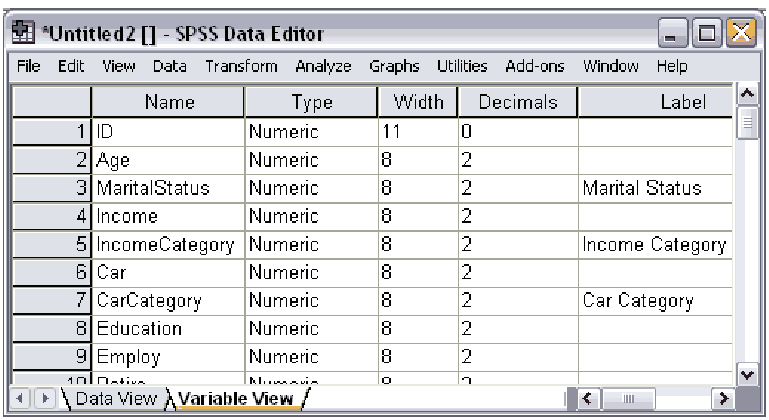
This example reads a single table from an Access database. It reads all records and fields in the table.

```
*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 Reading Multiple Worksheets on p. 32.
- `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 SPSS 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 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]'.
```

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

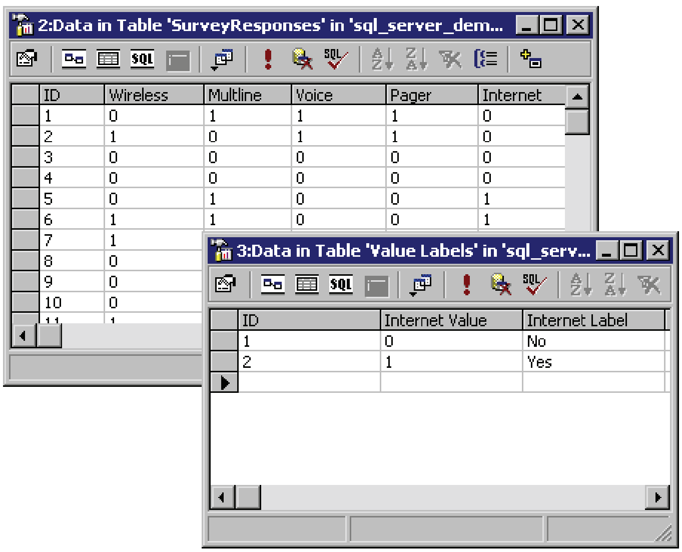
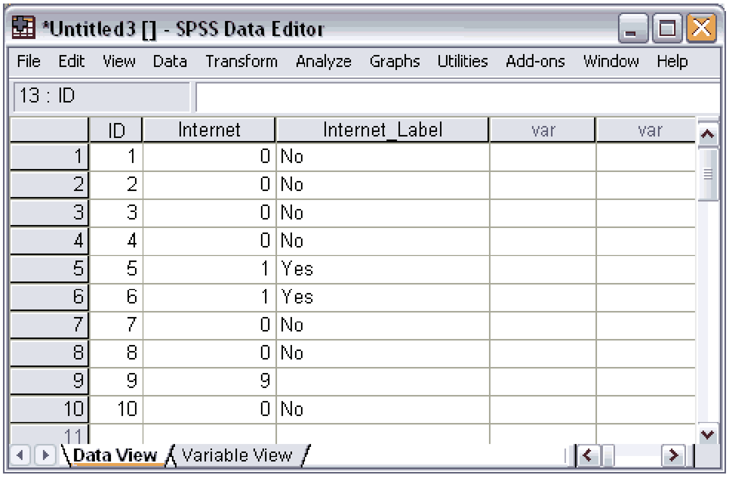


Figure 3-3
Active dataset in SPSS



- `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 Excel Files

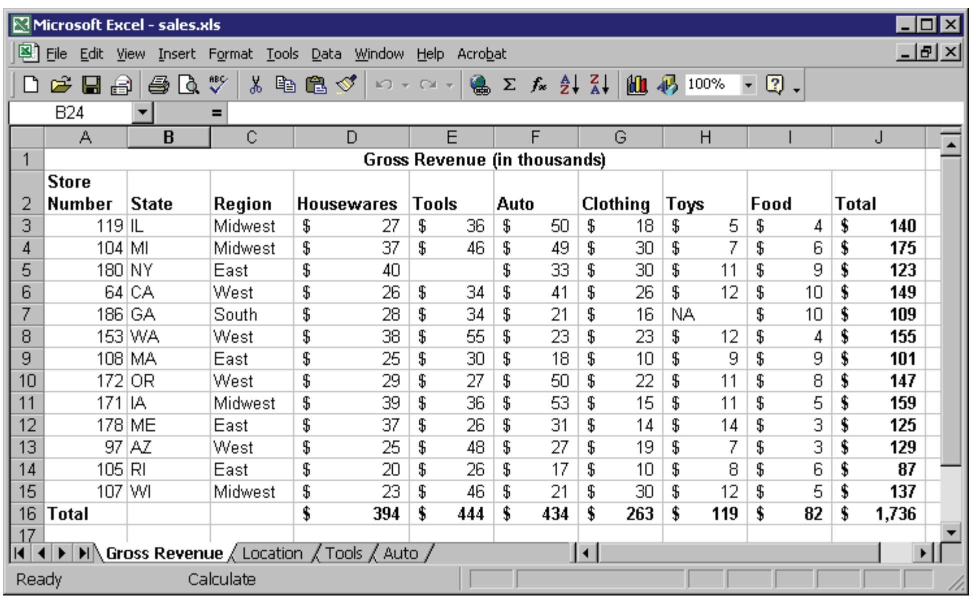
SPSS can read individual Excel worksheets and multiple worksheets in the same Excel workbook. The basic mechanics of reading Excel files are relatively straightforward—rows are read as cases and columns are read as variables. However, reading a typical Excel spreadsheet—where the data may not start in 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 into SPSS.

Reading a "Typical" Worksheet

When reading an individual worksheet, SPSS 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 an SPSS data file 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



	A	B	C	D	E	F	G	H	I	J
1					Gross Revenue (in thousands)					
2	Store Number	State	Region	Housewares	Tools	Auto	Clothing	Toys	Food	Total
3	119	IL	Midwest	$ 27	$ 36	$ 50	$ 18	$ 5	$ 4	$ 140
4	104	MI	Midwest	$ 37	$ 46	$ 49	$ 30	$ 7	$ 6	$ 175
5	180	NY	East	$ 40		$ 33	$ 30	$ 11	$ 9	$ 123
6	64	CA	West	$ 26	$ 34	$ 41	$ 26	$ 12	$ 10	$ 149
7	186	GA	South	$ 28	$ 34	$ 21	$ 16	NA	$ 10	$ 109
8	153	WA	West	$ 38	$ 55	$ 23	$ 23	$ 12	$ 4	$ 155
9	108	MA	East	$ 25	$ 30	$ 18	$ 10	$ 9	$ 9	$ 101
10	172	OR	West	$ 29	$ 27	$ 50	$ 22	$ 11	$ 8	$ 147
11	171	IA	Midwest	$ 39	$ 36	$ 53	$ 15	$ 11	$ 5	$ 159
12	178	ME	East	$ 37	$ 26	$ 31	$ 14	$ 14	$ 3	$ 125
13	97	AZ	West	$ 25	$ 48	$ 27	$ 19	$ 7	$ 3	$ 129
14	105	RI	East	$ 20	$ 26	$ 17	$ 10	$ 8	$ 6	$ 87
15	107	WI	Midwest	$ 23	$ 46	$ 21	$ 30	$ 12	$ 5	$ 137
16	Total			$ 394	$ 444	$ 434	$ 263	$ 119	$ 82	$ 1,736
17										

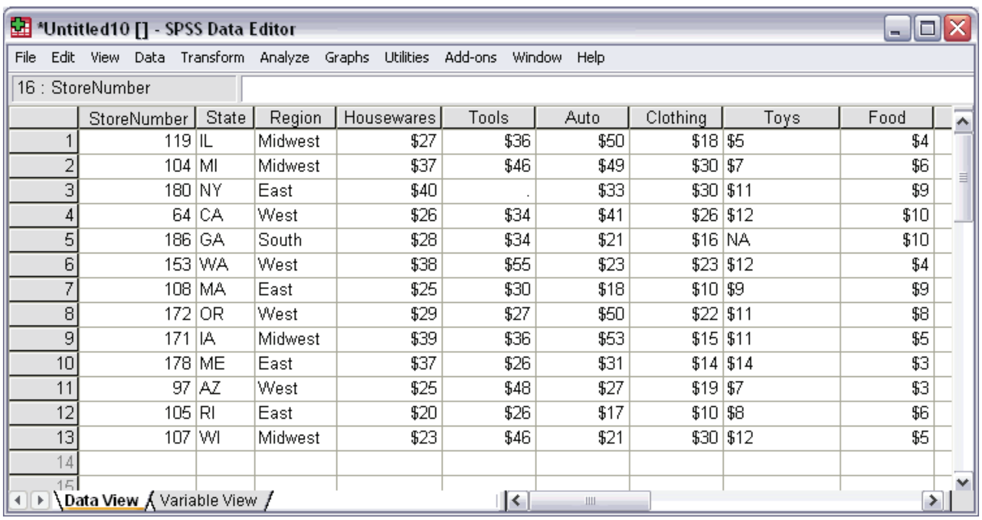
To read this spreadsheet without the title row or total row and column:

```
*readexcel.sps.
GET DATA
```

```
/TYPE=XLS
/FILE='c:\examples\data\sales.xls'
/SHEET=NAME 'Gross Revenue'
/CELLRANGE=RANGE 'A2:I15'
/READNAMES=on .
```

- The `TYPE` subcommand identifies the file type as Excel, version 5 or later. (For earlier versions, use `GET TRANSLATE`.)
- 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 SPSS should start reading 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 SPSS



	StoreNumber	State	Region	Housewares	Tools	Auto	Clothing	Toys	Food
1	119	IL	Midwest	$27	$36	$50	$18	$5	$4
2	104	MI	Midwest	$37	$46	$49	$30	$7	$6
3	180	NY	East	$40	.	$33	$30	$11	$9
4	64	CA	West	$26	$34	$41	$26	$12	$10
5	186	GA	South	$28	$34	$21	$16	NA	$10
6	153	WA	West	$38	$55	$23	$23	$12	$4
7	108	MA	East	$25	$30	$18	$10	$9	$9
8	172	OR	West	$29	$27	$50	$22	$11	$8
9	171	IA	Midwest	$39	$36	$53	$15	$11	$5
10	178	ME	East	$37	$26	$31	$14	$14	$3
11	97	AZ	West	$25	$48	$27	$19	$7	$3
12	105	RI	East	$20	$26	$17	$10	$8	$6
13	107	WI	Midwest	$23	$46	$21	$30	$12	$5
14									
15									

- The Excel column label *Store Number* is automatically converted to the SPSS 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, this isn't always possible.
- When SPSS encounters mixed data types 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

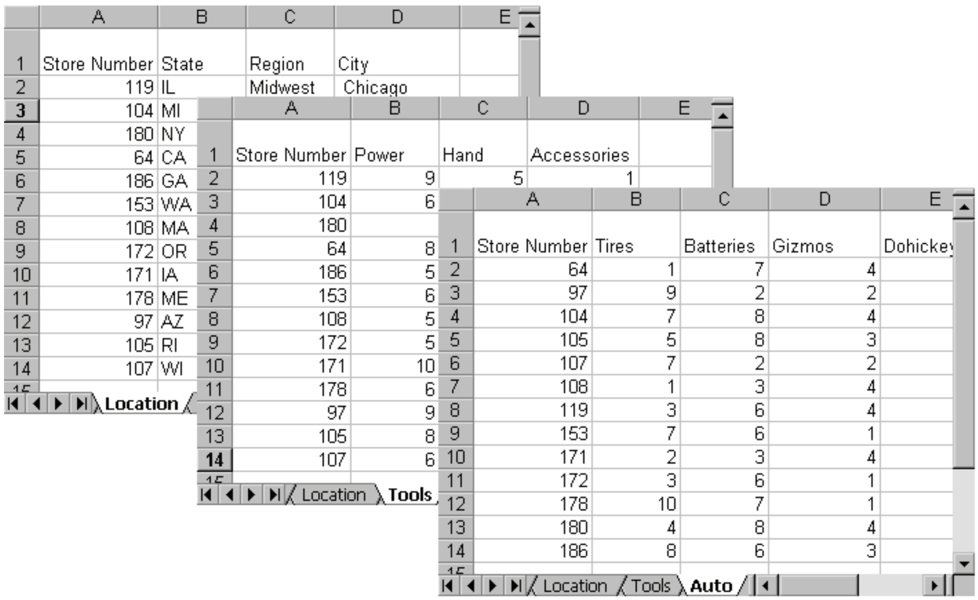
The `READNAMES` subcommand tells SPSS to treat the first row of the spreadsheet or specified range as either variable names (`ON`) or data (`OFF`). This subcommand will always affect the way the Excel spreadsheet is read, even when it isn't specified, since the default setting is `ON`.

- With `READNAMES=ON` (or in the absence of this subcommand), if the first row contains data instead of column headings, SPSS 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



	A	B	C	D	E
1	Store Number	State	Region	City	
2	119	IL	Midwest	Chicago	
3	104	MI			
4	180	NY			
5	64	CA			
6	186	GA			
7	153	WA			
8	108	MA			
9	172	OR			
10	171	IA			
11	178	ME			
12	97	AZ			
13	105	RI			
14	107	WI			

Location

	A	B	C	D	E
1	Store Number	Power	Hand	Accessories	
2	119	9	5	1	
3	104	6			
4	180				
5	64	8			
6	186	5			
7	153	6			
8	108	5			
9	172	5			
10	171	10			
11	178	6			
12	97	9			
13	105	8			
14	107	6			

Location Tools

	A	B	C	D	E
1	Store Number	Tires	Batteries	Gizmos	Dohickey
2	64	1	7	4	
3	97	9	2	2	
4	104	7	8	4	
5	105	5	8	3	
6	107	7	2	2	
7	108	1	3	4	
8	119	3	6	4	
9	153	7	6	1	
10	171	2	3	4	
11	172	3	6	1	
12	178	10	7	1	
13	180	4	8	4	
14	186	8	6	3	

Location Tools Auto

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

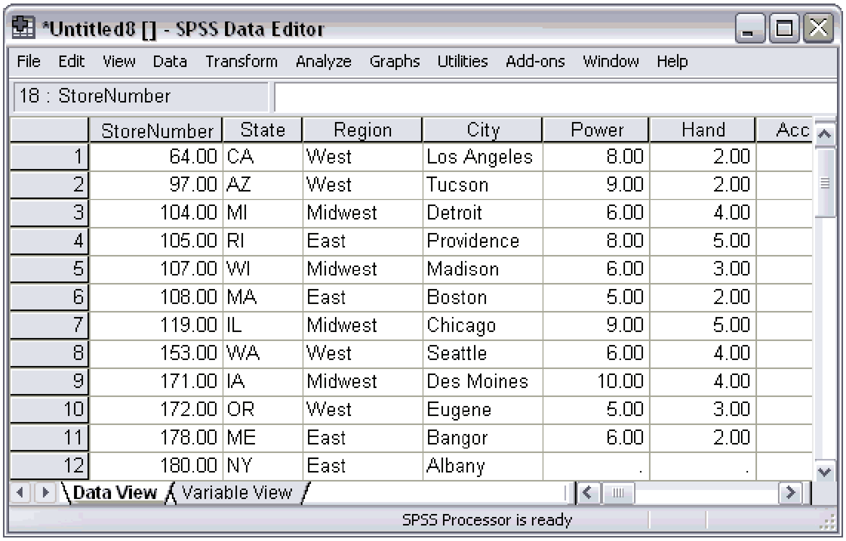
In this example, the first worksheet contains information about store location, and the second and third contain information for different departments. All three contain a column, *Store Number*, that uniquely identifies each store, so, the information in the three sheets can be merged correctly regardless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



	StoreNumber	State	Region	City	Power	Hand	Acc
1	64.00	CA	West	Los Angeles	8.00	2.00	
2	97.00	AZ	West	Tucson	9.00	2.00	
3	104.00	MI	Midwest	Detroit	6.00	4.00	
4	105.00	RI	East	Providence	8.00	5.00	
5	107.00	WI	Midwest	Madison	6.00	3.00	
6	108.00	MA	East	Boston	5.00	2.00	
7	119.00	IL	Midwest	Chicago	9.00	5.00	
8	153.00	WA	West	Seattle	6.00	4.00	
9	171.00	IA	Midwest	Des Moines	10.00	4.00	
10	172.00	OR	West	Eugene	5.00	3.00	
11	178.00	ME	East	Bangor	6.00	2.00	
12	180.00	NY	East	Albany	.	.	

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 SPSS.
- **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 menu, 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 SPSS 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 Reading Complex Text Data Files on p. 48.
- You want to use the `TO` keyword to define a large number of sequential variable names (for example, `var1 TO var1000`).

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 = 'c:\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 = 'c:\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.

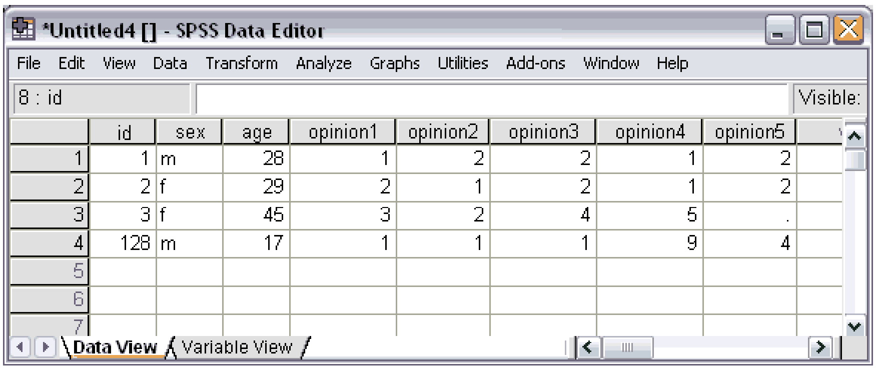
The data file, *delimited_list.txt*, contains one case that has only seven values recorded, whereas all of the others have eight:

```
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='c:\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



	id	sex	age	opinion1	opinion2	opinion3	opinion4	opinion5
1	1	m	28	1	2	2	1	2
2	2	f	29	2	1	2	1	2
3	3	f	45	3	2	4	5	.
4	128	m	17	1	1	1	9	4
5								
6								
7								

- 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 = 'C:\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 Reading Different Types of Text Data on p. 46.

Note: The command syntax in this example was adapted from the command syntax generated by the Text Wizard (File menu, Read Text Data), which automatically generated valid SPSS 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='c:\examples\data\simple_fixed.txt'
  /id 1-3 sex 5 (A) age 7-8 opinion1 TO opinion5 10-14.
EXECUTE.
```

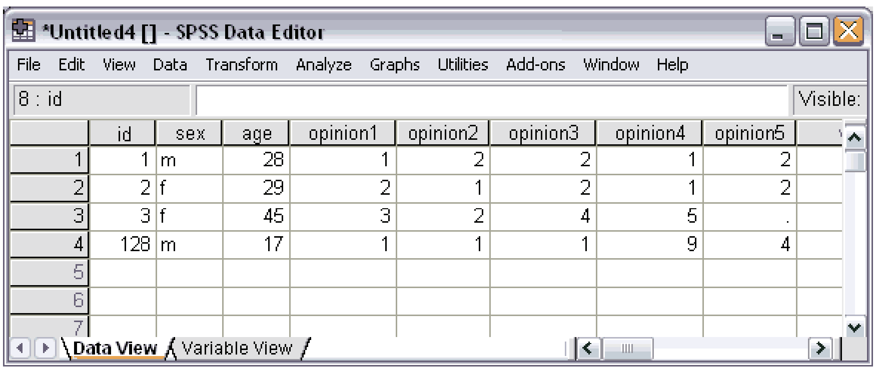
- The keyword `FIXED` is included in this example, but since it is the default format, it can be omitted.
- The forward slash before the variable *id* separates the variable definitions from the rest of the command specifications (unlike other commands where subcommands are separated by forward slashes). 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='c:\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 = 'C:\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
```

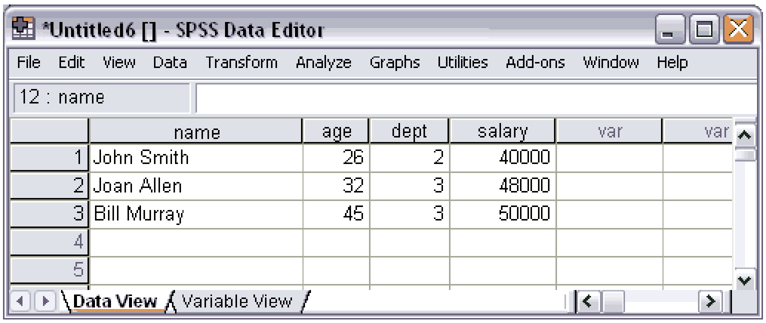
The `DATA LIST` command syntax to read the file is:

```
*skip_first_fixed.sps.
DATA LIST FIXED
 FILE = 'c:\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 = 'c:\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='c:\examples\data\wide_file.txt'
  /var1 TO var1000 (1000F10).
EXECUTE.
*Define record length first.
FILE HANDLE  wide_file NAME = 'c:\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

SPSS 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

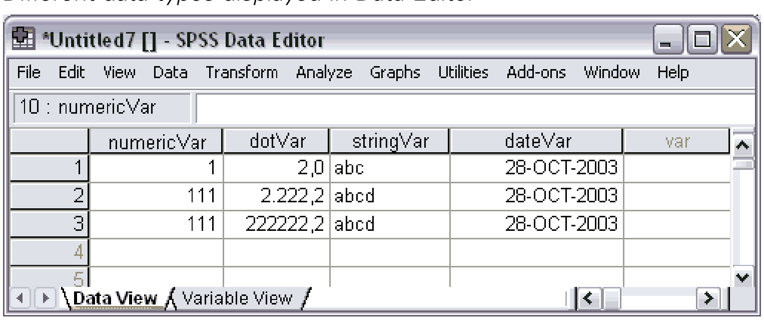
Type	Example	Format specification
Dollar	$12,345	DOLLAR7
	$12,234.50	DOLLAR9.2
String (alphanumeric)	Female	A6
International date	28-OCT-1986	DATE11
American date	10/28/1986	ADATE10
Date and time	28 October, 1986 23:56	DATETIME22

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

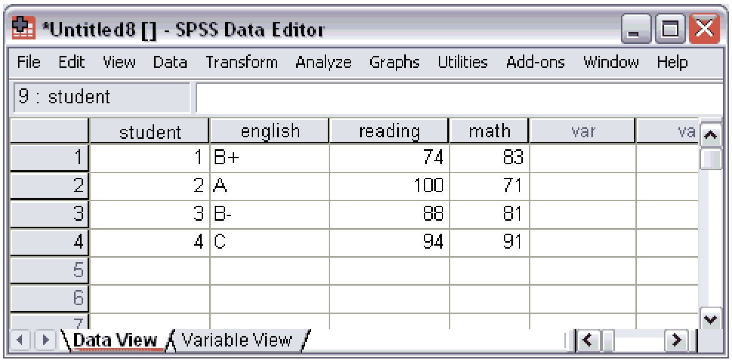
Example

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='c:\temp\tempdata.txt'
 /alldata.
EXECUTE.
* read the sorted file.
FILE TYPE GROUPED FILE='c:\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 SPSS 13.0, the maximum width of a string variable was 255 characters. So in earlier releases, for a file with records wider than 255 characters, 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-character string variables.

SPSS can now handle much longer strings in a single variable: 32,767 bytes. So this workaround is unnecessary for SPSS 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 *SPSS Command Syntax Reference* for more information.)

Nested (Hierarchical) Files

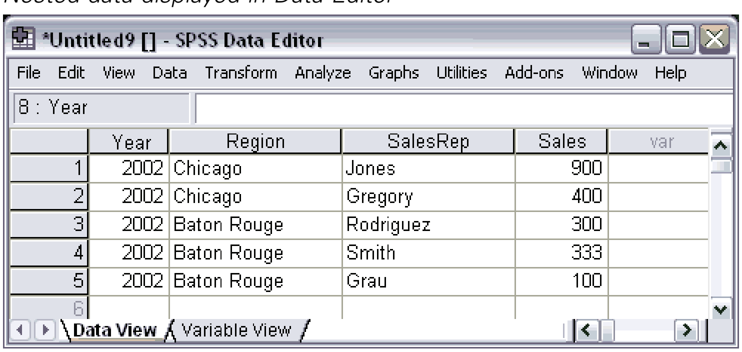
In a nested file, the record types are related to each other hierarchically. The record types are grouped together by a case identification number that identifies the highest level—the first record type—of the hierarchy. Usually, the last record type specified—the lowest level of the hierarchy—defines 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



	Year	Region	SalesRep	Sales	var
1	2002	Chicago	Jones	900	
2	2002	Chicago	Gregory	400	
3	2002	Baton Rouge	Rodriguez	300	
4	2002	Baton Rouge	Smith	333	
5	2002	Baton Rouge	Grau	100	
6					

- 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.
```

```
COMPUTE #count=0.
- 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.
```

- `COMPUTE #count=0` initializes a case-counter variable.
- `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.

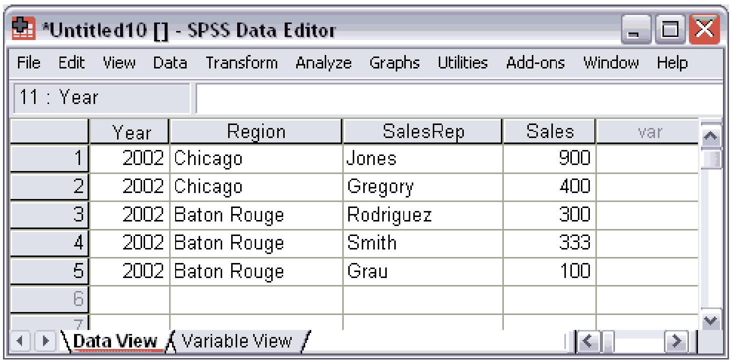
```
*repeating_data.sps.
INPUT PROGRAM.
DATA LIST FIXED
  /Year 1-4 Region 6-16 (A) #numrep 19.
REPEATING DATA STARTS=22 /OCCURS=#numrep
  /DATA=SalesRep 1-10 (A) Sales 12-14.
END INPUT PROGRAM.
BEGIN DATA
2002 Chicago      2  Jones      900Gregory    400
2002 Baton Rouge  3  Rodriguez  300Smith    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

SPSS 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 an SPSS data file—rows are cases (observations), and columns are variables—and reading SAS data files requires only a single, simple command: `GET SAS`.

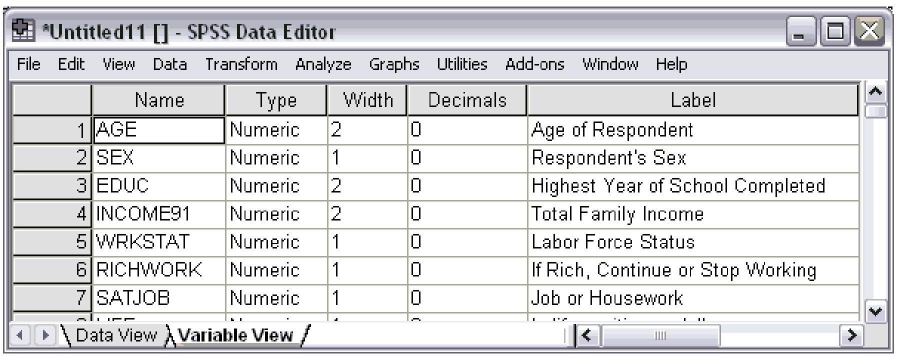
Example

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

```
*get_sas.sps.
GET SAS DATA='C:\examples\data\gss.sd2'.
```

- SAS variable names that do not conform to SPSS variable-naming rules are converted to valid SPSS variable names.
- SAS variable labels specified on the `LABEL` statement in the `DATA` step are used as variable labels in SPSS.

Figure 3-15
SAS data file with variable labels in SPSS



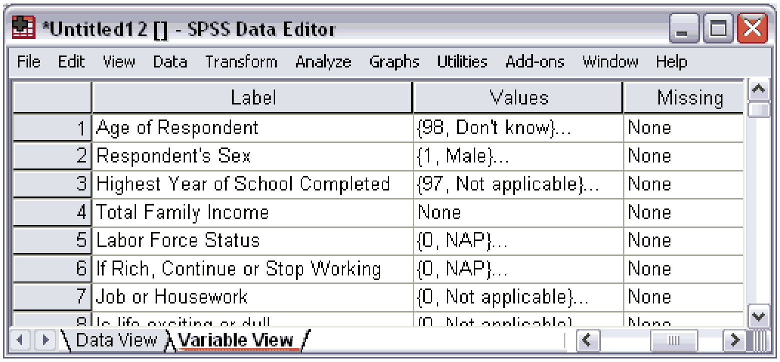
Example

SAS value formats are similar to SPSS 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_sas2.sps.
GET SAS DATA='C:\examples\data\gss.sd2'
 FORMATS='c:\examples\data\GSS_Fmts.sd2'.
```

- 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.
- Labels assigned to string variables and non-integer numeric values are ignored.

Figure 3-16
SAS value formats used as value labels



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 SPSS 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*, ..., etc.).
- **Variable labels.** Stata variable labels are converted to SPSS variable labels.
- **Value labels.** Stata value labels are converted to SPSS 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 `DATE` format (d-m-y) values. Stata “time-series” date format values (weeks, months, quarters, etc.) are converted to simple numeric (F) format, preserving the original, internal integer value, which is the number of weeks, months, quarters, etc., since the start of 1960.

Example

```
GET STATA FILE='c:\examples\data\statafile.dta'.
```

Chapter

4

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. 156.

Working with Multiple Data Sources

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

- When you use the dialog boxes and wizards in the graphical user interface to read data into SPSS, 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 SPSS 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.
- 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 a reference to an SPSS data file.
- Wherever a dataset name, file handle (defined by the FILE HANDLE command), or filename can be used to refer to an SPSS data file, 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.

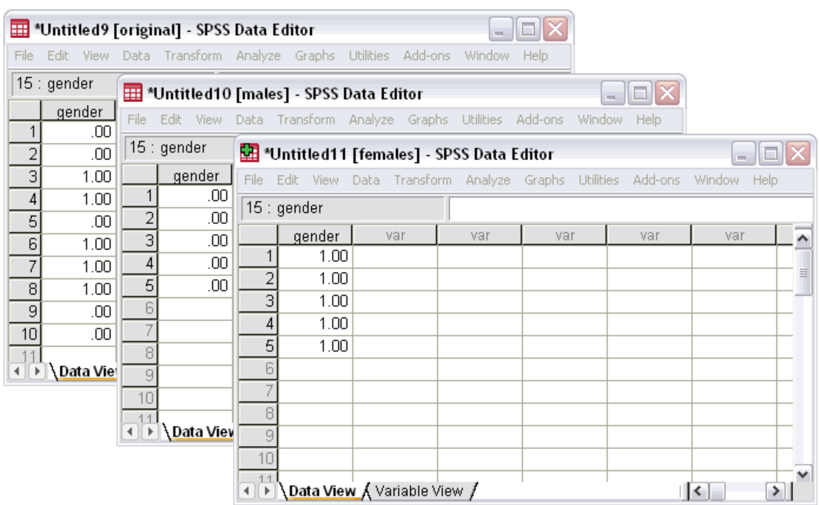
Example

```
*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 SPSS-format data files rather than database tables. For information on reading multiple database tables with joins, see Reading Multiple Tables in Chapter 3 on p. 26.

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='C:\examples\data\match_response1.sav'.
SORT CASES BY id.
DATASET NAME responses.
GET FILE='C:\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:

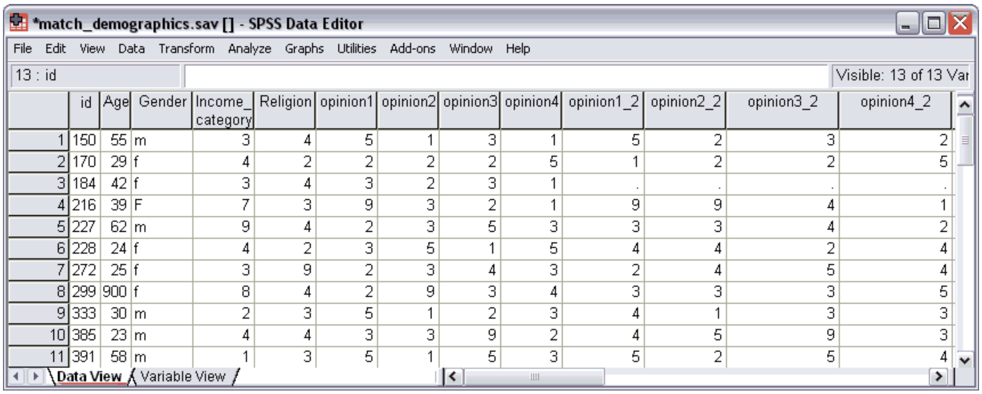
- The variable names for the survey questions are the same as the variable names in the survey response data file from the earlier date.
- One of the cases that is present in both the demographic data file and the first survey response file is missing from the new survey response data file.
- The source file is not an SPSS-format data file; it's an Excel worksheet.

```
*match_files2.sps.
GET FILE='C:\examples\data\match_response1.sav'.
SORT CASES BY id.
DATASET NAME response1.
GET DATA /TYPE=XLS
   /FILE='c:\examples\data\match_response2.xls'.
SORT CASES BY id.
DATASET NAME response2.
GET FILE='C:\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



	id	Age	Gender	Income_category	Religion	opinion1	opinion2	opinion3	opinion4	opinion1_2	opinion2_2	opinion3_2	opinion4_2
1	150	55	m	3	4	5	1	3	1	5	2	3	2
2	170	29	f	4	2	2	2	2	5	1	2	2	5
3	184	42	f	3	4	3	2	3	1	.	.	.	.
4	216	39	F	7	3	9	3	2	1	9	9	4	1
5	227	62	m	9	4	2	3	5	3	3	3	4	2
6	228	24	f	4	2	3	5	1	5	4	4	2	4
7	272	25	f	3	9	2	3	4	3	2	4	5	4
8	299	900	f	8	4	2	9	3	4	3	3	3	5
9	333	30	m	2	3	5	1	2	3	4	1	3	3
10	385	23	m	4	4	3	3	9	2	4	5	9	3
11	391	58	m	1	3	5	1	5	3	5	2	5	4

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 sources that you can read into SPSS. For information on reading different types of data into SPSS, see Chapter 3 on p. 22.

```
*match_table_lookup.sps.
DATA LIST LIST
  FILE='c:\examples\data\family_data.txt'
   /household_id total_income family_size region.
SORT CASES BY household_id.
```

```
DATASET NAME household.
DATA LIST LIST
  FILE='c:\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

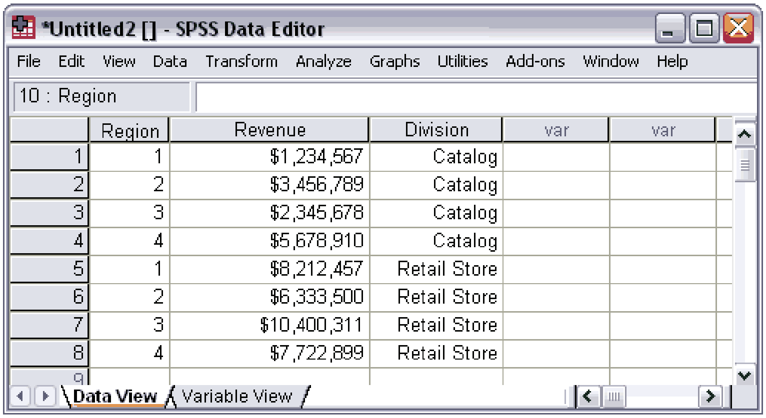
The ADD FILES command merges two or more data files that contain the same variables but different cases. For example, regional revenue for two different company divisions might be stored in two separate data files. Both files have the same variables (region indicator and revenue) but different cases (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 = 'c:\examples\data\catalog.sav'
  /FILE =' c:\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



	Region	Revenue	Division	var	var
1	1	$1,234,567	Catalog		
2	2	$3,456,789	Catalog		
3	3	$2,345,678	Catalog		
4	4	$5,678,910	Catalog		
5	1	$8,212,457	Retail Store		
6	2	$6,333,500	Retail Store		
7	3	$10,400,311	Retail Store		
8	4	$7,722,899	Retail Store		
9					

- 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

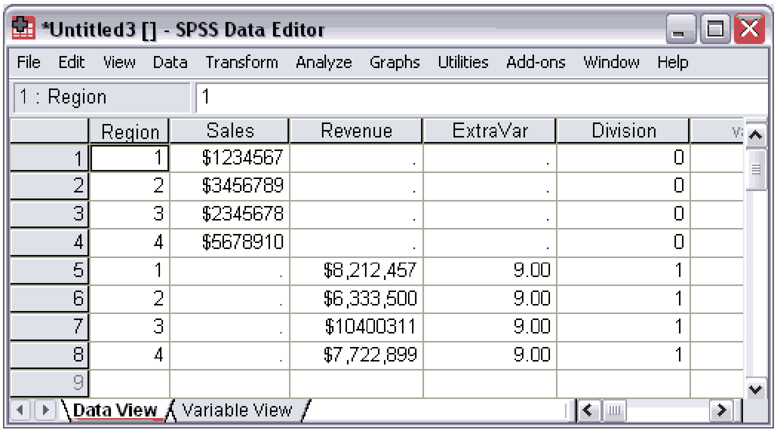
Now that we've had a good laugh over the likelihood that all of the files have the exact same structure with the exact same variable names, let's look at a more realistic example. What if the revenue variable had a different name in one of the files and one of the files contained additional variables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

```
*add_files2.sps.
***first throw some curves into the data***.
GET FILE = 'c:\examples\data\catalog.sav'.
RENAME VARIABLES (Revenue=Sales).
DATASET NAME catalog.
GET FILE = 'c:\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.
```

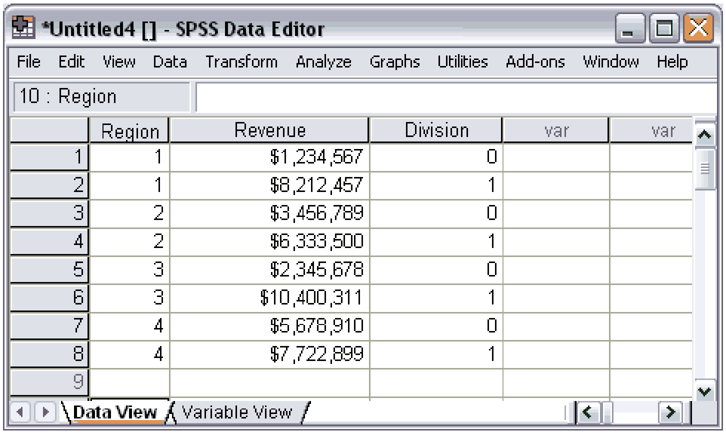
- All of the commands prior to the first `ADD FILES` command simply modify the original data files to contain minor variations—*Revenue* is changed to *Sales* in one data file, and an extra variable, *ExtraVar*, is added to the other data file.
- The first `ADD FILES` command is similar to the one in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



	Region	Revenue	Division	var	var
1	1	$1,234,567	0		
2	1	$8,212,457	1		
3	2	$3,456,789	0		
4	2	$6,333,500	1		
5	3	$2,345,678	0		
6	3	$10,400,311	1		
7	4	$5,678,910	0		
8	4	$7,722,899	1		
9					

Updating Data Files by Merging New Values from Transaction Files

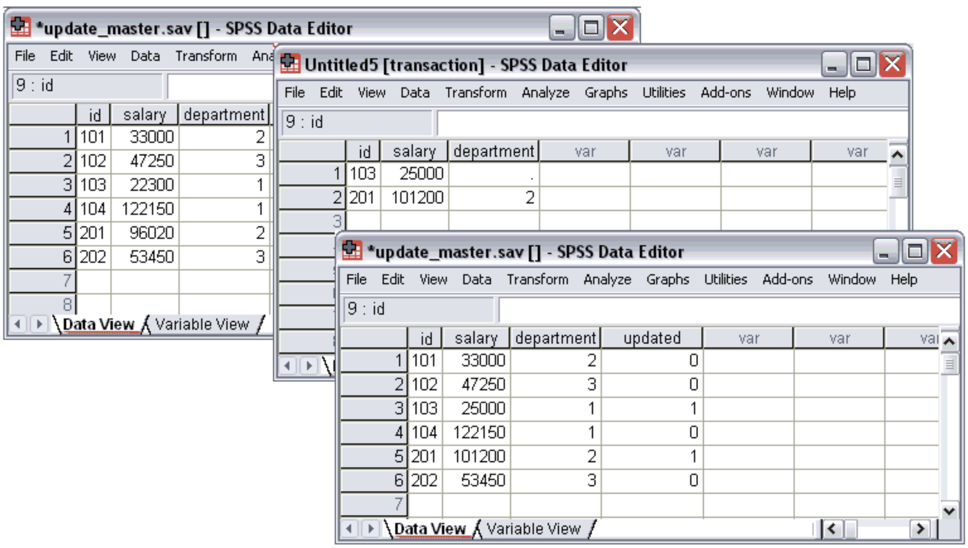
You can use the UPDATE command to replace values in a master file with updated values recorded in one or more files called transaction files.

```
*update.sps.
GET FILE = 'c:\examples\data\update_transaction.sav'.
SORT CASE BY id.
DATASET NAME transaction.
GET FILE = 'c:\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.

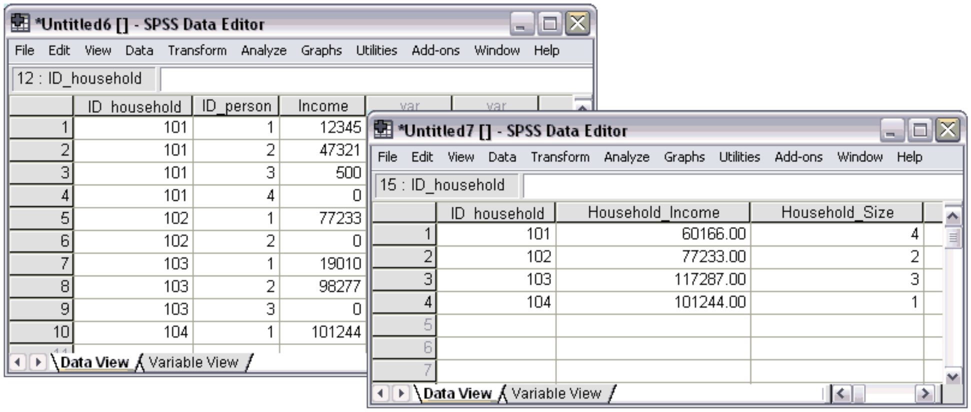
Example

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

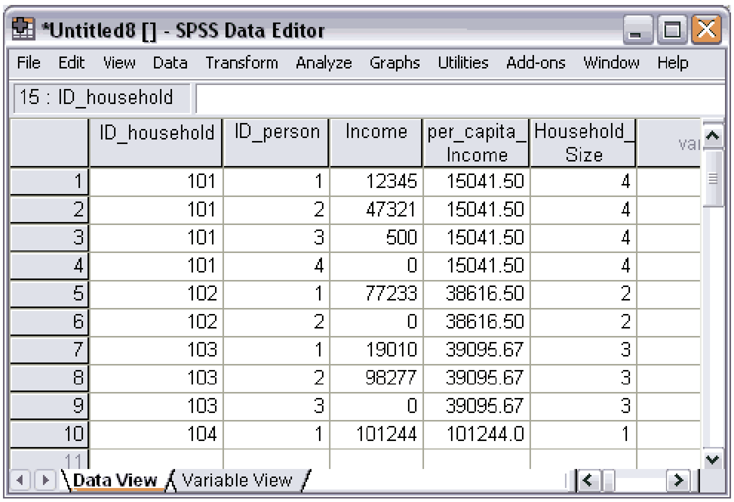
You can also use `MODE = ADDVARIABLES` to add group summary information to the original data file. For example, you could create two new variables in the original data file that contain the number of people in the household and the per capita income for the household (total income divided by number of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



	ID_household	ID_person	Income	per_capita_Income	Household_Size
1	101	1	12345	15041.50	4
2	101	2	47321	15041.50	4
3	101	3	500	15041.50	4
4	101	4	0	15041.50	4
5	102	1	77233	38616.50	2
6	102	2	0	38616.50	2
7	103	1	19010	39095.67	3
8	103	2	98277	39095.67	3
9	103	3	0	39095.67	3
10	104	1	101244	101244.0	1

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 *SPSS 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 into SPSS, 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

SPSS 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.

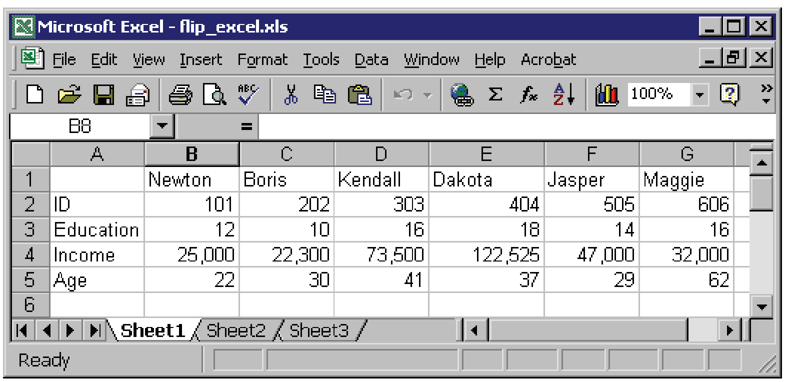
Transposing Cases and Variables

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 SPSS 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



	A	B	C	D	E	F	G
1		Newton	Boris	Kendall	Dakota	Jasper	Maggie
2	ID	101	202	303	404	505	606
3	Education	12	10	16	18	14	16
4	Income	25,000	22,300	73,500	122,525	47,000	32,000
5	Age	22	30	41	37	29	62
6							

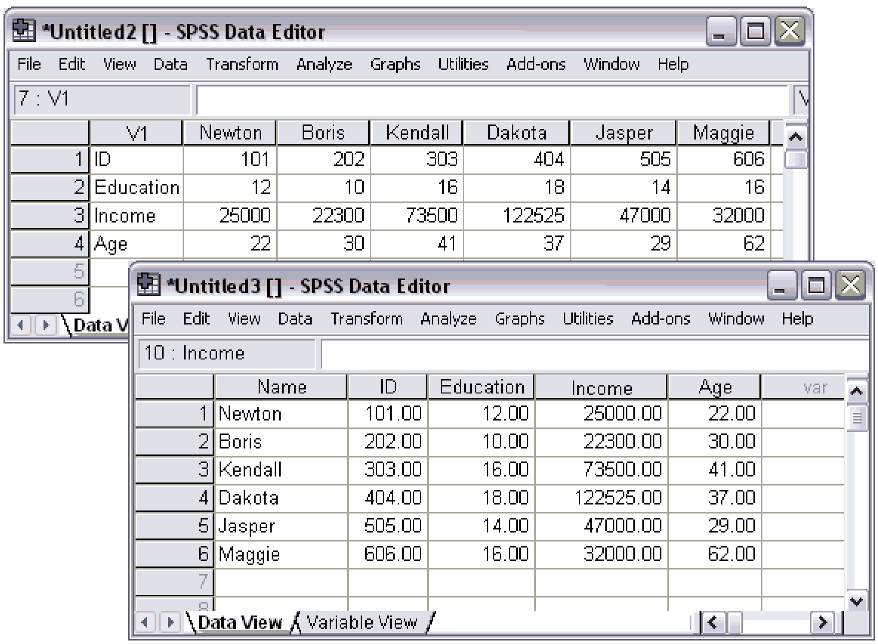
Here are the commands to read the Excel spreadsheet and transpose the rows and columns:

```
*flip_excel.sps.
GET DATA /TYPE=XLS
  /FILE='C:\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



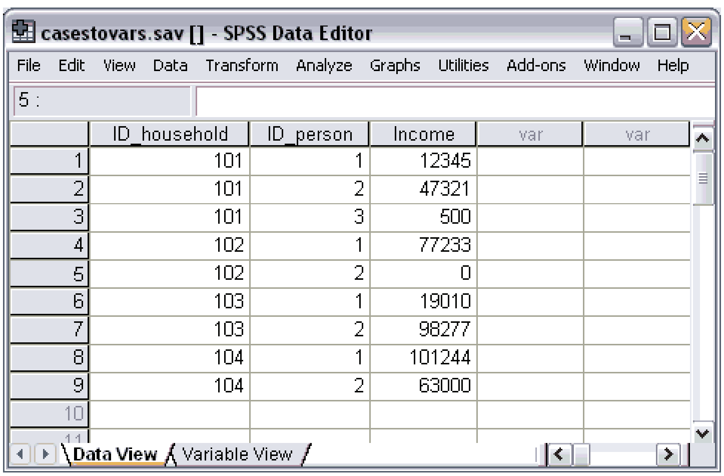
Cases to Variables

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

Many statistical techniques in SPSS 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 Aggregating Data on p. 76. 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



	ID_household	ID_person	Income	var	var
1	101	1	12345		
2	101	2	47321		
3	101	3	500		
4	102	1	77233		
5	102	2	0		
6	103	1	19010		
7	103	2	98277		
8	104	1	101244		
9	104	2	63000		
10					

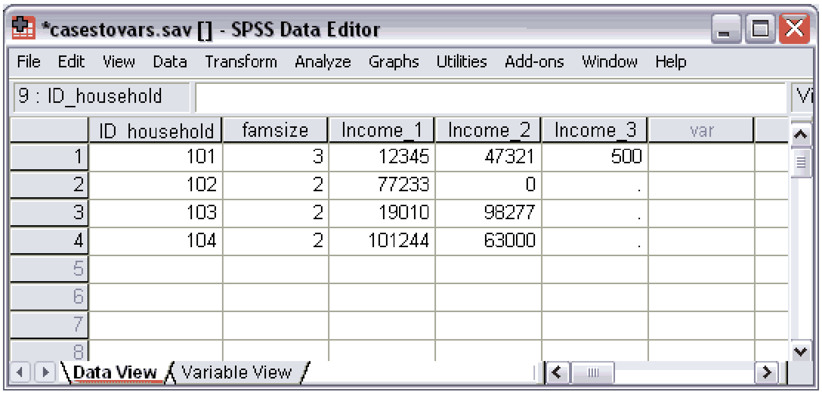
The `CASESTOVARS` command combines the related cases and produces the new variables.

```
*casestovars.sps.
GET FILE = 'c:\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

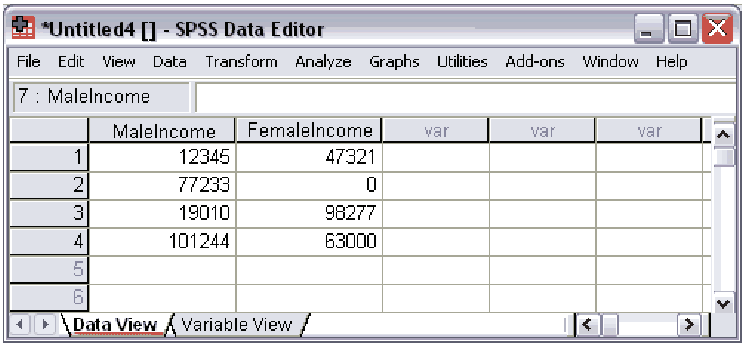
The previous example turned related cases into related variables for use with statistical techniques that compare and contrast related samples. But sometimes you may need to do the exact opposite—convert variables that represent unrelated observations to variables.

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

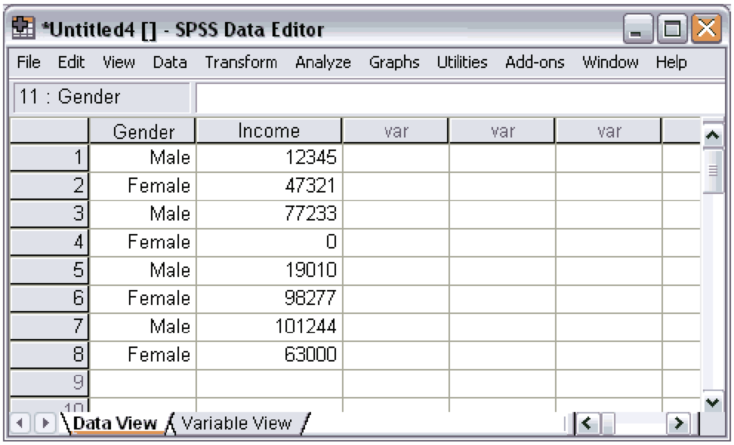


The VARSTOCASES command creates cases from the two columns of data.

```
*varstocases1.sps.
GET DATA /TYPE=XLS
 /FILE = 'c:\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

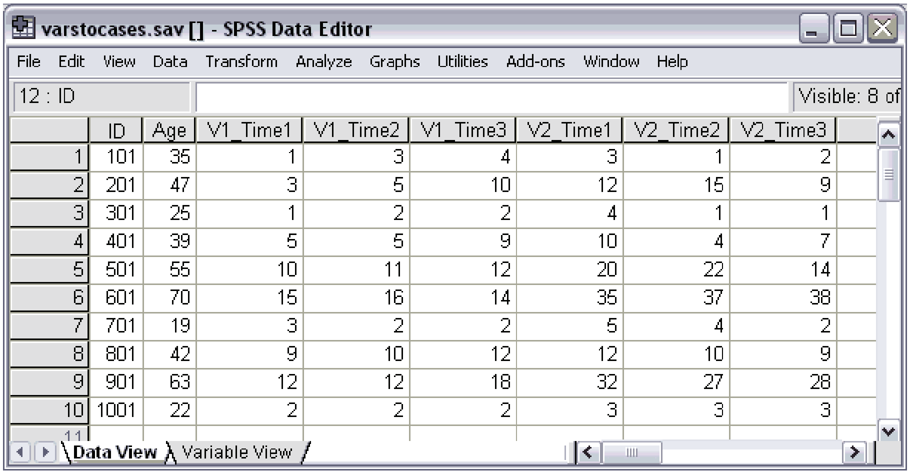


	Gender	Income	var	var	var
1	Male	12345			
2	Female	47321			
3	Male	77233			
4	Female	0			
5	Male	19010			
6	Female	98277			
7	Male	101244			
8	Female	63000			
9					

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 most procedures that compare related observations. However, there is one important exception: linear mixed models (available in the Advanced Statistics add-on module) requires a data structure in which related observations are recorded as separate cases.

Figure 4-16
Related observations recorded as separate variables

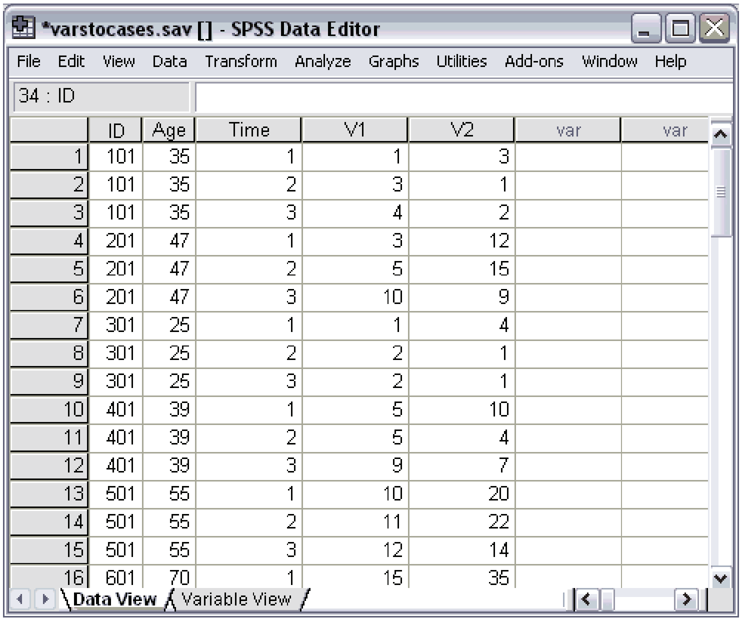


	ID	Age	V1_Time1	V1_Time2	V1_Time3	V2_Time1	V2_Time2	V2_Time3
1	101	35	1	3	4	3	1	2
2	201	47	3	5	10	12	15	9
3	301	25	1	2	2	4	1	1
4	401	39	5	5	9	10	4	7
5	501	55	10	11	12	20	22	14
6	601	70	15	16	14	35	37	38
7	701	19	3	2	2	5	4	2
8	801	42	9	10	12	12	10	9
9	901	63	12	12	18	32	27	28
10	1001	22	2	2	2	3	3	3

```
*varstocases2.sps.
GET FILE = 'c:\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



	ID	Age	Time	V1	V2	var	var
1	101	35	1	1	3		
2	101	35	2	3	1		
3	101	35	3	4	2		
4	201	47	1	3	12		
5	201	47	2	5	15		
6	201	47	3	10	9		
7	301	25	1	1	4		
8	301	25	2	2	1		
9	301	25	3	2	1		
10	401	39	1	5	10		
11	401	39	2	5	4		
12	401	39	3	9	7		
13	501	55	1	10	20		
14	501	55	2	11	22		
15	501	55	3	12	14		
16	601	70	1	15	35		

Chapter

5

Variable and File Properties

In addition to the basic data type (numeric, string, date, etc.), 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 as an SPSS-format data file.

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)
  Religion (NOMINAL).
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 *SPSS 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 an SPSS-format data file, 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 *SPSS 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)
  Religion (NOMINAL).
```

- By default, all new string variables are assigned a nominal measurement level, and all new numeric variables are assigned a scale measurement level. In our example, there is no need to explicitly specify a measurement level for *Interview_date* or *Gender*, since they already have the appropriate measurement levels (scale and nominal, respectively).
- The numeric opinion variables are assigned the ordinal measurement level because there is a meaningful order to the categories.
- The numeric variable *Religion* is assigned the nominal measurement level because there is no meaningful order of religious affiliation. No religion is “higher” or “lower” than another religion.

For many commands, the defined measurement level has no effect on the results. For a few commands, however, the defined measurement level can make a difference in the results and/or available options. These commands include: `GGRAPH`, `IGRAPH`, `XGRAPH`, `CTABLES` (Tables option), and `TREE` (Classification Trees option).

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('').
```

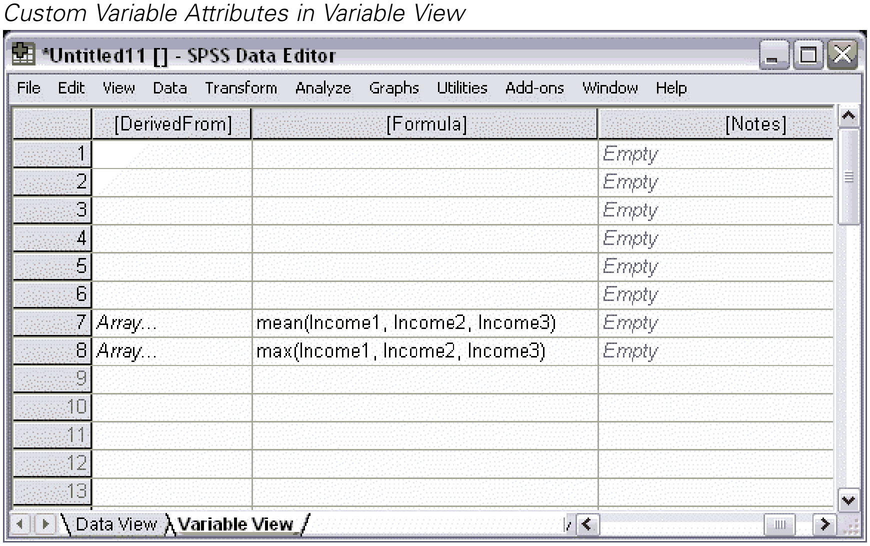
- The attributes *Formula* and *DerivedFrom* are assigned to the two computed variables. Each variable has a different value for *Formula*, which describes the code used to compute the value. For *DerivedFrom*, which lists the variables used to compute the values, both variables have the same attribute 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 SPSS-format 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 'C:\examples\data\variable_properties.sav'
  /SOURCE VARIABLES = Interview_date Age Gender Income_category
  /VARINFO ALL.
APPLY DICTIONARY
  /FROM 'C:\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 SPSS 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='c:\temp\temp.sav'.
NEW FILE.
GET FILE 'c:\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

1[a]	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 Custom Variable Properties on p. 96. Use `DISPLAY ATTRIBUTES` to display custom attribute values.

Chapter

6

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!'.
```

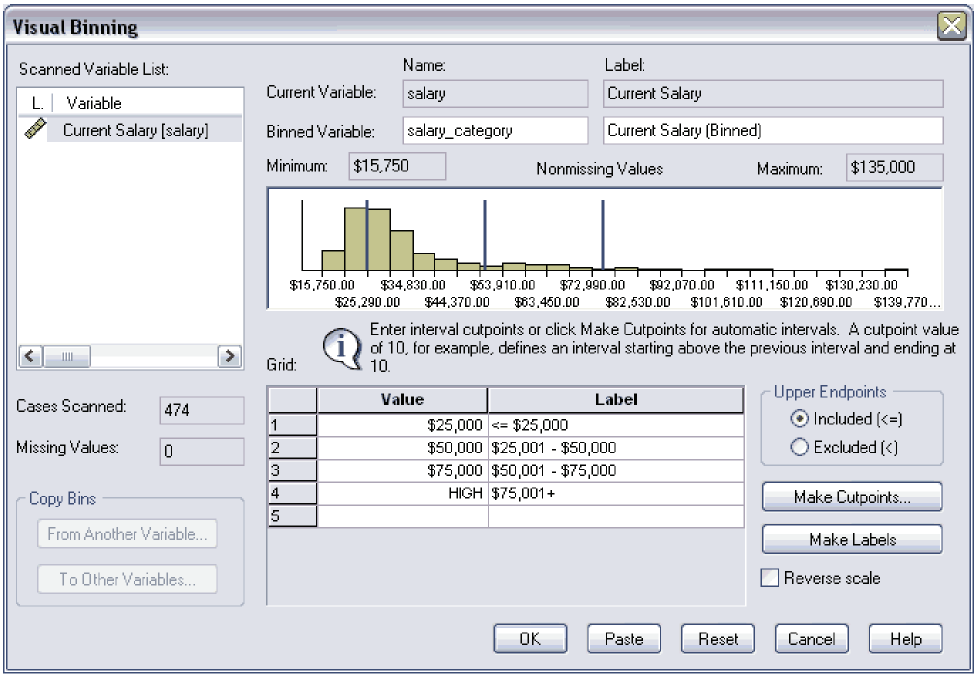
- The RECODE command essentially reverses the values of *opinion2*.

- `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

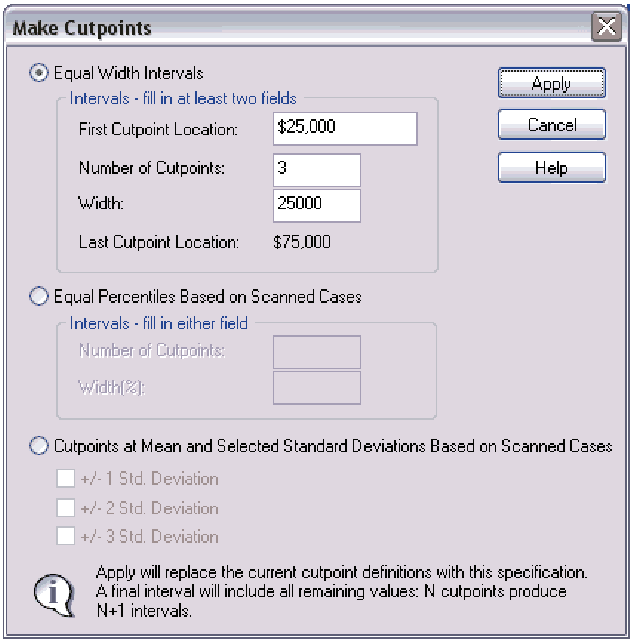
Creating a small number of discrete categories from a continuous scale variable is sometimes referred to as **binning**. For example, you can recode salary data into a few salary range categories. Although it is not difficult to write command syntax to bin a scale variable into range categories, we recommend 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 a number of different binning methods and can automatically generate descriptive labels for the binned categories.

Figure 6-1
Visual Binning



- 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

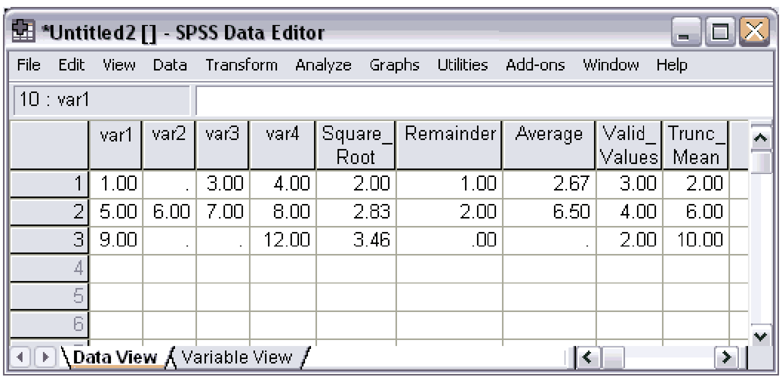
In addition to simple arithmetic operators, you can also transform data with a wide variety of functions, including arithmetic and statistical functions.

```
*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



	var1	var2	var3	var4	Square_Root	Remainder	Average	Valid_Values	Trunc_Mean
1	1.00	.	3.00	4.00	2.00	1.00	2.67	3.00	2.00
2	5.00	6.00	7.00	8.00	2.83	2.00	6.50	4.00	6.00
3	9.00	.	.	12.00	3.46	.00	.	2.00	10.00
4									
5									
6									

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

Random Value and Distribution Functions

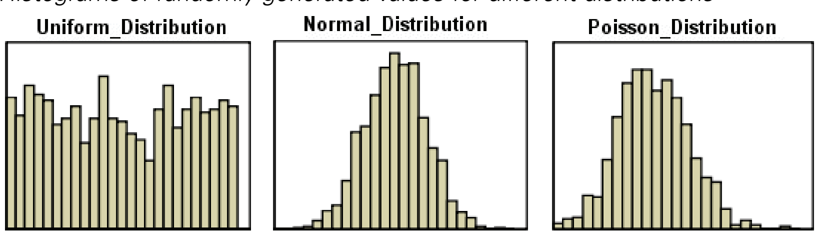
Random value and distribution functions generate random values based on the specified type of distribution and parameters, such as mean, standard deviation, or maximum value.

```
*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 *SPSS 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

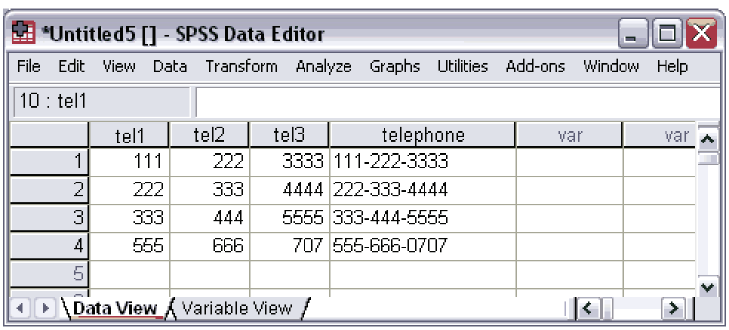
You can combine multiple string and/or numeric values to create new string variables. For example, you could combine three numeric variables for area code, exchange, and number into one string variable for telephone number with dashes between the values.

```
*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

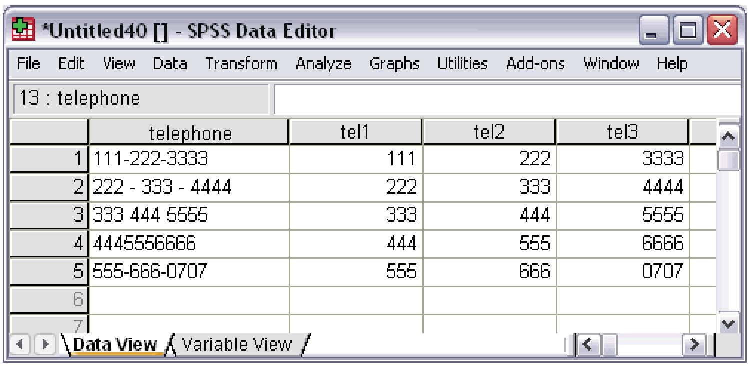
A dataset contains telephone numbers recorded as strings. You want to create separate variables for the three values that comprise the phone number. You know that each number contains 10 digits—but some contain spaces and/or dashes between the three portions of the number, and some do not.

```
*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(SUBSTR(#telstr, 1, 3), F5).
COMPUTE tel2=NUMBER(SUBSTR(#telstr, 4, 3), F5).
COMPUTE tel3=NUMBER(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 `SUBSTR` function. The general form of the function is `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 `SUBSTR("abcd", 2, 2)` would return "bc," and `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*, `SUBSTR(#telstr, 4, 3)` defines a substring three characters long, starting with the fourth character in the original string.
- For *tel3*, `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.
```

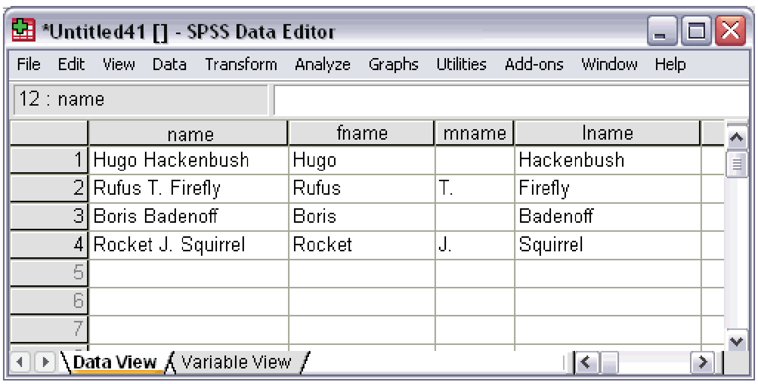
```
LOOP #i = 1 to 2.
- COMPUTE #space = INDEX(#n," ").
- COMPUTE vname(#i) = SUBSTR(#n,1,#space-1).
- COMPUTE #n = SUBSTR(#n,#space+1).
END LOOP.
COMPUTE lname=#n.
DO IF lname="".
- COMPUTE lname=mname.
- COMPUTE mname="".
END IF.
EXECUTE.
```

- 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 = 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) = SUBSTR(#n,1,#space-1)` extracts everything prior to the first dash and sets *fname* to that value.
- `COMPUTE #n = SUBSTR(#n,#space+1)` then sets *#tn* 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 INDEX function



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

SPSS automatically converts date information from databases, Excel files, and SAS files to equivalent SPSS date format variables. SPSS 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	SPSS 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 *SPSS 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 SPSS 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

Original display format	New format specification	New display format
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, SPSS 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.

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 *SPSS 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—and 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

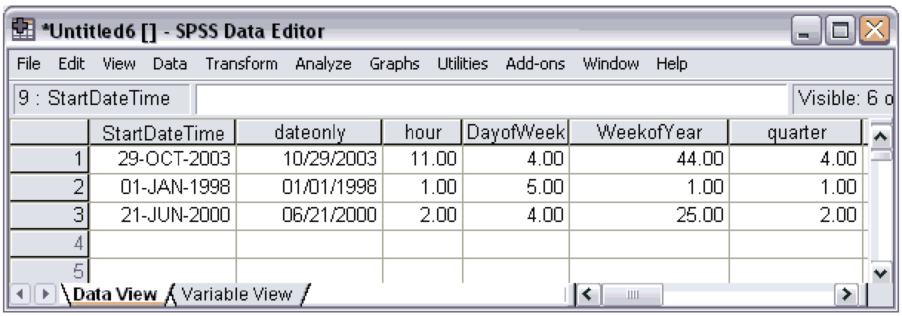
A great deal of information can be extracted from date and time variables. In addition to using `XDATE` functions to extract the more obvious pieces of information, such as year, month, day, hour, and so on, you can obtain information such as day of the week, week of the year, or quarter of the year.

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



- 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 *SPSS Command Syntax Reference*.

Chapter

7

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.

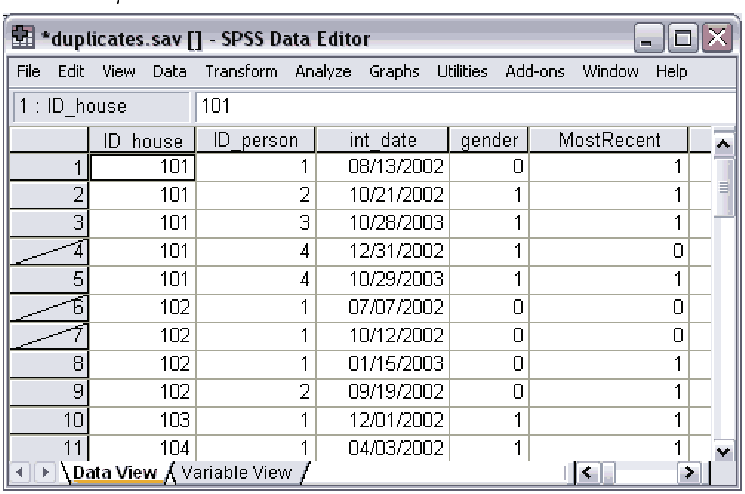
The interview date is recorded in the variable *int_date*, and for cases that match on both ID variables, we want to ignore all but the most recent interview.

```
* duplicates_filter.sps.
GET FILE='c:\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


*duplicates.sav [] - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Add-ons Window Help

1 : ID_house 101

	ID_house	ID_person	int_date	gender	MostRecent
1	101	1	08/13/2002	0	1
2	101	2	10/21/2002	1	1
3	101	3	10/28/2003	1	1
4	101	4	12/31/2002	1	0
5	101	4	10/29/2003	1	1
6	102	1	07/07/2002	0	0
7	102	1	10/12/2002	0	0
8	102	1	01/15/2003	0	1
9	102	2	09/19/2002	0	1
10	103	1	12/01/2002	1	1
11	104	1	04/03/2002	1	1

Data View / Variable 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='c:\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 Validation Option

The Data Validation 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 Validation 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.

Chapter
8
Conditional Processing, Looping, and Repeating

As with other programming languages, SPSS 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 SPSS 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 SPSS 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

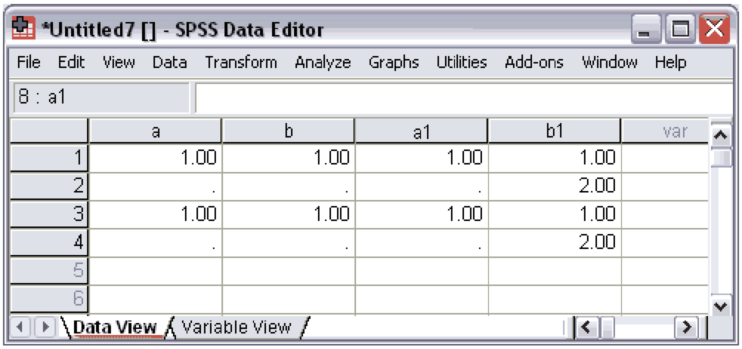
Missing values can affect the results from `DO IF` structures because if the expression evaluates to missing, then control passes immediately to the `END IF` command at that point. To avoid this type of problem, you should attempt to deal with missing values first in the `DO IF` structure before evaluating 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.
```

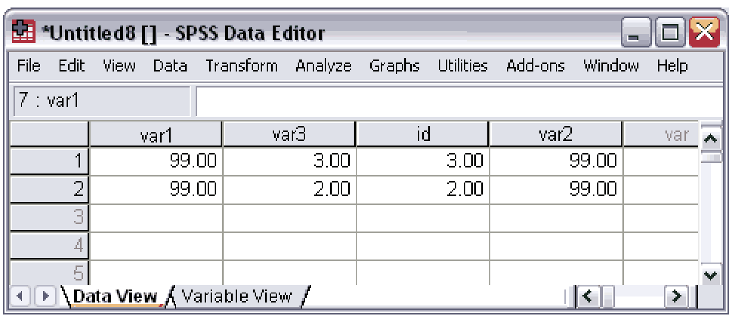
- The transformation commands inside the DO REPEAT structure are repeated for each variable or value assigned to the stand-in variable.
- Multiple stand-in variables and values can be specified in the same DO REPEAT structure by preceding each additional specification with a forward slash.
- The optional 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 Vectors on p. 141.

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.

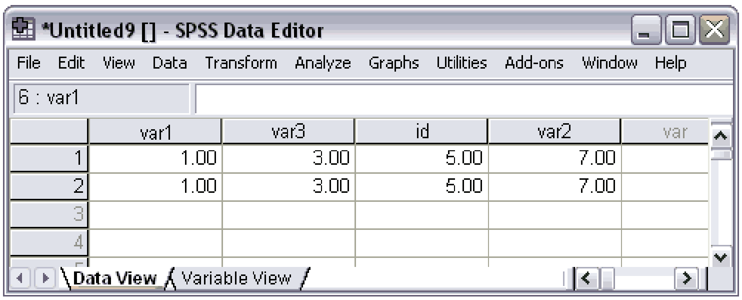
Example

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***.
```

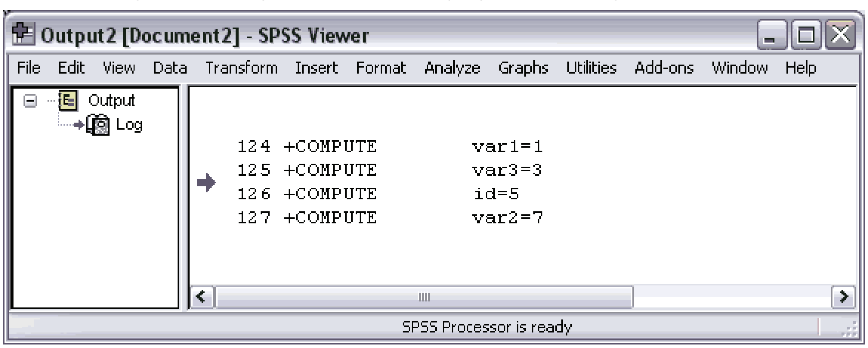
```
DO REPEAT v=var1 TO var2 /val=1 3 5 7.
- COMPUTE v=val.
END REPEAT PRINT.
EXECUTE.
```

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

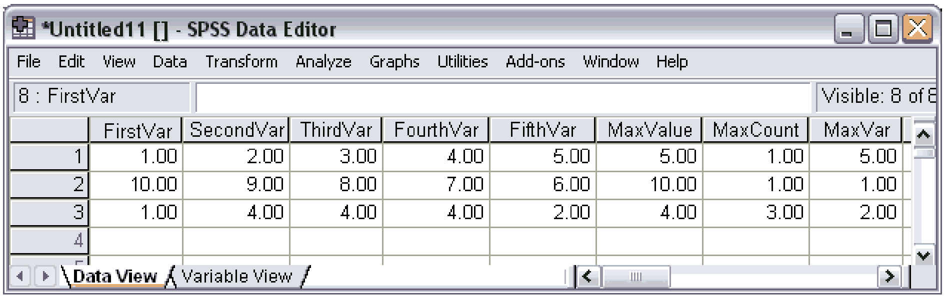
You can use the MAX function to find the highest value among a specified set of variables. But what if you also want to know which variable has that value—and if more than one variable has that value, how many variables have that value? Using VECTOR and LOOP, you can get the information you want.

```
*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 SPSS 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



	FirstVar	SecondVar	ThirdVar	FourthVar	FifthVar	MaxValue	MaxCount	MaxVar
1	1.00	2.00	3.00	4.00	5.00	5.00	1.00	5.00
2	10.00	9.00	8.00	7.00	6.00	10.00	1.00	1.00
3	1.00	4.00	4.00	4.00	2.00	4.00	3.00	2.00
4								

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='c:\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. 141 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. 150 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.

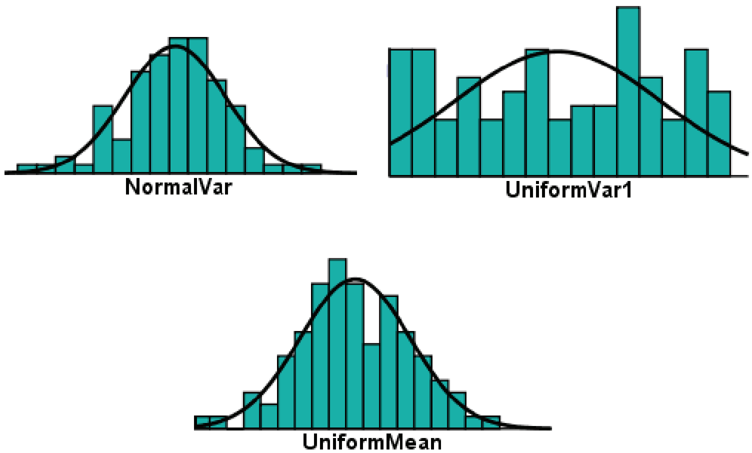
We can use an input program and nested loops to demonstrate the validity of the Central Limit Theorem. For this example, we'll assume that a sample size of 100 is "sufficiently large."

```
*loop_nested.sps.
NEW FILE.
SET 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)=UNIFORM(1000).
-  END LOOP.
-  END CASE.
- END LOOP.
- END FILE.
END INPUT PROGRAM.
COMPUTE UniformMean=mean(UniformVar1 TO UniformVar100).
COMPUTE NormalVar=500+NORMAL(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 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 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 SPSS 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="c:\temp\tempdata.sav"
    /KEEP Age Gender.
END LOOP.
```

```
EXECUTE.
GET FILE='c:\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

Chapter

9

Exporting Data and Results

You can export and save both data and results in a variety of formats for use by other applications, including:

- Save data in SAS, Stata, Excel, and text format.
- Write data to a database.
- Export results in HTML, Word, Excel, and text format.
- Save results in XML and SPSS data file (*.sav*) format.

Output Management System

The Output Management System provides the ability to automatically write selected categories of output to different output files in different formats. Formats include:

SPSS data file format (SAV). Output that would be displayed in pivot tables in the Viewer can be written out in the form of an SPSS data file, making it possible to use output as input for subsequent commands.

XML. Tables, text output, and even many charts can be written out in XML format.

HTML. Tables and text output can be written out in HTML format. Standard (not interactive) charts and tree model diagrams (Classification Tree option) can be included as image files.

Text. Tables and text output can be written out as tab-delimited or space-separated text.

The examples provided here are also described in the SPSS Help system, and they barely scratch the surface of what is possible with the `OMS` command. For a detailed description of the `OMS` command and related commands (`OMSEND`, `OMSINFO`, and `OMSLOG`), see the *SPSS Command Syntax Reference*.

Using Output as Input with OMS

Using the OMS command, you can save pivot table output to SPSS-format 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 in the *tutorial\sample_files* folder of the SPSS installation folder.

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.

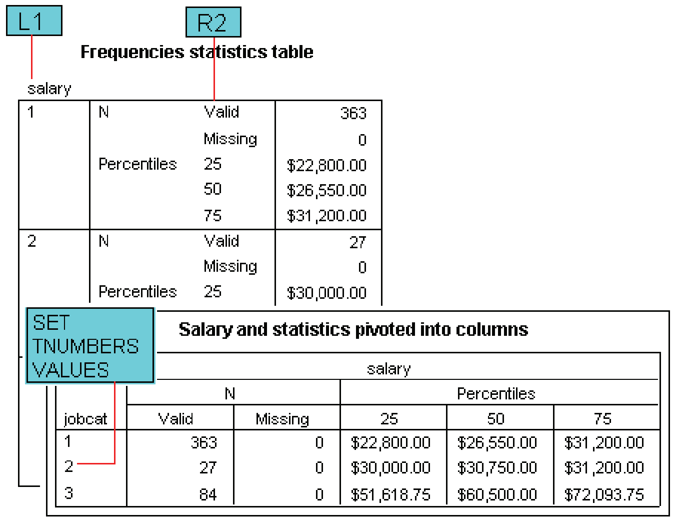
The command syntax used in this example is *oms_percentiles.sps*, located in the *tutorial\sample_files* folder of the SPSS installation folder.

```
***oms_percentiles.sps***.
GET
  FILE='c:\Program Files\spss\Employee data.sav'.
PRESERVE.
SET TVARS NAMES TNUMBERS VALUES.

***split file by job category to get group percentiles.
SORT CASES BY jobcat.
SPLIT FILE LAYERED BY jobcat.

DATASET DECLARE tempdata.

OMS
  /SELECT TABLES
  /IF COMMANDS=['Frequencies'] SUBTYPES=['Statistics']
  /DESTINATION FORMAT=SAV
   OUTFILE=tempdata
  /COLUMNS SEQUENCE=[L1 R2].
```

```
FREQUENCIES
  VARIABLES=salary
  /FORMAT=NOTABLE
  /PERCENTILES= 25 50 75.

OMSEND.

***restore previous  SET settings.
RESTORE.

MATCH FILES FILE=*
  /TABLE=tempdata
  /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 an SPSS-format 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 9-1
Default and pivoted statistics table



Frequencies statistics table

salary

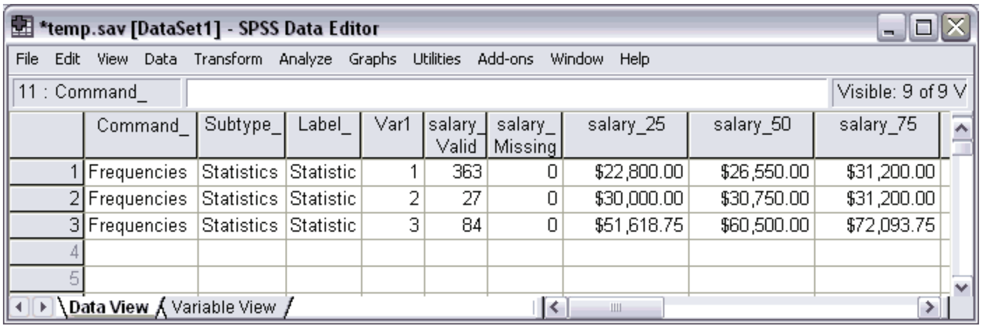
1	N	Valid	363
		Missing	0
	Percentiles	25	$22,800.00
		50	$26,550.00
		75	$31,200.00
2	N	Valid	27
		Missing	0
	Percentiles	25	$30,000.00

Salary and statistics pivoted into columns

	salary				
	N		Percentiles		
jobcat	Valid	Missing	25	50	75
1	363	0	$22,800.00	$26,550.00	$31,200.00
2	27	0	$30,000.00	$30,750.00	$31,200.00
3	84	0	$51,618.75	$60,500.00	$72,093.75

- 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 9-2
Data file created from pivoted table

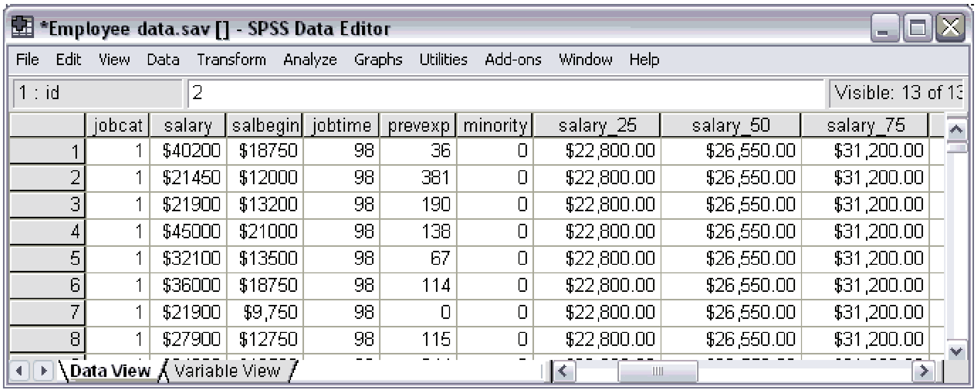


	Command_	Subtype_	Label_	Var1	salary_Valid	salary_Missing	salary_25	salary_50	salary_75
1	Frequencies	Statistics	Statistic	1	363	0	$22,800.00	$26,550.00	$31,200.00
2	Frequencies	Statistics	Statistic	2	27	0	$30,000.00	$30,750.00	$31,200.00
3	Frequencies	Statistics	Statistic	3	84	0	$51,618.75	$60,500.00	$72,093.75
4									
5									

- The `MATCH FILES` command merges the contents of the data file created from the statistics table with the original data file. 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=tempdata` identifies the data file 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 9-3
Percentiles added to original data file



	jobcat	salary	salbegin	jobtime	prevexp	minority	salary_25	salary_50	salary_75
1	1	$40200	$18750	98	36	0	$22,800.00	$26,550.00	$31,200.00
2	1	$21450	$12000	98	381	0	$22,800.00	$26,550.00	$31,200.00
3	1	$21900	$13200	98	190	0	$22,800.00	$26,550.00	$31,200.00
4	1	$45000	$21000	98	138	0	$22,800.00	$26,550.00	$31,200.00
5	1	$32100	$13500	98	67	0	$22,800.00	$26,550.00	$31,200.00
6	1	$36000	$18750	98	114	0	$22,800.00	$26,550.00	$31,200.00
7	1	$21900	$9,750	98	0	0	$22,800.00	$26,550.00	$31,200.00
8	1	$27900	$12750	98	115	0	$22,800.00	$26,550.00	$31,200.00

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*, located in the *tutorial\sample_files* folder of the SPSS installation folder.

OMS Commands to Create a Data File of Coefficients

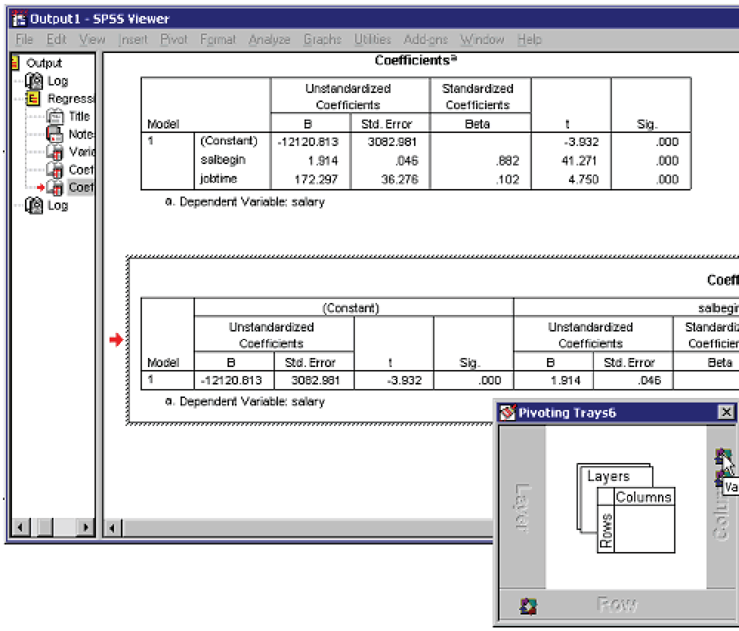
Although the command syntax file *oms_bootstrapping.sps* may seem long and/or complicated, the OMS commands that create the data file of sample regression coefficients are really very short and simple:

```
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 9-4

Variables dimension element pivoted into column dimension



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='D:\Program Files\SPSS\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 SPSS 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.

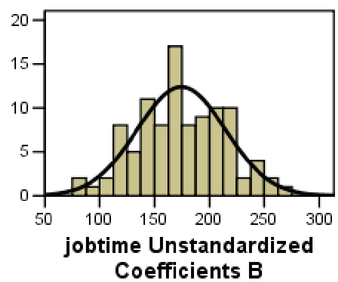
```
OMSEND.
DATASET ACTIVATE bootstrap_example.
FREQUENCIES
  VARIABLES=salbegin_B salbegin_Beta jobtime_B jobtime_Beta
  /FORMAT NOTABLE
  /PERCENTILES= 2.5 97.5
  /HISTOGRAM  NORMAL.
RESTORE.
```

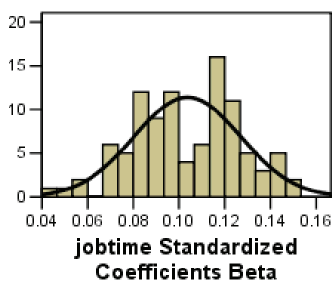
- `OMSEND` without any additional specifications ends all active `OMS` requests. In this example, there were two: one to suppress all Viewer output and one to save regression coefficients in a data file. If you don't end both `OMS` requests, either you won't be able to open the data file or you won't see any results of your subsequent analysis.
- The job ends with a `RESTORE` command that restores your previous `SET` specifications.

Figure 9-5
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





Transforming OXML with XSLT

Using the `OMS` command, you can route output to OXML, which is XML that conforms to the SPSS 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 in the *tutorial\sample_files* folder of the SPSS installation folder.
- The SPSS Output XML schema is documented in *SPSSOutputXML_schema.htm*, located in the *help\main* folder of the SPSS installation folder.

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 9-6
Viewer outline

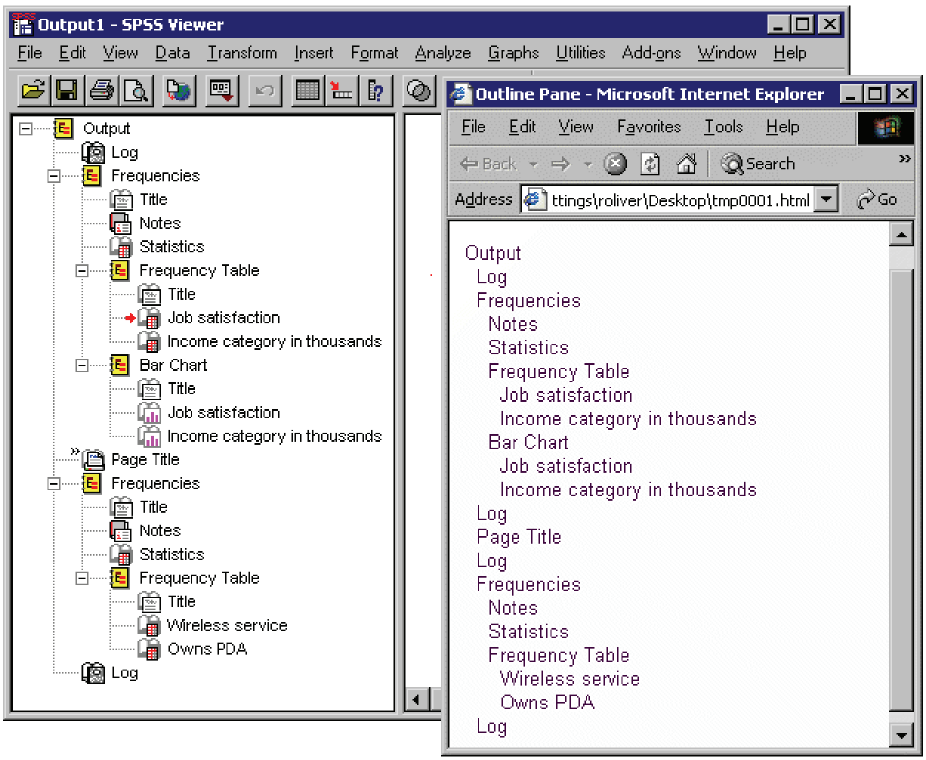


Figure 9-7
XSLT stylesheet oms_simple_outline_example.xsl

```
<?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 SPSS 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 9-8
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 9-9
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 9-10
XSLT stylesheet: oms_simple_frequency_tables.xsl

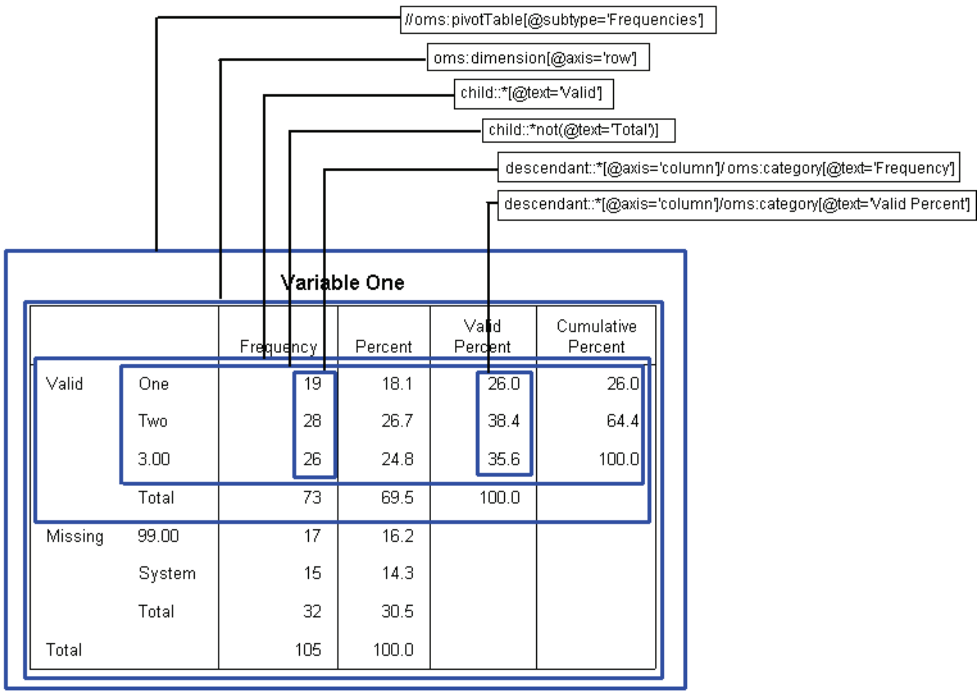
```
<?xml version="1.0" encoding="UTF-8"?>

<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 Advanced xsl:for-each "Pull" Example on p. 174.
- <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 9-11
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 9-12
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 9-13
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 Controlling Variable and Value Label Display on p. 178.
- <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 also 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 Positional Arguments versus Localized Text Attributes on p. 180.
- <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 Controlling Variable and Value Label Display on p. 178.
- <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 Controlling Decimal Display on p. 179.

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`—and 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 9-14
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 9-15
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.

Positional Arguments versus Localized Text Attributes

Whenever possible, it is always best to avoid XPath expressions that rely on localized text (text that differs for different output languages) or positional arguments. You will probably find, however, that this is not always possible.

Localized 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. If you do not need XSLT that will work in multiple languages, XPath expressions that select elements based on `text` attributes (for example, `@text='Frequency'`) will often provide a simple, reliable solution.

Positional Arguments

Instead of localized `text` attributes, 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 9-16
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		

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 SPSS variable names are not valid in SAS, such as @, #, and $. These illegal characters are replaced with an underscore when the data are exported.
- SPSS variable labels containing more than 40 characters are truncated when exported to a SAS v6 file.
- Where they exist, SPSS variable labels are mapped to the SAS variable labels. If no variable label exists in the SPSS data, the variable name is mapped to the SAS variable label.
- SAS allows only one value for missing, whereas SPSS allows the definition of numerous missing values. As a result, all missing values in SPSS are mapped to a single missing value in the SAS file.

Example

```
*save_as_SAS.sps.
GET FILE='c:\examples\data\employee data.sav'.
SAVE TRANSLATE OUTFILE='c:\examples\data\sas7datafile.sas7bdat'
  /TYPE=SAS  /VERSION=7  /PLATFORM=WINDOWS
  /VALFILE='c:\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 format file. Unlike SPSS, SAS variable and value labels are not saved with the data; they are stored in a separate file.

For more information, see the SAVE TRANSLATE command in the *SPSS 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='c:\examples\data\employee data.sav'.
SAVE TRANSLATE
  OUTFILE='c:\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. For string variables, value labels are dropped.
- 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.
- SPSS 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 Variable Type	Stata Variable Type	Stata Data Format
Numeric	Numeric	g
Comma	Numeric	g

SPSS Variable Type	Stata Variable Type	Stata Data Format
Dot	Numeric	g
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='c:\examples\data\employee data.sav'.
SAVE TRANSLATE OUTFILE='c:\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. 22 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 could 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 SPSS 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 *SPSS Command Syntax Reference* for more information.

Exporting Results to PDF, Word, Excel, and PowerPoint

The `OMS` command (discussed earlier in this chapter) is the method of choice for exporting results in XML, HTML, or text format, but `OMS` is not appropriate if you want to export results to PDF, Microsoft Word, Excel, or PowerPoint.

To export results to PDF, Word, Excel, or PowerPoint, you need to use the Export facility in the Viewer. From the Viewer window menus, choose:

File
  Export

For detailed examples, see the tutorials installed with SPSS. From the menus, choose:

Help
  Tutorial

In the Tutorial table of contents, choose:

Working with Output
  Using the Viewer
    Using Results in Other Applications

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 introduced in SPSS 15.0.

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='c:\temp\group1.spo'.
OUTPUT NEW NAME=group2.
COMPUTE filterVar=(GroupVar=2).
FILTER BY filterVar.
FREQUENCIES VARIABLES=SummaryVar.
OUTPUT SAVE OUTFILE='c:\temp\group2.spo'.
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.

Chapter

10

Scoring Data with Predictive Models

Introduction

The process of applying a predictive model to a set of data is referred to as **scoring** the data. A typical example is credit scoring, where a credit application is rated for risk based on various aspects of the applicant and the loan in question.

SPSS, Clementine, and AnswerTree have 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 as an XML file containing all of the information necessary to reconstruct the model. The SPSS Server product then provides the means to read an XML model file and apply the model to a data file.

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 variables—the predictor variables 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.

It is often the case that you need to apply transformations to your original data before building your model and that the same transformations will have to be applied to the data you need to score. You can apply those transformations first, followed by the transformations that score the data. The whole process, starting from raw data to predicted results, is then seen as a set of data transformations. The advantage to this unified approach is that all of the transformations can be processed with a single data pass. In fact, you can score the same data file with multiple models—each providing its own set of results—with just a single data pass. For large data files, this can translate into a substantial savings in computing time.

Scoring is available only with SPSS Server and is a task that can be done with SPSS command syntax. The necessary commands can be entered into a Syntax Editor window and run interactively by users working in distributed analysis mode. The set of commands can also be saved in a command syntax file and submitted to the SPSS Batch Facility, a separate executable version of SPSS provided with SPSS Server. For large data files, you will probably want to make use of the SPSS Batch Facility. For information about distributed analysis mode, see the *SPSS Base User's Guide*. For information about using the SPSS Batch Facility, see the *SPSS Batch Facility User's Guide*, provided as a PDF document on the SPSS Server product CD.

Basics of Scoring Data

Transforming Your Data

In order to build the best model, you may need to transform one or more variables. Assuming that your input data have the same structure as that used to build your model, you would need to perform these same transformations on the input data. This is easily accomplished by exporting the transformations to an external file in PMML format—specifically, PMML 3.1 with SPSS extensions—and then merging them with your model specification file. When you apply the model to the data, the transformations will be automatically applied before scoring the data. The transformations are carried out as part of the internal scoring process and have no effect on the active dataset.

To export a set of transformations, you include them in a `TMS BEGIN-TMS END` block in command syntax, and you run this command syntax on a dataset that contains the variables to be transformed, which will often be the dataset used to build the model.

```
TMS BEGIN
    /DESTINATION OUTFILE='file specification'.

COMPUTE var_new = ln(var).

TMS END.
```

- `TMS BEGIN` marks the beginning of a block of transformation commands that will be evaluated for export. The `DESTINATION` subcommand specifies the file where the transformations will be exported (include the file extension *xml*). In this case, the block contains a single log transformation.
- `TMS END` marks the end of the block and causes the destination file to be written but has no effect on the state of the transformations contained in the block. In the present example, the transformation to create *var_new* is pending after the completion of the block.

Merging Transformations and Model Specifications

Once a predictive model has been built and the necessary transformations have been exported, you merge the transformations with the model. This is done using the `TMS MERGE` command.

```
TMS MERGE
    /DESTINATION OUTFILE='file specification'
    /TRANSFORMATIONS INFILE='file specification'
    /MODEL INFILE='file specification'.
```

- The `DESTINATION` subcommand specifies the file that will contain both the transformations and the specifications for a given model (include the file extension *xml*). This is the file you will use to score your data.
- The `TRANSFORMATIONS` subcommand specifies the file containing the exported data transformations—that is, the destination file specified on the `TMS BEGIN` command.
- The `MODEL` subcommand specifies the file containing the model specifications.

Command Syntax for Scoring

Scoring can be done through the use of command syntax. The sample syntax in this example contains all of the essential elements needed to score data.

```
*Get data to be scored.
GET FILE='\samples\data\sample.sav'.

*Read in the XML model file.
MODEL HANDLE NAME=cluster_mod FILE='\samples\data\cmod.xml'.

*Apply the model to the data.
COMPUTE PredRes = ApplyModel(cluster_mod,'predict').

*Read the data.
EXECUTE.
```

- The command used to get the input data depends on the form of the data. For example, if your data are in SPSS format, you'll use the `GET` command, but if your data are stored in a database, you'll use the `GET DATA` command. For details, see the *SPSS Command Syntax Reference*, accessible as a PDF file from the Help menu. In the current example, the data are in SPSS format and are assumed to be in a file named *sample.sav*, located in the *samples\data* folder on the computer on which SPSS Server is installed. SPSS Server expects that file paths, specified as part of command syntax, are relative to the computer on which SPSS Server is installed.
- The `MODEL HANDLE` command is used to read the XML file containing the model specifications and any associated data transformations. It caches the model specifications and associates a unique name with the cached model. In the current example, the model is assigned the name *cluster_mod*, and the model specifications are assumed to be in a file named *cmod.xml*, located in the *samples\data* folder on the server computer.
- The `ApplyModel` function is used with the `COMPUTE` command to apply the model. `ApplyModel` has two arguments: the first identifies the model using the name defined on the `MODEL HANDLE` command, and the second identifies the type of result to be returned, such as the model prediction (as in this example) or the probability associated with the prediction. For details on the `ApplyModel` function, including the types of results available for each model type, see "Scoring Expressions" in the "Transformation Expressions" section of the *SPSS Command Syntax Reference*.
- In this example, the `EXECUTE` command is used to read the data. The use of `EXECUTE` is not necessary if you have subsequent commands that read the data, such as `SAVE`, or any statistical or charting procedure.

After scoring, the active dataset contains the results of the predictions—in this case, the new variable *PredRes*. If your data were read in from a database, you'll probably want to write the results back to the database. This is accomplished with the SAVE TRANSLATE command (for details, see the *SPSS Command Syntax Reference*).

Mapping Model Variables to SPSS Variables

You can map any or all of the variables specified in the XML model file to different variables in the current active dataset. By default, the model is applied to variables in the current active dataset with the same names as the variables in the model file. The MAP subcommand of a MODEL HANDLE command is used to map variables.

```
MODEL HANDLE NAME=cluster_mod FILE='C:\samples\data\cmod.xml'
      /MAP VARIABLES=Age_Group Log_Amount MODELVARIABLES=AgeGrp LAmt.
```

In this example, the model variables *AgeGrp* and *LAmt* are mapped to the variables *Age_Group* and *Log_Amount* in the active dataset.

Missing Values in Scoring

A missing value in the context of scoring refers to one of the following: a predictor variable with no value (system-missing for numeric variables, a null string for string variables), a value defined as user-missing in the model, or a value for a categorical predictor variable that is not one of the categories defined in the model. Other than the case where a predictor variable has no value, the identification of a missing value is based on the specifications in the XML model file, not those from the variable properties in the active dataset. This means that values defined as user-missing in the active dataset but not as user-missing in the XML model file will be treated as valid data during scoring.

By default, the scoring facility attempts to substitute a meaningful value for a missing value. The precise way in which this is done is model dependent. For details, see the MODEL HANDLE command in the *SPSS Command Syntax Reference*. If a substitute value cannot be supplied, the value for the variable in question is set to system-missing. Cases with values of system-missing, for any of the model's predictor variables, give rise to a result of system-missing for the model prediction.

You have the option of suppressing value substitution and simply treating all missing values as system-missing. Treatment of missing values is controlled through the value of the `MISSING` keyword on the `OPTIONS` subcommand of a `MODEL HANDLE` command.

```
MODEL HANDLE NAME=cluster_mod FILE='C:\samples\data\cmod.xml'
      /OPTIONS MISSING=SYSMIS.
```

In this example, the keyword `MISSING` has the value `SYSMIS`. Missing values encountered during scoring will then be treated as system-missing. The associated cases will be assigned a value of system-missing for a predicted result.

Using Predictive Modeling to Identify Potential Customers

A marketing company is tasked with running a promotional campaign for a suite of products. The company has already targeted a regional base of customers and has sufficient information to build a model for predicting customer response to the campaign. The model is then to be applied to a much larger set of potential customers in order to determine those most likely to make purchases as a result of the promotion.

This example makes use of the information in the following data files: *customers_model.sav*, which contains the data from the individuals who have already been targeted, and *customers_new.sav*, which contains the list of potentially new customers. All sample files for this example are located in the *tutorial\sample_files* folder of the SPSS installation folder. If you are working in distributed analysis mode (not required for this example), you'll need to copy *customers_model.sav* to the computer on which SPSS Server is installed.

Building and Saving Predictive Models

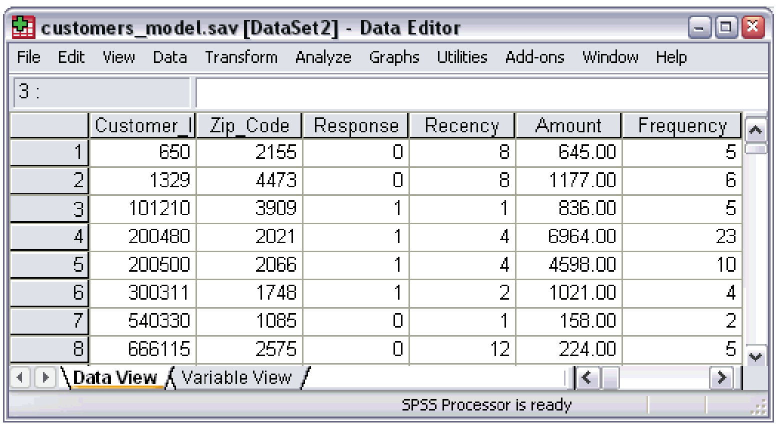
The first task is to build a model for predicting whether or not a potential customer will respond to a promotional campaign. The result of the prediction, then, is either yes or no. In the language of predictive models, the prediction is referred to as the **target variable**. In the present case, the target variable is categorical since there are only two possible values of the result.

Choosing the best predictive model is a subject unto itself. The goal here is simply to lead you through the steps to build a model and save the model specifications. Two models that are appropriate for categorical target variables, a multinomial logistic regression model and a classification tree model, will be considered.

▶ If you haven't already done so, open *customers_model.sav*.

The method used to retrieve your data depends on the form of the data. In the common case that your data are in a database, you'll want to make use of the built-in features for reading from databases. For details, see the *SPSS Base User's Guide*.

Figure 10-1
Data Editor window



	Customer_I	Zip_Code	Response	Recency	Amount	Frequency
1	650	2155	0	8	645.00	5
2	1329	4473	0	8	1177.00	6
3	101210	3909	1	1	836.00	5
4	200480	2021	1	4	6964.00	23
5	200500	2066	1	4	4598.00	10
6	300311	1748	1	2	1021.00	4
7	540330	1085	0	1	158.00	2
8	666115	2575	0	12	224.00	5

The Data Editor window should now be populated with the sample data you'll use to build your models. Each case represents the information for a single individual. The data include demographic information, a summary of purchasing history, and whether or not each individual responded to the regional campaign.

Transforming Your Data

In an ideal situation, your raw data are perfectly suitable for the type of analysis you want to perform. Unfortunately, this is rarely the case. Preliminary analysis may reveal inconvenient coding schemes for categorical variables or the need to apply numeric transformations to scale variables. Any transformations applied to the data used to build the model will also usually need to be applied to the data that are to be scored. This is easily accomplished by exporting the transformations to an external file in PMML format—specifically, PMML 3.1 with SPSS extensions—and then merging them with the model specification file. When you apply the model to the data, the transformations will be automatically applied before scoring the data. The transformations are carried out as part of the internal scoring process and have no effect on the active dataset.

To export a set of transformations, you include them in a `TMS BEGIN-TMS END` block in command syntax. The file *scoring_transformations.sps*, located in the *tutorial\sample_files* folder of the local SPSS installation folder, contains a `TMS BEGIN-TMS END` block with the few simple transformations of the raw data used to obtain the file *customers_model.sav*—the file that will be used for modeling.

```
TMS BEGIN
    /DESTINATION OUTFILE='file specification'.

* Recode Age into a categorical variable.
RECODE Age
     ( MISSING = -9 )
     ( LO THRU 37 =1 )
     ( LO THRU 43 =2 )
     ( LO THRU 49 =3 )
     ( LO THRU HI = 4 ) INTO Age_Group.

* The Amount distribution is skewed, so take the log of it.
COMPUTE Log_Amount = ln(Amount).

TMS END.
```

- `TMS BEGIN` marks the beginning of a block of transformation commands that will be evaluated for export. The `DESTINATION` subcommand specifies the file where the transformations will be exported.
- The existing values of *Age* are consolidated into five categories and stored in the new variable *Age_Group*.
- A histogram of *Amount* would show that the distribution is skewed. This is something that is often cured by a log transformation, as done here.
- `TMS END` marks the end of the block and causes the destination file to be written but has no effect on the state of the transformations contained in the block. In the present example, the transformations to create *Age_Group* and *Log_Amount* are pending after the completion of the block.

▶ If you haven't already, open *scoring_transformations.sps*.

▶ Enter a file location in place of `'file specification'` on the `DESTINATION` subcommand, and include the file extension *xml*.

- Highlight the `TMS BEGIN-TMS END` block.
- From the menus in the SPSS Syntax Editor window, choose:

Run
  Selection

Notice that we ran the `TMS BEGIN-TMS END` command syntax on *customers_model.sav*. In general, you need to run `TMS BEGIN-TMS END` on a dataset that contains the variables to be transformed, which will often be the dataset used to build the model.

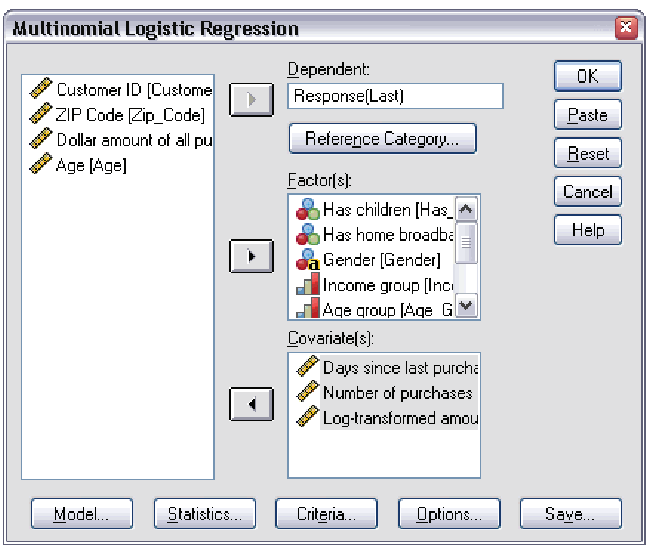
Building and Saving a Multinomial Logistic Regression Model

To build a Multinomial Logistic Regression model (requires the Regression Models option):

- From the menus, choose:

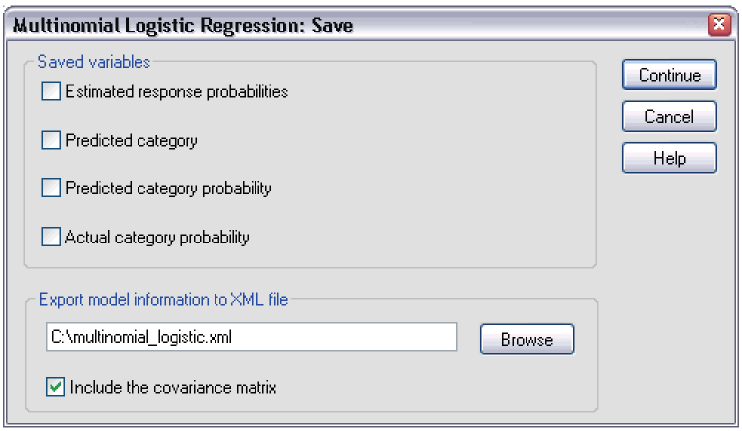
Analyze
  Regression
    Multinomial Logistic...

Figure 10-2
Multinomial Logistic Regression dialog box



- Select *Response* for the dependent variable.
- Select *Has_Child*, *Has_Broadband*, *Gender*, *Income_Group*, and *Age_Group* for the factors.
- Select *Recency*, *Frequency*, and *Log_Amount* for the covariates.
- Click Save.

Figure 10-3
Multinomial Logistic Regression Save dialog box



- Click the Browse button in the Multinomial Logistic Regression Save dialog box.

  This will take you to a standard dialog box for saving a file.

- Navigate to the directory in which you would like to save the XML model file, enter a filename, and click Save.

  Note: If you're licensed for SPSS Adaptor for Predictive Enterprise Services, you can store the model file to a repository by clicking Store File To Predictive Enterprise Repository in the Save dialog box.

  The path to your chosen file should now appear in the Multinomial Logistic Regression Save dialog box. You'll eventually include this path as part of the command syntax file for scoring. For purposes of scoring, paths in syntax files are interpreted relative to the computer on which SPSS Server is installed.

- Click Continue in the Multinomial Logistic Regression Save dialog box.
- Click OK in the Multinomial Logistic Regression dialog box.

This results in creating the model and saving the model specifications as an XML file. For convenience, the command syntax for creating this model and saving the model specifications is included in the section labeled *Multinomial logistic regression model* in the file *scoring_models.sps*.

Building and Saving a Classification Tree Model

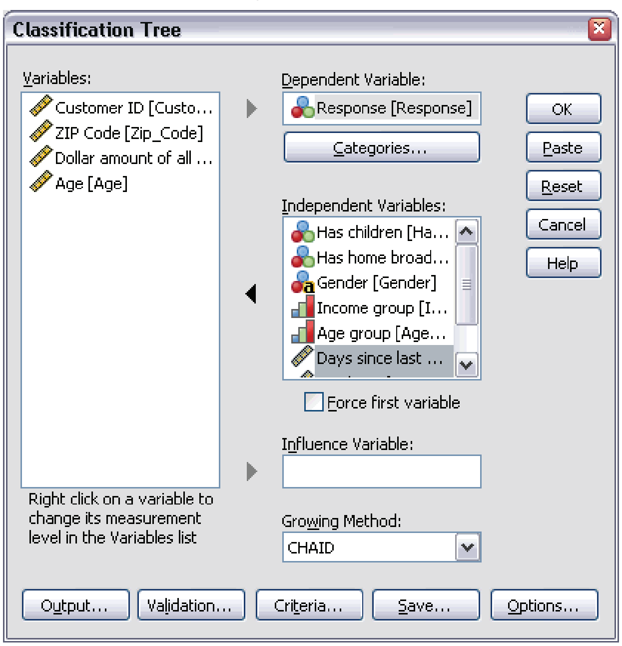
The Tree procedure, available in the Classification Tree option (not included with the Base system), provides a number of methods for growing a classification tree. The default method is CHAID and is sufficient for the present purposes.

To build a CHAID tree model:

▶ From the menus, choose:

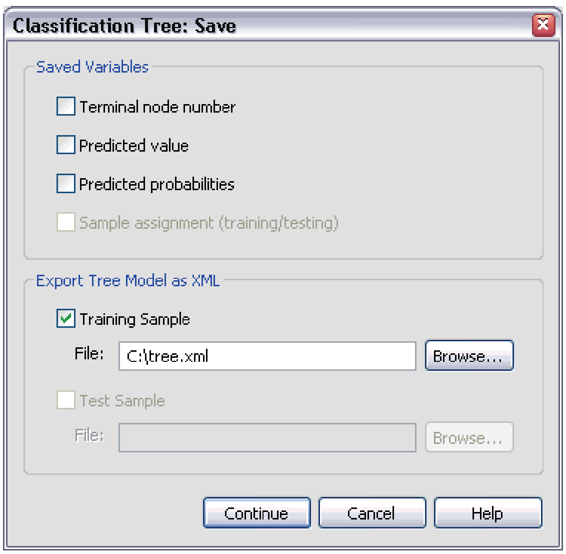
Analyze
  Classify
    Tree...

Figure 10-4
Classification Tree dialog box



- ▶ Select *Response* for the dependent variable.
- ▶ Select *Has_Child*, *Has_Broadband*, *Gender*, *Income_Group*, *Age_Group*, *Log_Amount*, *Recency*, and *Frequency* for the independent variables.
- ▶ Click Save.

Figure 10-5
Classification Tree Save dialog box



- ▶ Select Training Sample in the Export Tree Model as XML group.
- ▶ Click the Browse button.

  This will take you to a standard dialog box for saving a file.
- ▶ Navigate to the directory in which you would like to save the XML model file, enter a filename, and click Save.

  The path to your chosen file should now appear in the Classification Tree Save dialog box.
- ▶ Click Continue in the Classification Tree Save dialog box.
- ▶ Click OK in the Classification Tree dialog box.

This results in creating the model and saving the model specifications as an XML file. For convenience, the command syntax for creating this model and saving the model specifications is included in the section labeled *Classification tree model* in the file *scoring_models.sps*.

Merging Transformations and Model Specifications

You've built your models and saved them. It's now time to merge the transformations you saved earlier with the model specifications. You merge transformations and model specifications using the `TMS MERGE` command. The file *scoring_transformations.sps*, located in the *tutorial\sample_files* folder of the local SPSS installation folder, contains a template for `TMS MERGE`.

```
TMS MERGE
    /DESTINATION OUTFILE='file specification'
    /TRANSFORMATIONS INFILE='file specification'
    /MODEL INFILE='file specification'.
```

- The `DESTINATION` subcommand specifies the file that will contain both the transformations and the specifications for a given model. This is the file you will use to score your data.
- The `TRANSFORMATIONS` subcommand specifies the file containing the exported data transformations—that is, the destination file specified on the `TMS BEGIN` command.
- The `MODEL` subcommand specifies the file containing the model specifications. In the present example, there are two such files—the file where you saved the multinomial logistic regression model and the file where you saved the classification tree model. You'll need a separate `TMS MERGE` command for each of these model files.

If you haven't already done so, open *scoring_transformations.sps*. For the multinomial logistic regression model:

▶ Enter the location of the model file in place of `'file specification'` on the `MODEL` subcommand (include quotes in the file specification).

▶ Replace `'file specification'` on the `TRANSFORMATIONS` subcommand with the location of the file containing the exported data transformations (include quotes in the file specification).

- Replace `'file specification'` on the `DESTINATION` subcommand with the location of a file where the merged results will be written, and include the extension *xml* (include quotes in the file specification).

  Note: If you're licensed for SPSS Adaptor for Predictive Enterprise Services, you can store the merged file to a repository by using a file specification for a repository location. See the topic "File Specifications for Predictive Enterprise Repository Objects" (under "SPSS Adaptor for Predictive Enterprise Services" > "Command Syntax") in the Help system for more information.

- Place the cursor anywhere in the command syntax for the `TMS MERGE` command.

- From the menus in the SPSS Syntax Editor window, choose:

  Run
    Current

- Repeat this process for the classification tree model. Use the same `'file specification'` on the `TRANSFORMATIONS` subcommand (include quotes in the file specification) and choose a different location for the destination file.

Commands for Scoring Your Data

Now that you've built your models and merged the necessary transformations, you're ready to score your data.

Opening a Model File—The Model Handle Command

Before a model can be applied to a data file, the model specifications and any associated data transformations must be read into the current working session. This is accomplished with the `MODEL HANDLE` command.

Command syntax for the necessary `MODEL HANDLE` commands can be found in the section labeled *Read in the XML model files* in the file *scoring.sps*.

```
/**** Read in the XML model files ****.

MODEL HANDLE NAME=mregression FILE='file specification'.

MODEL HANDLE NAME=tree FILE='file specification'.
```

- Each model read into memory is required to have a unique name referred to as the model handle name.
- In this example, the name *mregression* is used for the multinomial logistic regression model and the name *tree* is used for the classification tree model. A separate `MODEL HANDLE` command is required for each XML model file.
- Before scoring the sample data, you'll need to replace the `'file specification'` strings in the `MODEL HANDLE` commands with the paths to the final model files created from `TMS MERGE` (include quotes in the file specification). Paths are interpreted relative to the computer on which SPSS Server is installed.

For further details on the `MODEL HANDLE` command, see the *SPSS Command Syntax Reference*, accessible as a PDF file from the Help menu.

Applying the Models—The ApplyModel and StrApplyModel Functions

Once a model file has been successfully read with the `MODEL HANDLE` command, you use the `ApplyModel` and/or the `StrApplyModel` functions to apply the model to your data.

The command syntax for the `ApplyModel` function can be found in the section labeled *Apply the model to the data* in the file *scoring.sps*.

```
/**** Apply the model to the data ****.

COMPUTE PredCatReg = ApplyModel(mregression,'predict').
COMPUTE PredCatTree = ApplyModel(tree,'predict').
```

- The `ApplyModel` and `StrApplyModel` functions are used with the `COMPUTE` command. `ApplyModel` returns results as numeric data. `StrApplyModel` returns the same results but as character data. Unless you need results returned as a string, you can simply use `ApplyModel`.
- These functions have two arguments: the first identifies the model using the model handle name defined on the `MODEL HANDLE` command (for example, *mregression*), and the second identifies the type of result to be returned, such as the model prediction or the probability associated with the prediction.
- The string value `'predict'` (include the quotes) indicates that `ApplyModel` should return the predicted result—that is, whether an individual will respond to the promotion. The new variables *PredCatReg* and *PredCatTree* store the predicted results for the multinomial logistic regression and tree models, respectively. A

value of 1 means that an individual is predicted to make a purchase; otherwise, the value is 0. The particular values 0 and 1 reflect the fact that the dependent variable, *Response* (used in both models), takes on these values.

For further details on the `ApplyModel` and `StrApplyModel` functions, including the types of results available for each model type, see "Scoring Expressions" in the "Transformation Expressions" section of the *SPSS Command Syntax Reference*.

Including Post-Scoring Transformations

Since scoring is treated as a set of data transformations, you can include transformations in your command syntax file that follow the ones for scoring—for example, transformations used to compare the results of competing models—and cause them to be processed in the same single data pass. For large data files, this can represent a substantial savings in computing time.

As a simple example, consider computing the agreement between the predictions of the two models used in this example. The necessary command syntax can be found in the section labeled *Compute comparison variable* in the file *scoring.sps*.

```
* Compute comparison variable.
COMPUTE ModelsAgree = PredCatReg=PredCatTree.
```

- This `COMPUTE` command creates a comparison variable called *ModelsAgree*. It has the value of 1 when the model predictions agree and 0 otherwise.

Getting Data and Saving Results

The command used to get the data to be scored depends on the form of the data. For example, if your data are in SPSS format, you will use the `GET` command, but if your data are stored in a database, you will use the `GET DATA` command.

After scoring, the active dataset contains the results of the predictions—in this case, the new variables *PredCatReg*, *PredCatTree*, and *ModelsAgree*. If your data were read in from a database, you will probably want to write the results back to the database. This is accomplished with the `SAVE TRANSLATE` command. For details on the `GET DATA` and `SAVE TRANSLATE` commands, see the *SPSS Command Syntax Reference*.

The command syntax for getting the data for the current example can be found in the section labeled *Get data to be scored* in the file *scoring.sps*.

```
/**** Get data to be scored ****.

GET FILE='file specification'.
```

- The data to be scored are assumed to be in an SPSS-format file (*customers_new.sav*). The `GET FILE` command is then used to read the data.
- Before scoring the sample data, you'll need to replace the `'file specification'` string in the `GET FILE` command with the path to *customers_new.sav* (include quotes in the file specification). Paths are interpreted relative to the computer on which SPSS Server is installed.

The command syntax for saving the results for the current example can be found in the section labeled *Save sample results* in the file *scoring.sps*.

```
/**** Save sample results ****.

SAVE OUTFILE='file specification'.
```

- The `SAVE` command can be used to save the results as an SPSS-format data file. In the case of writing results to a database table, the `SAVE TRANSLATE` command would be used.
- Before scoring the sample data, you will need to replace the `'file specification'` string in the `SAVE` command with a valid path to a new file (include quotes in the file specification). Paths are interpreted relative to the computer on which SPSS Server is installed. You'll probably want to include a file type of *.sav* for the file so that SPSS will recognize it. If the file doesn't exist, the `SAVE` command will create it for you. If the file already exists, it will be overwritten.

The saved file will contain the results of the scoring process and will be composed of the original file, *customers_new.sav*, with the addition of the three new variables, *PredCatReg*, *PredCatTree*, and *ModelsAgree*. You are now ready to learn how to submit a command file to the SPSS Batch Facility.

Running Your Scoring Job Using the SPSS Batch Facility

The SPSS Batch Facility is intended for automated production, providing the ability to run SPSS analyses without user intervention. It takes an SPSS syntax file (such as the command syntax file you have been studying), executes all of the commands in the file, and writes output to the file you specify. The output file contains a listing of the command syntax that was processed, as well as any output specific to the commands that were executed. In the case of scoring, this includes tables generated from the `MODEL HANDLE` commands showing the details of the variables read from the model files. This output is to be distinguished from the results of the `ApplyModel` commands used to score the data. Those results are saved to the appropriate data source with the `SAVE` or `SAVE TRANSLATE` command included in your syntax file.

The SPSS Batch Facility is invoked with the `spssb` command, run from a command line on the computer on which SPSS Server is installed.

```
/** Command line for submitting a file to the SPSS Batch Facility **

spssb -f \jobs\scoring.sps -type text -out \jobs\score.txt
```

- The sample command in this example will run the command syntax file *scoring.sps* and write text style output into *score.txt*.
- All paths in this command line are relative to the computer on which SPSS Server is installed.

Try scoring the data in *customers_new.sav* by submitting *scoring.sps* to the batch facility. Of course, you'll have to make sure you've included valid paths for all of the required files, as instructed above.

Part II: Programming with SPSS and Python

Chapter

11

Introduction

The SPSS-Python Integration Plug-In extends the SPSS command syntax language with the full capabilities of the Python programming language. With this feature, Python programs can access SPSS variable dictionary information, case data, procedure output, and error codes from SPSS commands. They can also submit command syntax to SPSS for processing, create new variables and new cases in the active dataset, and create output in the form of pivot tables and text blocks. A wide variety of tasks can be accomplished in a programmatic fashion with this technology.

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 an SPSS command 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 an SPSS command.

Dynamically Create and Submit SPSS Command Syntax

Python programs can dynamically construct SPSS command syntax and submit it to SPSS 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 SPSS variables from the active dataset that have a particular attribute and then use that list as the variable list for a given SPSS command.
- Perform SPSS data management operations on a dynamically selected set of files—for example, combine cases from all SPSS-format data files located in a specified directory.

Apply Custom Algorithms to Your Data

Access to the case data in the active dataset allows you to use the power of the Python language to perform custom calculations on your SPSS data. This opens up the possibility of using the vast set of scientific programming libraries available for the Python language. You can write the results back to the active dataset in the form of new variables or new cases 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 procedures, such as `DESCRIPTIVES` and `REGRESSION`.

Server-Side Scripting

Python programs (sometimes referred to as scripts in the context used here) interacting with SPSS execute on the computer that hosts the SPSS backend—which of course is where SPSS command syntax is always executed. In local mode, these programs execute on your local (desktop) computer, but in distributed mode, they execute on the server computer—a fact that allows you to perform operations on the server that were previously available only through client-side scripting on a Windows operating system. In that regard, the SPSS-Python Integration Plug-In is available for both Windows and UNIX-based operating systems.

Develop and Debug Code Using Third-Party IDEs That Drive SPSS

The SPSS-Python Integration Plug-In provides functionality to drive the SPSS backend from any Python IDE (Integrated Development Environment) or any separate Python process, such as the Python interpreter. You can then develop and debug your code with the Python IDE of your choice. 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 incorporate it into an SPSS command syntax job or put it into production as a job that drives SPSS from the standard Python interpreter.

Prerequisites

The SPSS-Python Integration Plug-In works with SPSS release 14.0.1 or later and requires only SPSS Base. The plug-in is available, along with installation instructions, on the installation CD for SPSS release 15.0 or later. It is also available for download from SPSS Developer Central at *http://www.spss.com/devcentral*.

The chapters that follow include hands-on examples of integrating the Python language with SPSS command syntax and assume a basic working knowledge of the language, although aspects of the language are discussed when deemed necessary. Unless stated otherwise, the examples presume SPSS 15.0.1.

For help getting started with the Python programming language, see the Python tutorial, available at *http://docs.python.org/tut/tut.html*.

Note: SPSS 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 does not make any statement about the quality of the Python program. SPSS fully disclaims all liability associated with your use of the Python program.

Additional Plug-Ins

The SPSS Programmability Extension, included with SPSS Base, provides a general framework for supporting external languages through integration plug-ins, such as the SPSS-Python Integration Plug-In. In particular, SPSS also provides a freeware integration plug-in for .NET, available from SPSS Developer Central. The .NET plug-in supports development in any .NET language and is intended for applications that interact with the SPSS backend but present their own user interface and provide their own means for displaying output.

Chapter

12

Getting Started with Python Programming in SPSS

Once you've installed the SPSS-Python Integration Plug-In, you have full access to all of the functionality of the Python programming language from within `BEGIN PROGRAM-END PROGRAM` program blocks in SPSS command syntax. The basic structure is:

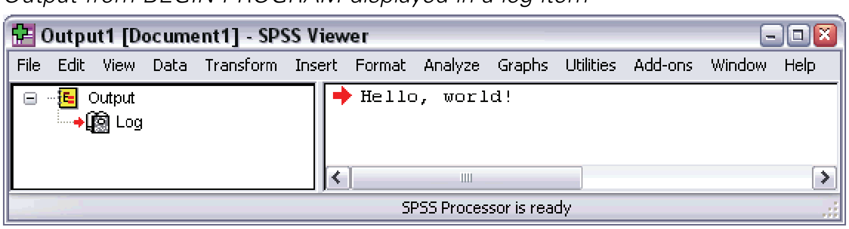
```
BEGIN PROGRAM.
Python statements
END PROGRAM.
```

Here's the classic "Hello, world!" example:

```
BEGIN PROGRAM.
print "Hello, world!"
END PROGRAM.
```

The example uses the Python `print` statement to write output to Python's standard output, which is directed to a log item in the SPSS Viewer if a Viewer is available.

Figure 12-1
Output from BEGIN PROGRAM displayed in a log item



Within a program block, the Python processor is in control, so all statements must be valid Python statements. Even though program blocks are part of SPSS command syntax, you can't include syntax commands as statements in a program block. For example,

```
BEGIN PROGRAM.
FREQUENCIES VARIABLES=var1, var2, var3.
END PROGRAM.
```

will generate an error because `FREQUENCIES` is not a Python command. Since the goal of a program block is often to generate some statements that SPSS can understand, there must be a way to specify SPSS commands within a program block. This is done using a function from the `spss` Python module, as discussed in the topic Submitting Commands to SPSS on p. 215.

The spss Python Module

The `spss` Python module is installed as part of the SPSS-Python Integration Plug-In and contains a number of SPSS-specific functions that enable the process of using Python programming features with SPSS command syntax.

The `spss` module provides functions to:

- Build and run SPSS command syntax
- Get information about data in the current SPSS session
- Get data, add new variables, and append cases to the active dataset
- Get output results
- Create custom pivot tables and text blocks
- Create macro variables
- Get error information
- Manage multiple versions of the SPSS-Python Integration Plug-in

The functions in the module are accessed by including the Python statement `import spss` as the first line in a program block, as in:

```
BEGIN PROGRAM.
import spss
spss.Submit("SHOW ALL.")
END PROGRAM.
```

You need to include the `import spss` statement only once in a given SPSS session. Repeating an `import` statement in subsequent `BEGIN PROGRAM` blocks essentially has no effect.

As you'll learn in the next topic, the `Submit` function shown above allows you to send commands to SPSS for processing. The prefix `spss` in `spss.Submit` specifies that this function can be found in the `spss` module. 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.
```

Many of the functions in the `spss` module are used in examples in the sections that follow. Appendix A contains details of all of the functions in the `spss` module. 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.

Submitting Commands to SPSS

The common task of submitting SPSS 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 an SPSS command and submits the command text to SPSS 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.

The 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='c:\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='c:/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 SPSS command.
- 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 always accepts a forward slash in file specifications. You can include backslashes and avoid escape sequences by using a raw string for the file specification. For more information, see Using Raw Strings in Python in Chapter 13 on p. 241.

SPSS command syntax generated within a program block and submitted to SPSS must follow interactive syntax rules. For most practical purposes, this means that SPSS command strings that you build in a programming block must contain a period (.) at the end of each SPSS 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 SPSS INSERT command in interactive mode (the default), as opposed to the INCLUDE command.

When you submit commands for SPSS 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 SPSS 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.

For more information on the `Submit` function, see Appendix A on p. 424.

Dynamically Creating SPSS Command Syntax

Using the functions in the `spss` module, you can dynamically compose SPSS 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='c:/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 SPSS. The set of SPSS 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, 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 to SPSS with a single call to `Submit` by passing to `Submit` a list of strings, each of which represents a single SPSS command. For more information, see Appendix A on p. 424. You can also submit a block of SPSS commands as a single string that spans multiple lines, resembling the way you might normally write command syntax. For more information, see Creating Blocks of Command Syntax within Program Blocks in Chapter 13 on p. 237.

Capturing and Accessing Output

Functionality provided with the SPSS-Python Integration Plug-In allows you to access SPSS 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.

```
*python_retrieve_output_value.sps.
BEGIN PROGRAM.
import spss,spssaux
spss.Submit("GET FILE='c:/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")
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 available from SPSS Developer Central at *http://www.spss.com/devcentral*. 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; 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 and to 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.

For more information, see Retrieving Output from SPSS Commands in Chapter 16 on p. 314.

Python Syntax Rules

Within a program block, only statements and functions recognized by the Python processor are allowed. Python syntax rules differ from SPSS command syntax rules in a number of ways:

Python is case-sensitive. This includes 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 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.

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 in 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 Using Raw Strings in Python in Chapter 13 on p. 241.

Python Quoting Conventions

- Strings in the Python programming language can be enclosed in matching single quotes (`'`) or double quotes (`"`), as in SPSS.
- 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 quote) within a string, use single quotes to enclose the string, as in

```
'Categories Labeled "UNSTANDARD" in the Report'
```

- The Python programming language treats doubled quotes of the same type as the outer quotes differently than SPSS. 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 SPSS Command Syntax

Within a program block, you can define an SPSS macro variable that can be used outside of the block in SPSS 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 Dynamically Creating SPSS Command Syntax on p. 217.

```
*python_set_varlist_macros.sps.
BEGIN PROGRAM.
import spss
spss.Submit("GET FILE='c:/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 an SPSS 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 SPSS 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 an SPSS 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 Retrieving Variable or Datafile Attributes in Chapter 14 on p. 271.

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 an SPSS 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 from a function in the `spss` module, 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.

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 an SPSS format file, so your attempt to obtain SPSS variable information may fail. Here's a code sample that handles this, assuming 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 SPSS 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 valid SPSS data file, 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.

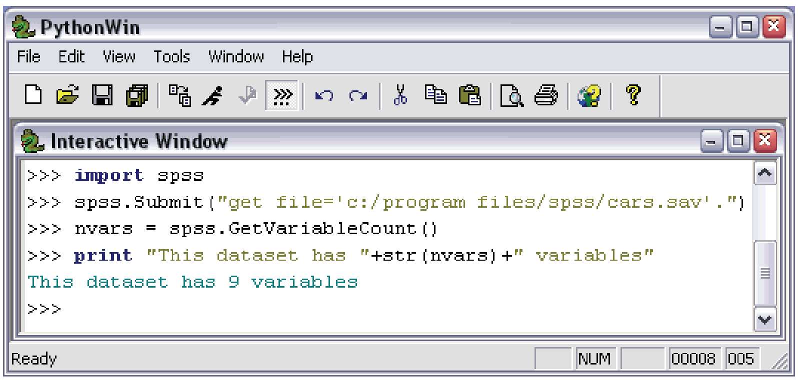
For more information on the `GetLastErrorLevel` and `GetLastErrorMessage` functions, see Appendix A on p. 424.

Using a Python IDE

The SPSS-Python Integration Plug-In provides functionality to drive the SPSS backend 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 copy it into an SPSS command syntax job.

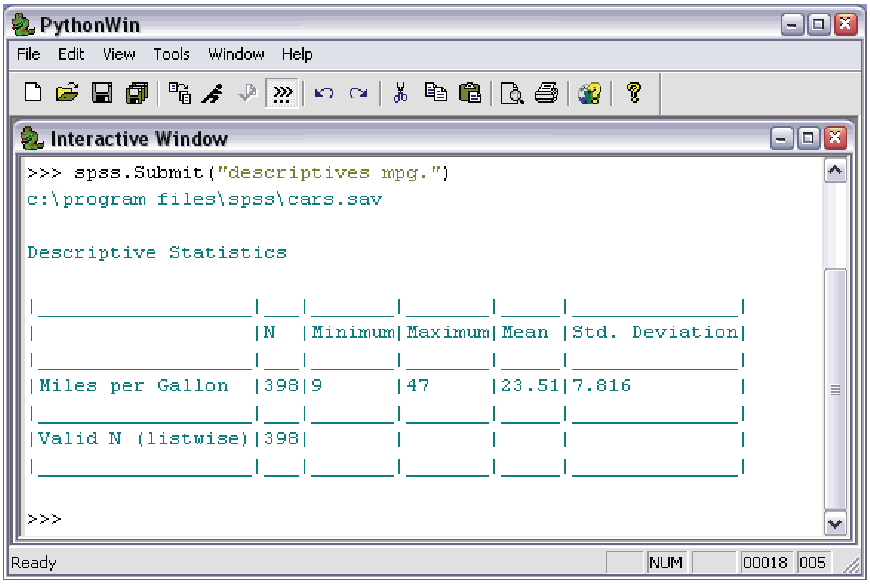
To drive the SPSS backend from a Python IDE, simply include an `import spss` statement in the IDE's code window. You can follow the `import` statement with calls to any of the functions in the `spss` module, just like with program blocks in SPSS command syntax jobs, but you don't include the `BEGIN PROGRAM-END 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 SPSS from a Python IDE



When you submit SPSS commands that would normally generate Viewer output, the output is directed to the IDE's output window, as shown below.

Figure 12-3
Output from an SPSS command displayed in a Python IDE



You can suppress output that would normally go to an SPSS 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 backend is being driven by an IDE. This might be the case if you have code that manipulates objects in the SPSS Viewer. Since no Viewer exists when you drive the SPSS backend from an IDE, you would need to know if your code were being run from an IDE so that you could raise an appropriate exception. 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 backend and 0 if SPSS is driving the SPSS backend.

Note: You can drive the SPSS backend with any separate Python process, such as the Python interpreter. Once you've installed the SPSS-Python Integration Plug-In, you initiate this mode with the `import spss` statement, just like driving the SPSS backend from a Python IDE.

Working with Multiple SPSS Versions

Beginning with SPSS 15.0, multiple versions of the SPSS-Python Integration Plug-in can be used on the same machine, each associated with an SPSS major version such as SPSS 14.0 or SPSS 15.0.

- When running Python code from within SPSS, SPSS will automatically use the appropriate plug-in—for example, the 15.0 Python plug-in will be used when running Python code from within SPSS 15.0.
- When driving the SPSS backend from a separate Python process, such as the Python interpreter or a Python IDE, the highest installed version of the plug-in is used by default (the plug-in starts up a compatible version of the SPSS backend). You can change which version of the plug-in is used with the spss.SetDefaultPlugInVersion function. You can view the default version with the spss.GetDefaultPlugInVersion function. And you can view all installed versions with the spss.ShowInstalledPlugInVersions function. For detailed information on these functions, see Appendix A on p. 424.

Note: Beginning with SPSS 15.0, a restructuring of the SPSS-Python Integration Plug-in installation directory and changes to some class structures may affect Python code written for an earlier SPSS version and used with an SPSS 15.0 or higher version. Specifically, the type of an object, as given by the Python `type` function, may return a different result. For example:

```
cur=spss.Cursor()
print type(cur)
```

will return `spss.spss150.cursors.ReadCursor` when run with SPSS 15.0, and `spss.cursors.Cursor` when run with SPSS 14.0.

Creating a Graphical User Interface

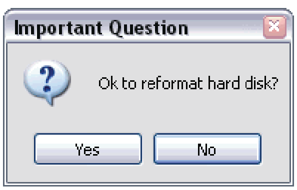
A variety of toolkits are available for creating graphical user interfaces with the Python programming language. You can use the `Tkinter` module, which is provided with Python, or choose from a number of third-party products. For illustration purposes, we'll work with wxPython, which is freely available 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 SPSS.

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 *.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 = "SPSS 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 SPSS 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.

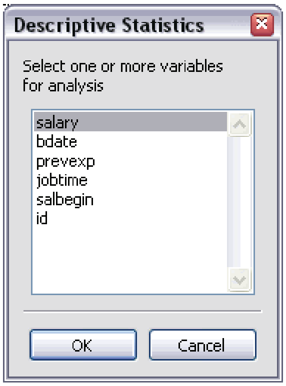
- The `ShowModal` method of the `FileDialog` instance is used to display the dialog box and returns the button clicked by the user—`wx.ID_OK` or `wx.ID_CANCEL`.
- The `GetPath` method of the `FileDialog` instance returns the full path of the selected file.
- If the user clicked OK and a non-empty file path was retrieved from the dialog box, then submit a `GET` command to SPSS 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("c:/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 available from SPSS Developer Central at *http://www.spss.com/devcentral*—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 SPSS 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 Getting Started with the VariableDict Class in Chapter 14 on p. 263.
- 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. SPSS 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 Dynamically Creating SPSS Command Syntax on p. 217.

Supplementary Python Modules for Use with SPSS

The `spss` module, included with the SPSS-Python Integration Plug-In, provides the base functionality for writing Python code that interacts with SPSS. A number of supplementary Python modules that build on, and in some cases extend, the functionality provided by the `spss` module are available for download from SPSS Developer Central at *http://www.spss.com/devcentral*. These modules include but are not limited to:

- Utilities to work with SPSS variable dictionary information and procedure output.
- Tools for working with the case data in the active dataset.
- Functionality to manipulate items in the SPSS Viewer.

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. And you'll get practice in using a number of functions from these modules in examples to follow. In many cases, the modules provide classes that wrap functionality from the `spss` module, allowing you to exploit object-oriented methods. 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 at SPSS Developer Central.

Getting Help

Help with using the features of the SPSS-Python Integration Plug-In is available from a number of resources:

- Appendix A provides descriptions and basic usage examples for each of the functions in the `spss` module. Once you've installed the plug-in, this material is also available as part of the PDF document *SPSS-Python Integration package*, located under *\help\programmability* in your SPSS application directory or accessed by choosing the Programmability option from the Help menu.
- An online description of a particular function, class, method, or module is available using the Python `help` function, once the associated module has been imported. 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.
- 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.
- Usage examples for the supplementary Python modules can be accessed from SPSS Developer Central at *http://www.spss.com/devcentral*. Examples for a particular module are bundled in command syntax (*.sps*) files and included with the topic for the module.
- Detailed command syntax reference information for `BEGIN PROGRAM-END PROGRAM` can be found in the SPSS Help system under the "Programmability" heading.
- For help in getting started with the Python programming language, see the Python tutorial, available at *http://docs.python.org/tut/tut.html*.

Chapter

13

Best Practices

This section provides advice for dealing with some common issues and introduces a number of features that will help you with writing code in program blocks.

Creating Blocks of Command Syntax within Program Blocks

Often, it is desirable to specify blocks of SPSS 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='c:/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 SPSS 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, which is standard for Windows and DOS. That said, it is a good practice to use forward slashes (/) in file paths, since you may at times forget to use raw mode, and SPSS accepts a forward slash for any backslash in a file specification. For more information, see Using Raw Strings in Python on p. 241.
- 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 SPSS command syntax in triple quotes within a `BEGIN PROGRAM-END PROGRAM` block allows you to easily convert an SPSS syntax job to a Python job. For more information, see Migrating Command Syntax Jobs to Python in Chapter 20 on p. 371.

Dynamically Specifying Command Syntax Using String Substitution

Most often, you embed SPSS command syntax within program blocks so that you can dynamically specify pieces of the syntax, such as SPSS 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 SPSS 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='c:/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 SPSS 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).
```

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 or regular expressions, that contain these sequences. For example, the 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 SPSS command syntax. SPSS also accepts a forward slash (/) for any backslash in a file specification. This provides an alternative to using raw strings for file specifications.

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 Using Regular Expressions to Select Variables in Chapter 14 on p. 273.

Displaying Command Syntax Generated by Program Blocks

For debugging purposes, it is convenient to see the completed syntax passed to SPSS 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='c:/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 Viewer, if the Viewer is available, and shows the completed `FREQUENCIES` command as well as the `GET` command:

```
300 M>  GET FILE='c:/examples/data/Employee data.sav'.
302 M>  FREQUENCIES /VARIABLES=minority.
```

Handling Wide Output in the Viewer

Within a program block, information sent to standard output, such as output from a Python `print` statement, is displayed in a log item in the Viewer. To help prevent long output lines from being truncated, it's a good idea to set your Viewer preferences to show wide lines. This is accomplished from the Viewer tab of the Options dialog box by selecting Wide (132 characters) in the Text Output Page Size group.

If you need to display a long list from Python, consider displaying each item in the list on a separate line. For example, given a Python list called *alist*, use:

```
print "\n".join(alist)
```

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 `"\n"` (the Python escape sequence for a line break), which causes each element of *alist* to be displayed on a separate line.

Creating User-Defined Functions in Python

`BEGIN PROGRAM-END PROGRAM` blocks encapsulate program code, so they might seem to be analogous to subroutines in other languages, but they differ fundamentally from subroutines since they can't be called. Undoubtedly, you will eventually want to create something like a subroutine and pass parameters to it. This is best done with a user-defined function in Python. 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, typically *C:\Python24\Lib\site-packages*.

  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. The block should contain an `import` statement for the module containing the function (unless you've imported the module in a previous block).

Example

Let's say you have a function that effectively changes the width of an SPSS string variable. The function has two parameters: the name of the variable and the new width. The function definition is:

```
def ChangeStringWidth(var,width):
    """Change the width of an SPSS string variable.
    var is the name of the variable
    width is the new width
    """
    width = int(width) or 1
    allnames = [spss.GetVariableName(v)
                for v in range(spss.GetVariableCount())]
    tempName="T" + str(random.random())
    varnames=" ".join(allnames[:allnames.index(var)] +
                 [tempName] + allnames[allnames.index(var)+1:])
    spss.Submit("""
    STRING %(tempName)s (A%(width)s).
    COMPUTE %(tempName)s = %(var)s.
    APPLY DICTIONARY FROM=*
           /SOURCE VARIABLES=%(var)s /TARGET VARIABLES=%(tempName)s.
    MATCH FILES FILE=* /KEEP %(varnames)s.
    RENAME VARIABLE (%(tempName)s=%(var)s).
    """ %locals())
```

- The `def` statement signals the beginning of a function named `ChangeStringWidth`. The colon at the end of the `def` statement is required.
- The function takes two parameters with the names *var* and *width*.
- The function combines Python code with a block of SPSS command syntax, which is specified dynamically and submitted to SPSS for processing. The values needed to specify the command syntax come from the function parameters and Python variables and are inserted into the command string using string substitution. For more information, see Dynamically Specifying Command Syntax Using String Substitution on p. 238.
- The techniques used in this function are very similar to those used in the example on reducing a string to minimum length, in the chapter *Working with Case Data in the Active Dataset*. For more information, see Chapter 15 on p. 286.

You include the function in a module named `samplelib` and now want to use the function to change the width of a string variable named *type* to a width of 30. The following is a command syntax file, including sample data, to do this:

```
*python_change_string_width.sps.
DATA LIST LIST /id(F8) type(A15).
BEGIN DATA
123 'Stilton'
267 'Jarlsberger'
103 'Danish Blue'
END DATA.

BEGIN PROGRAM.
import samplelib
samplelib.ChangeStringWidth('type',30)
END PROGRAM.
```

The `BEGIN PROGRAM` block starts with a statement to import the `samplelib` module, which contains the definition for the `ChangeStringWidth` function.

Note: To run this program block, you need to copy the module file *samplelib.py* from *\examples\python* on the accompanying CD to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*. Because the `samplelib` module uses functions in the `spss` module, it includes an `import spss` statement.

Creating a File Handle to the SPSS Install Directory

Depending on how you work with SPSS, it may be convenient to have easy access to files stored in the SPSS installation directory. This is best done by defining a file handle to the SPSS 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/Employee data.sav'.")
END PROGRAM.
```

- The program block imports and uses the supplementary module `spssaux`, available for download from SPSS Developer Central at *http://www.spss.com/devcentral*.

- 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 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 Using Raw Strings in Python on p. 241.

Note: To run this program block, you'll need to download the `spssaux` module from SPSS Developer Central and save it to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*.

Choosing the Best Programming Technology

With the introduction of the SPSS-Python Integration Plug-In, you now have three programming technologies (in addition to SPSS command syntax) available for use with SPSS—the SPSS macro language, the SPSS scripting facility, and Python. This section provides some advice on choosing the best technology for your task.

To start with, the ability to use Python to dynamically create and control SPSS command syntax renders SPSS 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. 375. However, macros are still important for passing information from a `BEGIN PROGRAM` block so that it is available to SPSS command syntax outside of the block. For more information, see the section "Passing Values from a Program Block to SPSS Command Syntax" in Mixing Command Syntax and Program Blocks on p. 223.

Like the SPSS scripting facility, Python provides a solution for programming tasks that cannot readily be done with SPSS command syntax. In that sense, it is not intended as a replacement for the SPSS command syntax language. Python is, however, almost always the preferred choice over the SPSS scripting facility. It 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. And Python statements always run synchronously with command syntax.

Consider using Python for tasks you may have previously done with the scripting facility, such as:

- Accessing the SPSS 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
- Manipulating output that appears in the Viewer, and integrating Viewer output into applications that support OLE automation, such as Microsoft PowerPoint
- Encapsulating a set of tasks in a program that accepts parameters and can be invoked from command syntax

Use the SPSS scripting facility for:

- Automatically performing a set of actions when a particular kind of object is created in the Viewer (referred to as autoscripting)
- Running custom dialog boxes or driving SPSS dialog boxes when operating in distributed mode

You'll still want to use the SPSS OLE automation interfaces if you're interested in controlling SPSS from an application that supports Visual Basic, such as Microsoft Office or Visual Basic itself

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. 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'll illustrate this by testing if 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 an SPSS format file, 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 SPSS 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 valid SPSS data file, 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 error level for the submitted SPSS 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:

```
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 Sax Basic

For users familiar with programming in Sax 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 Your Python Code

Two modes of operation are available for running Python code that interacts with SPSS: 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.

To get started with the Python IDE approach, see Using a Python IDE on p. 226. Because the SPSS-Python Integration Plug-In does not include a Python IDE, you'll have to obtain one yourself. Several are available, and a number of them 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 SPSS-Python Integration Plug-In, 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 SPSS 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 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='c:/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 SPSS 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 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 runtime in Python. For example, when Python encounters a syntax error, it terminates execution and indicates the earliest point in the line where the error was detected.

Chapter

14

Working with Variable Dictionary Information

The `spss` module provides a number of functions for retrieving variable 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)
- Missing values
- Custom variable attributes

A complete list of the available functions can be found in Appendix A on p. 424. Information about value labels and datafile attributes is most easily retrieved with object-oriented methods, as described in Using Object-Oriented Methods for Retrieving Dictionary Information on p. 262.

Functions in the `spss` module retrieve information for a specified variable using 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()`.

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='c:/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 SPSS. The command strings passed to the `Submit` function are:

  `'FREQUENCIES gender educ jobcat minority.'`

  `'DESCRIPTIVES id bdate salary salbegin jobtime prevexp.'`

For more information on the `GetVariableMeasurementLevel` function, see Appendix A on p. 424.

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, you need to copy the module file *samplelib.py* from *\examples\python* on the accompanying CD to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*. Because the `samplelib` module uses functions in the `spss` module, it includes an `import spss` statement.

The result is:

```
['timevar1', 'timevar2']
```

For more information on the `GetVariableFormat` function, see Appendix A on p. 424.

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='c:/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 SPSS using the `Submit` function. The name of the gender variable is inserted into the command block using string substitution. For more information, see Dynamically Specifying Command Syntax Using String Substitution in Chapter 13 on p. 238.
- 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.

For more information on the `GetVariableName` and `GetVariableLabel` functions, see Appendix A on p. 424.

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='c:/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.

For more information on the GetVariableType function, see Appendix A on p. 424.

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(f) v2(f) v3(f) v4(f) 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
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
   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, -1.7976931348623155e+308, 0.0, 999.0)
v5 (0, '    ', None, None)
```

- 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 `(1,0,1.5,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`. The result from `GetVarMissingValues` is `(2, -1.7976931348623155e+308, 0.0, 999.0)`. When a missing value range is specified with `LO` or `HI`, the result contains the value SPSS uses for `LO` or `HI`.

  Note: When specifying missing value ranges for new variables, you can use the `GetSPSSLowHigh` function, from the `spss` module, to get the correct values of `LO` and `HI`. For more information, see spss.GetSPSSLowHigh Function in Appendix A on p. 492.
- 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. *Note*: `GetVarMissingValues` is valid for long string variables (string variables with a maximum width greater than eight bytes) but will always return `(0,None,None,None)` since long string variables cannot have user-missing values.

The `Spssdata` class in the `spssdata` module (a supplementary module available for download from SPSS Developer Central at *http://www.spss.com/devcentral*) provides a number of convenient functions, built on `GetVarMissingValues`, for dealing with missing values when reading data. For more information, see Reading Case Data with the Spssdata Class in Chapter 15 on p. 292.

Using Object-Oriented Methods for Retrieving Dictionary Information

The `spssaux` module, a supplementary module available for download from SPSS Developer Central at *http://www.spss.com/devcentral*, provides object-oriented methods that simplify the task of retrieving variable dictionary information. You can retrieve the same variable dictionary information available with the functions from the `spss` module, but you can also retrieve value label information and datafile attributes.

In addition, you have the option of retrieving information by variable name rather than the variable's index value in the dataset. You can also set many variable properties, such as value labels, missing values, and measurement level.

Note: To run the examples in this section, you need to download the `spssaux` module from SPSS Developer Central and save it to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*.

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 `Variables` method, as in:

```
varDict.Variables()
```

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 `Variables` 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. 273. 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. You can loop through the SPSS 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 SPSS variable in *varDict* and provides access to that variable's dictionary information through method calls. For example, `var.VariableName` calls the `VariableName` method to retrieve the variable name for the SPSS 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 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('c:/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.

Identifying Variables without Value Labels

The `VariableDict` class allows you to retrieve value label information through the `ValueLabels` method. 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='c:/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` invokes the `ValueLabels` method for the variable represented by *var*. It returns a Python dictionary containing value label information for the specified variable. 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` method uses `OMS` to get value labels from the SPSS 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, it is better to work with an in-memory XML representation of the dictionary for the active dataset. This is created by the `CreateXPathDictionary` function from the `spss` module. Information can be retrieved with a variety of tools, including the

EvaluateXPath function from the spss module. In this example, we'll utilize the xml.sax module, a standard module distributed with Python that simplifies the task of working with XML. 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 variables *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 SPSS 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 from 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 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
    # Convert from Unicode to the current character set
    nolabelset = handler.varset.difference(handler.vallabelset)
    encoding = locale.getlocale()[1]
    if nolabelset:
        print "The following variables have no value labels:"
        print "\n".join([codecs.encode(v,encoding) 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*.
- The XML dictionary is represented in Unicode, but the results of `FindVarsWithoutValueLabels` will be displayed in the SPSS locale code page. To make the output compatible with the current SPSS character set, you use the `encode` method from the `codecs` module to convert from Unicode to the SPSS locale code page before invoking any string operations. You can set and display the SPSS locale using the `SET LOCALE` and `SHOW LOCALE` commands.

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='c:/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, you need to copy the module file *FindVarsUtility.py* from *\examples\python* on the accompanying CD to your Python “site-packages” directory, which is typically *C:\Python24\Lib\site-packages*. If you are interested in making use of the `xml.sax` module, the `FindVarsUtility` module may provide a helpful starting point.

Retrieving Variable or Datafile Attributes

The `VariableDict` class allows you to retrieve variable attributes or datafile attributes through the `Attributes` method.

Example

A number of variables in the sample dataset *Employee data.sav* have a variable attribute named `'DemographicVars'`. Create a list of these variables.

```
*python_var_attr.sps.
BEGIN PROGRAM.
import spss, spssaux
spss.Submit("GET FILE='c:/examples/data/Employee data.sav'.")
varDict = spssaux.VariableDict()
demvarList = [var.VariableName for var in varDict
              if var.Attributes.has_key('DemographicVars')]
print "Variables with the attribute DemographicVars:"
print "\n".join(demvarList)
END PROGRAM.
```

- `var.Attributes` invokes the `Attributes` method for the variable represented by *var*. It returns a Python dictionary containing any variable attributes for the specified variable. Each attribute, including each element of an attribute array, is assigned a separate key equal to the name of the attribute. The Python `has_key` method evaluates to *true* if there is a key in the associated dictionary with the specified name. Putting this all together, `var.Attributes.has_key('DemographicVars')` evaluates to *true* if the variable represented by *var* has an attribute named `'DemographicVars'`. Note that the `Attributes` method is case sensitive to the attribute name, although SPSS is not.
- The Python variable *demvarList* contains the list of variables that have the specified attribute. *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*.

Example

The sample dataset *Employee data.sav* has a number of datafile attributes. Retrieve the value of the attribute named `'LastRevised'`.

```
*python_file_attr.sps.
BEGIN PROGRAM.
import spss, spssaux
spss.Submit("GET FILE='c:/examples/data/Employee data.sav'.")
varDict = spssaux.VariableDict(namelist=[0])
LastRevised = varDict.Attributes().get('LastRevised')
print "Dataset last revised on:", LastRevised
END PROGRAM.
```

`varDict.Attributes()` invokes the `Attributes` method without an argument. In this mode, the method returns a Python dictionary containing any datafile attributes for the active dataset. Each attribute, including each element of an attribute array, is assigned a separate key equal to the name of the attribute. The Python `get` method operates on a dictionary and returns the value associated with the specified key.

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='c:\examples\data\Employee data.sav'.
DATAFILE ATTRIBUTE ATTRIBUTE=pythonArg('cheese').
BEGIN PROGRAM.
import spss, spssaux
varDict = spssaux.VariableDict(namelist=[0])
product = varDict.Attributes().get('pythonArg')
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 `Attributes` method of the `VariableDict` class (as in the preceding example) to retrieve the value of *pythonArg*. The value is stored to the Python variable *product*.

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='c:/examples/data/demo.sav'.")
varDict = spssaux.VariableDict(pattern=r'own')
print "\n".join(varDict.Variables())
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'`.

- 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 Using Raw Strings in Python in Chapter 13 on p. 241.
- The `Variables` 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.Variables(pattern=r'.*\d$'))
END PROGRAM.
```

- The argument *pattern* can be used with the `Variables` 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. 386.

Chapter

15

Working with Case Data in the Active Dataset

The SPSS-Python Integration Plug-In 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 once you've imported the `spss` module.

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. For more information, see spss.Cursor Methods in Appendix A on p. 467.

Note: For users of a 14.0.x version of the plug-in who are upgrading to the 15.0 version, 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 the 15.0 version.

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])
```

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.
- 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*.

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   '))
```

For more information on read mode, see Appendix A on p. 460. If the data to retrieve include SPSS datetime values, you should use the `spssdata` module, which properly converts SPSS datetime values to Python datetime objects. The `spssdata` module provides a number of other useful features, such as the ability to specify a list of variable names, rather than indexes, when retrieving a subset of variables, and addressing elements of tuples (containing case data) by the name of the associated variable. For more information, see Using the spssdata Module on p. 291.

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 (For more information, see spss.SplitChange Function in Appendix A on p. 503.) 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—*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`. For more information, see IsEndSplit Method in Appendix A on p. 475.

Creating New SPSS Variables with the Cursor Class

To add new SPSS 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. 288.

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. For more information, see spss.Cursor Methods in Appendix A on p. 467.
- 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.
- 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, use the mode `'n'` of the `Spssdata` class. For more information, see Creating a New Dataset with the Spssdata Class on p. 307. For more information on write mode, see Appendix A on p. 462.

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 SPSS 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, use the mode `'n'` of the `Spssdata` class. For more information, see Creating a New Dataset with the Spssdata Class on p. 307. For more information on append mode, see Appendix A on p. 465.

Example: Reducing a String to Minimum Length

For various reasons, we often find ourselves with strings of greater length than necessary. The following Python user-defined function reduces the defined length of a string variable to the length needed to accommodate the variable's values. The approach is to find the maximum number of bytes (excluding trailing spaces) in the case data for the variable, create a new string variable with a defined length equal to that value, drop the original variable, and rename the new string to the original name.

```
def ReformatString(varList):
    """Reformat a set of SPSS string variables in the active dataset
    so that the defined length of each variable is the minimum
    required to accommodate the variable's values.
    varList is a list of variable names which can include numeric
    and string variables. Reformatting will be done for each of
    the string variables found in varList.
    """
    allnames = [spss.GetVariableName(v)
                for v in range(spss.GetVariableCount())]
    indexList = [allnames.index(var) for var in varList]
    # Build a list of string variables found in varList
    strvarList=[var for i,var in enumerate(varList)
                if spss.GetVariableType(indexList[i])>0]

    if strvarList:
        # Find the maximum length for each of the string variables
        strLens=[1 for var in strvarList]
        curObj=spss.Cursor([allnames.index(var) for var in strvarList])
        for i in range(spss.GetCaseCount()):
            row=curObj.fetchone()
            strLens=[max(len(row[j].rstrip()),strL)
                     for j,strL in enumerate(strLens)]
        curObj.close()

        # Reformat each of the string variables
        for i,var in enumerate(strvarList):
            maxlen=strLens[i]
            tempName="T" + str(random.random())
            # Build a list of variable names to preserve file order
            varnames=" ".join(allnames[:allnames.index(var)] +
                        [tempName] + allnames[allnames.index(var)+1:])
            spss.Submit("""
            STRING %(tempName)s (A%(maxlen)s).
            COMPUTE %(tempName)s = %(var)s.
            APPLY DICTIONARY FROM=*
                /SOURCE VARIABLES=%(var)s /TARGET VARIABLES=%(tempName)s.
            MATCH FILES FILE=* /KEEP %(varnames)s.
            RENAME VARIABLE (%(tempName)s=%(var)s).
            """ %locals())
```

- `ReformatString` is a Python user-defined function that requires a single argument, *varList*.
- The Python variable *indexList* contains the list of variable indexes associated with the variable names passed in as *varList*. It relies on the variable *allnames*, which contains a list of the names of all variables in the active dataset in file order.
- The Python variable *strvarList* contains the names of the string variables found in *varList*. It uses the `GetVariableType` method from the `spss` module to determine if a given variable is a string.

  Note: If you're 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 Python variable *curObj* holds an instance of the `Cursor` class that will read the case data for just the string variables found in *varList*. The argument to the `Cursor` class is a list of index values of the string variables.
- The first `for` loop reads the case data to determine the maximum length needed to accommodate the values for each of the string variables. The Python string method `rstrip` is used to strip trailing blanks. The minimum allowable length is 1.
- The code `random.random()` generates a random number between 0 and 1. The string representation of this number can be used to build the name of a temporary variable that is virtually assured not to conflict with the name of any existing variable. The Python module that contains the `ReformatString` function includes a statement to import the `random` module, a standard module provided with Python.
- The `Submit` function stores the original variable to a temporary variable, applies the dictionary format from the original variable to the temporary one, uses `MATCH FILES` to maintain the file order of the variables while dropping the original variable, and renames the temporary variable to the original variable's name.
- String substitution is used to specify the dynamic pieces of the command syntax submitted to SPSS. For more information, see Dynamically Specifying Command Syntax Using String Substitution in Chapter 13 on p. 238.

Example

As an example, create a sample dataset with string variables and call `ReformatString`.

```
*python_reformat_string.sps.
DATA LIST FREE /string1 (A10) string2 (A10) string3 (A10).
BEGIN DATA
a ab abc
a abcde ab
abcdef abcdefgh abcdefghi
END DATA.

BEGIN PROGRAM.
import samplelib
samplelib.ReformatString(['string1','string2','string3'])
END PROGRAM.
```

- The `BEGIN PROGRAM` block starts with a statement to import the `samplelib` module, which contains the definition for the `ReformatString` function.
- The result is that *string1* is resized to a length of 6; *string2*, to a length of 8; and *string3*, to a length of 9.

Note: To run this program block, you need to copy the module file *samplelib.py* from *\examples\python* on the accompanying CD to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*. Because the `samplelib` module uses functions in the `spss` module, it includes an `import spss` statement.

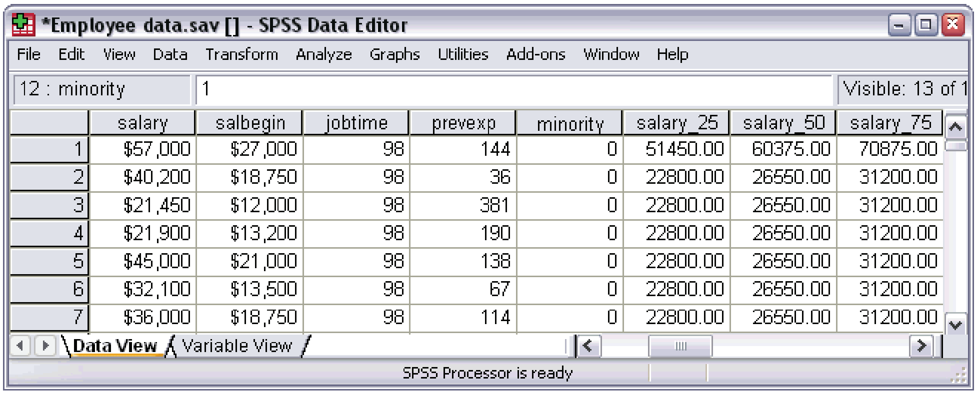
Example: Adding Group Percentile Values to a Dataset

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 SPSS `Rank` procedure.

```
*python_add_group_percentiles.sps.
BEGIN PROGRAM.
import spss, math
spss.Submit("GET FILE='c:/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]
   if not salaries.has_key(jobcat):
      salaries[jobcat]={}
      counts[jobcat]=0
   salaries[jobcat][salary]=salaries[jobcat].get(salary,0) + 1
   counts[jobcat]+=1

# Calculate the cutpoint salary value for each percentile for each
# employment category
percentiles={}
for jobcat in salaries:
   cutpoints = [int(math.ceil(counts[jobcat]*1/4.)),
                int(math.ceil(counts[jobcat]*1/2.)),
                int(math.ceil(counts[jobcat]*3/4.))]
   tempcount=0; pctindex=0
   percentiles[jobcat]=[]
   salarylist=salaries[jobcat].keys()
   salarylist.sort()
   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



	salary	salbegin	jobtime	prevexp	minority	salary_25	salary_50	salary_75
1	$57,000	$27,000	98	144	0	51450.00	60375.00	70875.00
2	$40,200	$18,750	98	36	0	22800.00	26550.00	31200.00
3	$21,450	$12,000	98	381	0	22800.00	26550.00	31200.00
4	$21,900	$13,200	98	190	0	22800.00	26550.00	31200.00
5	$45,000	$21,000	98	138	0	22800.00	26550.00	31200.00
6	$32,100	$13,500	98	67	0	22800.00	26550.00	31200.00
7	$36,000	$18,750	98	114	0	22800.00	26550.00	31200.00

Using the spssdata Module

The `spssdata` module, a supplementary module that you can download from SPSS Developer Central at *http://www.spss.com/devcentral*, 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.
- You can specify that SPSS datetime values be converted to Python datetime objects. And you can easily convert from a date (represented as a four-digit year, month, and day) to the internal representation used by SPSS.
- You can create an entirely new dataset, specifying variables and appending cases.

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. 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 the 15.0 version, read mode for the 15.0 version of the `Spssdata` class 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 the 15.0 version.
- To run the examples in this section, you need to download the `spssdata` module, the accompanying `namedtuple` module, and the `spssaux` module from SPSS Developer Central and save them to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*.
- 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 or tuple. 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.close()
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 in the form of a customized tuple called a **named tuple**. Like a tuple returned by the Cursor class, a named tuple contains the data values for a single case. In addition, a named tuple contains an attribute for each retrieved variable, with a name equal to the variable name and with a value equal to the variable's value for the current case. 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. Since the variable *row* is a tuple, you can also access elements by index; for example, row[0] gives the value of *sku* and row[1] gives the value of *qty*.

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.close()
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.close()
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 (For more information, see spss.SplitChange Function in Appendix A on p. 503.). 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=[]; splitcount=0
for row in data:
   if data.IsStartSplit():
      counts.append(splitcount)
      splitcount=1
   else:
      splitcount+=1
data.close()
counts.append(splitcount)
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—*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 SPSS Datetime Values from SPSS

Dates and times in SPSS are represented internally as seconds. This means that data retrieved for an SPSS datetime variable will simply be an integer representing some number of seconds. SPSS knows how to correctly interpret this number when performing datetime calculations and displaying datetime values, but without special instructions, Python won't. To illustrate this point, consider the following sample data and code (using the `Cursor` class) 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 stores dates internally as the number of seconds since October 14, 1582. Fortunately, the `Spssdata` class will do the necessary transformations for you and convert an SPSS 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 an SPSS 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.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.close()
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 SPSS Variables with the Spssdata Class

To add new SPSS variables with 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 Reading Case Data with the Spssdata Class on p. 292.

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. 308.

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.close()
END PROGRAM.
```

- The `append` method from the `Spssdata` class is used to add the specifications for a new SPSS 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, etc. 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 `close` method is used to close 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, use the mode `'n'` of the `Spssdata` class. For more information, see Creating a New Dataset with the Spssdata Class on p. 307.

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.close()
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 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.close()
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. 261.

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.close()
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.close()
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 SPSS Date Format Variables

In SPSS, dates are stored internally as the number of seconds from midnight on October 14, 1582. When setting values for new SPSS 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.close()
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.

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 SPSS 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, use the mode `'n'` of the `Spssdata` class. For more information, see Creating a New Dataset with the Spssdata Class on p. 307.

Creating a New Dataset with the Spssdata Class

To create a new dataset with the `Spssdata` class, you create an instance of the class in new mode, as in:

```
data = spss.Spssdata(accessType='n')
```

Creating a new dataset requires functions used in both creating new variables (write mode) and appending new cases (append mode).

Example

```
*python_Spssdata_create_dataset.sps.
BEGIN PROGRAM.
import spssdata
cur = spssdata.Spssdata(accessType='n')
cur.append(spssdata.vdef('case',
                          vlabel='Sample numeric variable',
                          vfmt=["F",2,0]))
cur.append(spssdata.vdef('value',
                          vlabel='Sample string variable',
                          vtype=1))
cur.commitdict()
cur.appendvalue('case',1)
cur.appendvalue('value','a')
cur.CommitCase()
cur.appendvalue('case',2)
cur.appendvalue('value','b')
cur.CommitCase()
cur.CClose()
END PROGRAM.
```

- You first use the `append` method from the `Spssdata` class to add the specifications for the SPSS variables in the new dataset. In the current example, the new dataset will contain the single numeric variable *case* and the single string variable *value*. For more information about specifying properties of new variables, see Creating New SPSS Variables with the Spssdata Class on p. 300.
- Once variable specifications have been added with the `append` method, the `commitdict` method is called to create the variables.

- Case values for each new case 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. 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. For more information, see Appending New Cases with the Spssdata Class on p. 306.
- 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: 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. 288), 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='c:/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:
   if not salaries.has_key(row.jobcat):
      salaries[row.jobcat]={}
      counts[row.jobcat]=0
   salaries[row.jobcat][row.salary]= \
      salaries[row.jobcat].get(row.salary,0) + 1
   counts[row.jobcat]+=1

# Calculate the cutpoint salary value for each percentile for each
# employment category
percentiles={}
for jobcat in salaries:
   cutpoints = [int(math.ceil(counts[jobcat]*1/4.)),
                int(math.ceil(counts[jobcat]*1/2.)),
                int(math.ceil(counts[jobcat]*3/4.))]
   tempcount=0; pctindex=0
   percentiles[jobcat]=[]
   salarylist=salaries[jobcat].keys()
   salarylist.sort()
   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()
cur.close()
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 Example: Adding Group Percentile Values to a Dataset on p. 288.
- 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 Creating New SPSS Variables with the Spssdata Class on p. 300.
- 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.

```
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 Creating New SPSS Variables with the Spssdata Class on p. 300.
- Case values are set using the `appendvalue` method from the `Spssdata` class. For more information, see Appending New Cases with the Spssdata Class on p. 306.
- 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

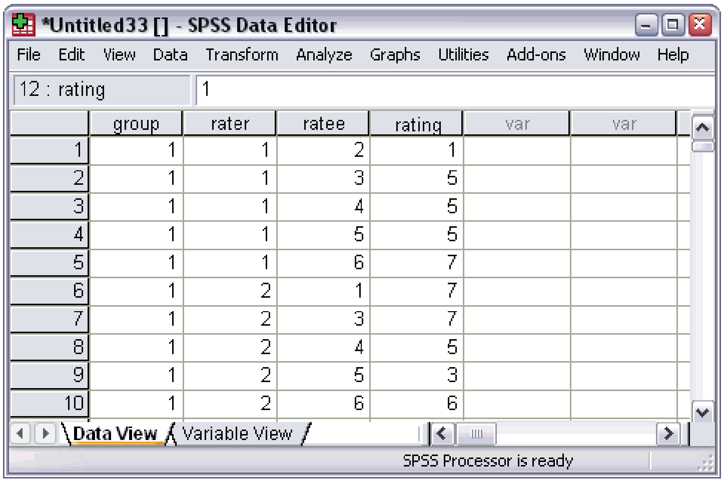
As an example, generate a sample dataset for 10 groups with 6 members each and a maximum score of 7.

```
*python_generate_data.sps.
BEGIN PROGRAM.
import samplelib_supp
samplelib_supp.GenerateData(10,6,7)
END PROGRAM.
```

- The `BEGIN PROGRAM` block starts with a statement to import the `samplelib_supp` module, which contains the definition for the `GenerateData` function.

Note: To run this program block, you need to copy the module file *samplelib_supp.py* from *\examples\python* on the accompanying CD to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*. The `samplelib_supp` module uses functions in the `spssaux`, `viewer`, `spssdata`, and `namedtuple` modules, so you will also need copies of these modules in your "site-packages" directory. You can download them from SPSS Developer Central at *http://www.spss.com/devcentral*.

Figure 15-2
Resulting dataset



Chapter

16

Retrieving Output from SPSS Commands

The `spss` module provides the means to retrieve the output produced by SPSS 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 SPSS 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 provided by SPSS, 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 Using the spssaux Module on p. 318.

The example in this section utilizes an explicit XPath expression. Constructing the correct XPath expression (SPSS 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 "SPSS 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='c:\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
spss.DeleteXPathHandle(handle)
END PROGRAM.
```

- The `OMS` command is used to direct output from an SPSS 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 Using the spssaux Module on p. 318.

- 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 SPSS command `SHOW OLANG`.

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 *c:\temp\descriptives_table.xml*.

```
*python_write_workspace_item.sps.
GET FILE='c:\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','c:/temp/descriptives_table.xml')
spss.DeleteXPathHandle('desc_table')
END PROGRAM.
```

The section of *c:\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 available for download from SPSS Developer Central at *http://www.spss.com/devcentral*, 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.

Note: To run the examples in this section, download the `spssaux` module from SPSS Developer Central and save it to your Python "site-packages" directory, which is typically *C:\Python24\Lib\site-packages*.

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 SPSS.
- `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='c:/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 SPSS 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 16-1
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 to SPSS. 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 error level for the submitted SPSS commands (0 if there were no SPSS 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 SPSS 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 Writing XML Workspace Contents to a File on p. 317.

  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 SPSS command `SHOW OLANG`.

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='c:/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 16-2
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.

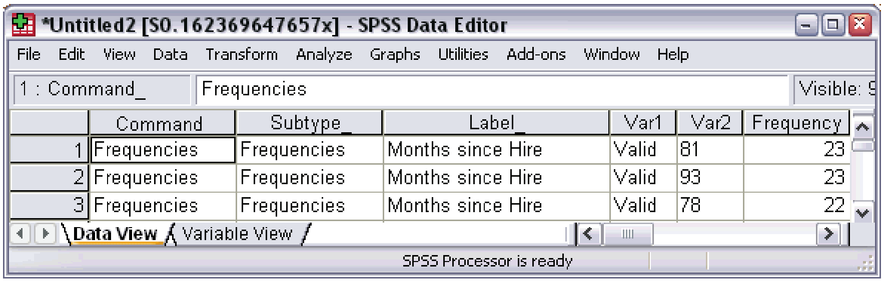
Example: Retrieving Output without the XML Workspace

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='c:/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 16-3
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 to SPSS. 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 error level for the submitted SPSS commands (0 if there were no SPSS 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 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`.

Note: In addition to the `spssaux` module, this example uses the `spssdata` module, available for download from SPSS Developer Central at *http://www.spss.com/devcentral*. Once you have downloaded the module, save it to your Python "site-packages" directory, which is typically *C:\Python24\Lib\site-packages*. For more information on working with the `Spssdata` class, see Using the spssdata Module on p. 291.

Chapter

17

Creating Procedures

The SPSS-Python Integration Plug-In enables you to create user-defined Python programs that have almost the same capabilities as SPSS procedures, such as `FREQUENCIES` or `REGRESSION`. Since they behave like built-in SPSS 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 SPSS procedures and/or SPSS 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 SPSS commands from Python, you can write a procedure that uses Python logic to drive the SPSS 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—that you want to apply to SPSS datasets. You can code the algorithm in Python and include it in a procedure that reads the data from SPSS and applies the algorithm. You might even use the powerful data transformation abilities of SPSS 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 SPSS-Python Integration Plug-In 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.

```
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.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 available from SPSS Developer Central). 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 Reading Case Data with the Spssdata Class in Chapter 15 on p. 292.
- 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 17-1. It is also the command name associated with this procedure when routing output from this procedure with OMS (Output Management System), as well as the name associated with this procedure for use with autoscripts. For more information, see spss.StartProcedure Function in Appendix A on p. 506.
- In order that names associated with output do not conflict with names of existing SPSS commands (when working with OMS or autoscripts), SPSS recommends that they have the form *yourorganization.com.procedurename*, as done here. When working with autoscripts, note that periods (.) contained in an output name are replaced with zeros (0), dashes are replaced with underscores, and spaces are removed in the associated autoscript's name. Avoid any other punctuation characters that might create illegal names in a programming language. For instance, output associated with the name `Smith & Jones, Ltd` generates an associated autoscript named `Smith&Jones,Ltd`, which would be illegal as part of a subroutine name in Sax Basic.
- 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 Creating Pivot Table Output on p. 335.
- 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. For more information, see spss.TextBlock Class in Appendix A on p. 512.

  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 SPSS 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_supp.py* located in *\examples\python* on the accompanying CD. 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, 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, which is typically *C:\Python24\Lib\site-packages*.

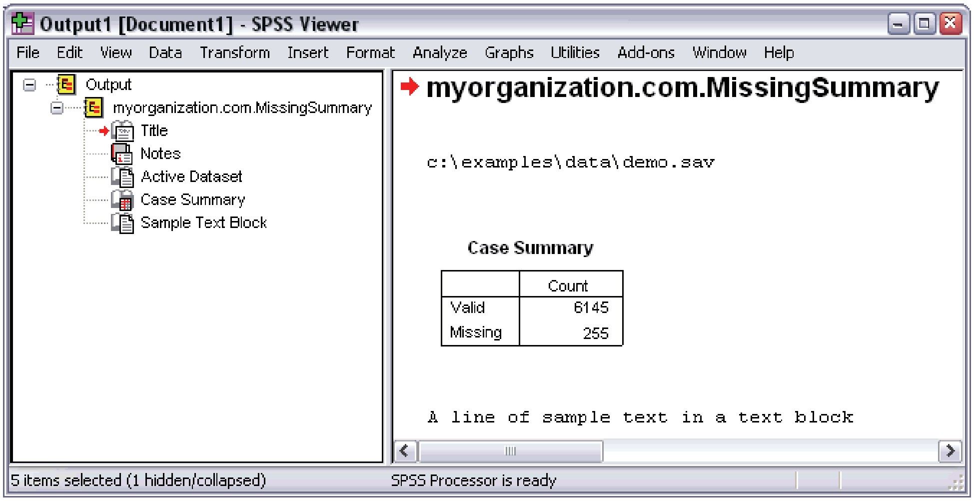
To run a procedure, you import the module containing it and call it with the necessary arguments. As an example, we'll run the `MissingSummary` procedure on the *demo.sav* dataset.

```
*python_missing_summary.sps.
BEGIN PROGRAM.
import samplelib_supp
samplelib_supp.MissingSummary("c:/examples/data/demo.sav")
END PROGRAM.
```

Note: To run this program block, you need to copy the module file *samplelib_supp.py* from *\examples\python* on the accompanying CD to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*. The `samplelib_supp` module uses functions in the `spssaux`, `viewer`, `spssdata`, and `namedtuple` modules, so you will also need copies of these modules in your "site-packages" directory. You can download them from SPSS Developer Central at *http://www.spss.com/devcentral*.

Result

Figure 17-1
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='c:/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 spss.BaseProcedure Class in Appendix A on p. 456.

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 SPSS 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 available from SPSS Developer Central. For more information, see Using the spssdata Module in Chapter 15 on p. 291.
- 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 Creating Pivot Table Output on p. 335.

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_supp
spss.Submit("GET FILE='c:/examples/data/Employee data.sav'.")
samplelib_supp.GroupMeans("educ","salary")
END PROGRAM.
```

The BEGIN PROGRAM block starts with a statement to import the samplelib_supp module, which contains the definition for the GroupMeans function.

Note: To run this program block, you need to copy the module file *samplelib_supp.py* from *\examples\python* on the accompanying CD to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*. The samplelib_supp module uses functions in the spssaux, viewer, spssdata, and namedtuple modules, so you will also need copies of these modules in your "site-packages" directory. You can download them from SPSS Developer Central at *http://www.spss.com/devcentral*.

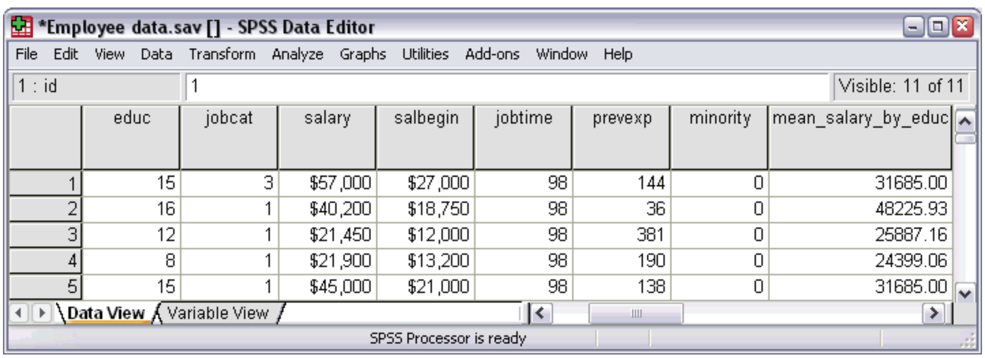
Results

Figure 17-2
Pivot table output from the GroupMeans procedure

Mean salary by educ

educ	mean salary
8	24399
12	25887
14	31625
15	31685
16	48226
17	59527
18	65128
19	72520
20	64313
21	65000

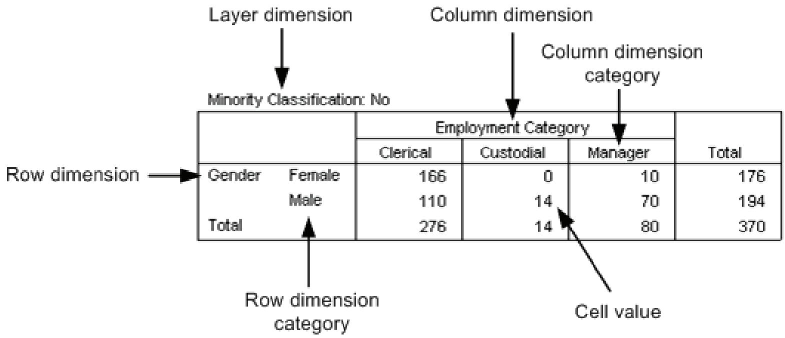
Figure 17-3
New variable mean_salary_by_educ created by the GroupMeans procedure



Creating Pivot Table Output

Procedures can produce output in the form of pivot tables, which can be displayed in the SPSS viewer or written to an external file using the SPSS Output Management System. The following figure shows the basic structural components of a pivot table.

Figure 17-4
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.

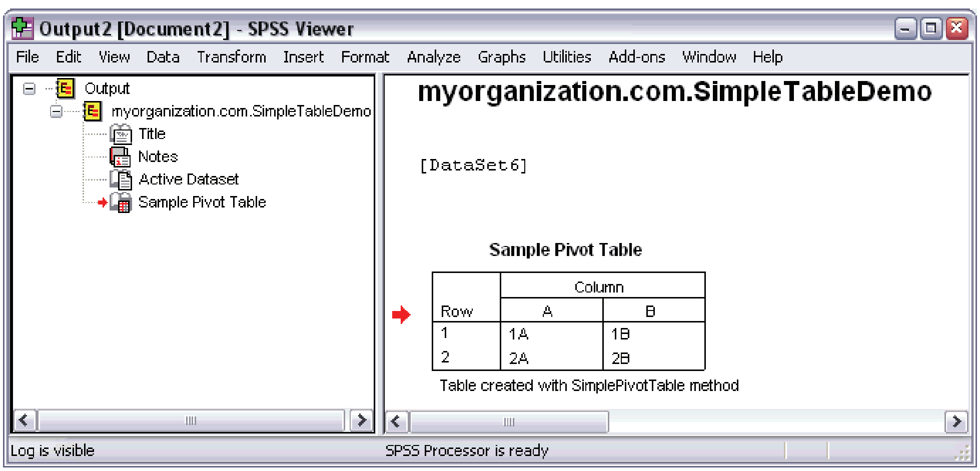
Pivot tables are created with the `BasePivotTable` class. For the common case of creating a pivot table with a single row dimension and a single column dimension, the `BasePivotTable` class provides the `SimplePivotTable` method. As a simple example, consider a pivot table with static values.

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 17-5
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 Getting Started with Procedures on p. 327.
- 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 17-5.
- 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 17-5 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. For more information, see spss.BasePivotTable Class in Appendix A on p. 425.
- The optional argument *caption* used in this example specifies a caption for the table, as shown in Figure 17-5 .

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 SPSS variable names or SPSS variable values. Treating labels as SPSS 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 SPSS variable values means that display settings for variable values in pivot tables (values, labels, or both) are honored. For more information, see Treating Categories or Cells as Variable Names or Values on p. 340.

  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*. See SimplePivotTable Method on p. 447 for examples of the resulting table structure when column labels are omitted.
- **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 Treating Categories or Cells as Variable Names or Values on p. 340.

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. For more information, see General Approach to Creating Pivot Tables in Appendix A on p. 429. 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 spss.SplitChange Function in Appendix A on p. 503.

Treating Categories or Cells as Variable Names or Values

The `BasePivotTable` class supports treating categories (row or column) and cell values as SPSS variable names or SPSS variable values. Treating categories as SPSS 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 SPSS 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 SPSS variable values and the single column category is specified as an SPSS variable name.

```
*python_ptable_VarValue_VarName.sps.
BEGIN PROGRAM.
import spss, spssdata
spss.Submit("GET FILE='c:/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 17-6
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 17-7
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 available from SPSS Developer Central) to read the data (only the values for *jobcat* and *gender* are read) and skip over cases with missing values. For more information, see Using the spssdata Module in Chapter 15 on p. 291.
- 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 SPSS 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'`. For more information, see VarValue Class in Appendix A on p. 454.
- 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 SPSS 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 an SPSS 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. For more information, see VarName Class in Appendix A on p. 453.
- Figure 17-6 and Figure 17-7 show the results for two different settings of output labels for pivot tables (set from Edit > Options > Output Labels). Figure 17-6 shows the case of variable names displayed as the associated variable label and variable values as the associated value label. Figure 17-7 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

By default, numeric values provided for pivot table cells and category labels are rounded to the nearest integer in the table output. You can change the default format 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='c:/examples/data/Employee data.sav'.")
spss.StartProcedure("myorganization.com.Demo")
table = spss.BasePivotTable("Table Title","OMS table subtype")
table.SetDefaultFormatSpec(spss.FormatSpec.GeneralStat)
table.SimplePivotTable(collabels=["Numeric Value"],
                       cells = [1.2345,2.34])
spss.EndProcedure()
END PROGRAM.
```

Result

Figure 17-8
Numeric cells formatted with the GeneralStat format

	Numeric Value
row0	1.235
row1	2.340

- 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.GeneralStat`. 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 section on the `Number` class on p. 450. For more information, see SetDefaultFormatSpec in Appendix A on p. 446.
- 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='c:/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.GeneralStat),
               2.34])
spss.EndProcedure()
END PROGRAM.
```

Result

Figure 17-9
Default format overridden for selected cell

	Numeric value
Default overridden	1.235
Default used	2

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`. For more information, see Number Class in Appendix A on p. 450.

Chapter

18

Data Transformations

The Python module `trans`, a supplementary module available for download from SPSS Developer Central at *http://www.spss.com/devcentral*, provides the framework for using Python functions as casewise transformation functions to be applied to an SPSS dataset. This enables an essentially limitless extension to the set of functions that can be used to transform SPSS 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 SPSS variables (modification of existing SPSS 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, you need to download the following modules from SPSS Developer Central and save them to your Python "site-packages" directory (typically *C:\Python24\Lib\site-packages*): `trans`, `extendedTransforms`, `spssaux` (at least version 2.0.0), and `spssdata` (at least version 2.0.0).

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 SPSS variables in the active dataset using existing SPSS variables or Python expressions as inputs. For example, consider applying the hyperbolic sine function (available with Python but not with SPSS) 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 SPSS variables that will contain the results of the function.

  The third argument specifies the format(s) for the resulting SPSS variable(s). Formats should be given as strings that specify SPSS 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 SPSS 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.

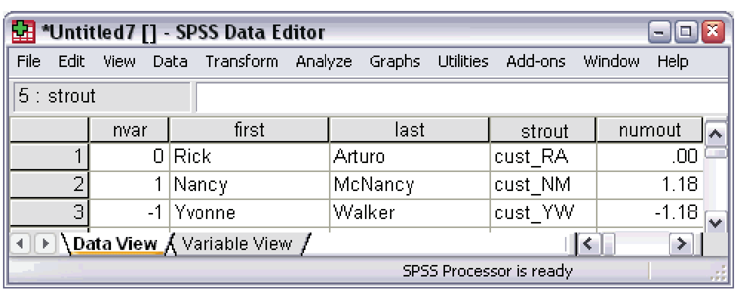
- In the present example, the Python function `strfunc` requires three arguments, so the call to `append` for `strfunc` contains a list with three elements, one for each argument. The first argument specifies the string constant `'cust'`. Python string expressions, to be passed as arguments, are specified as `trans.const(expression)` to distinguish them from strings representing variable names. The remaining two arguments specify SPSS variable names. The active dataset is assumed to contain the string variables *first* and *last*. The result is stored to the new SPSS 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 SPSS variable *nvar*, which is assumed to be numeric. The result—the hyperbolic sine of the input variable—is stored to the new SPSS numeric variable *numout*.

- The `execute` method from the `Tfunction` class initiates a data pass, applying the specified functions to the active dataset. Any pending SPSS transformations are executed on the same data pass before applying the specified functions.

Figure 18-1
Resulting dataset



	nvar	first	last	strout	numout
1	0	Rick	Arturo	cust_RA	.00
2	1	Nancy	McNancy	cust_NM	1.18
3	-1	Yvonne	Walker	cust_YW	-1.18

Missing Values

The `Tfunction` class provides options for handling missing values encountered in the case data.

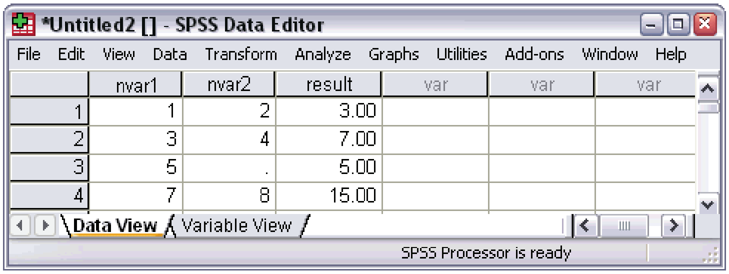
- By default, the `Tfunction` class converts user-missing values to the Python data type *None* before applying the specified functions to the data (system missing values are always converted to *None*). You can override the conversion by using `Tfunction(convertUserMissing=False)` when instantiating 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 SPSS 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 SPSS output variables specified with a numeric format and to blanks for SPSS 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 18-2
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 SPSS Developer Central.

Tracking Function Calls

By default, an SPSS 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 SPSS Developer Central, includes a number of functions that provide transformations not available with the SPSS 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. 344 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 SPSS 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*.

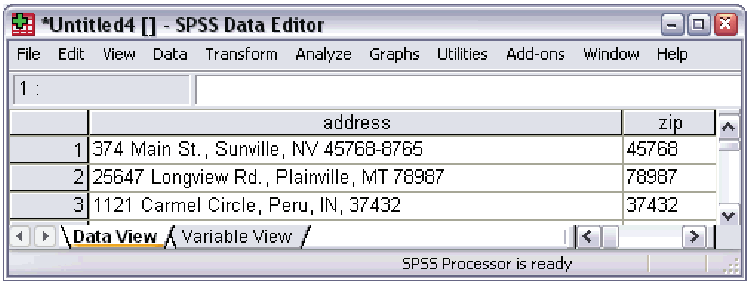
Example

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 SPSS 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 18-3
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

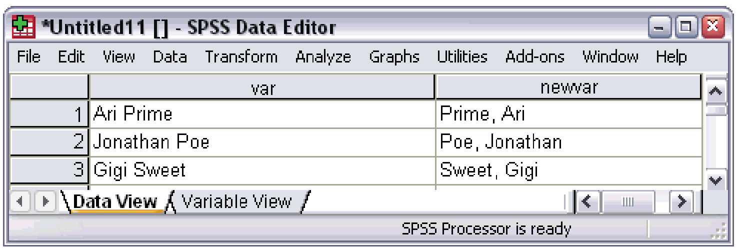
In this example, we'll use the subs function to create a string that has the form '<last name>, <first name>' from one of the form '<first name> <last name>'.

```
*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 non-whitespace 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 18-4
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 SPSS variables. The template used to construct the strings uses variable values and a variable label.

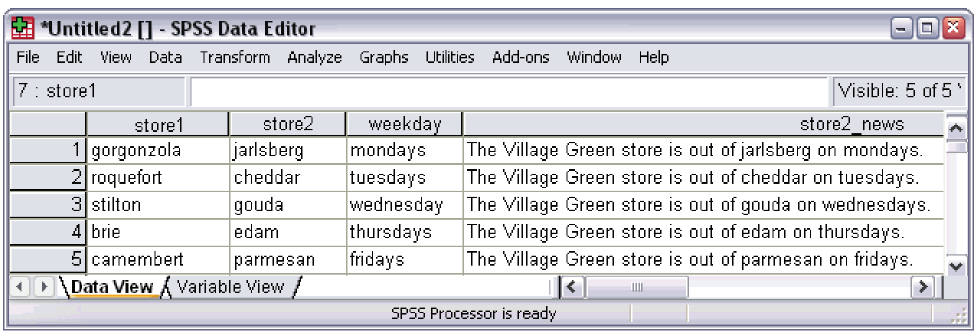
```
*python_extendedTransforms_templatesub.sps.
DATA LIST FREE / store1 (A15) store2 (A15) weekday (A10).
BEGIN DATA
gorgonzola jarlsberg mondays
roquefort cheddar tuesdays
stilton gouda wednesdays
brie edam thursdays
camembert parmesan fridays
END DATA.

VARIABLE LABELS store1 'Main St.' store2 'Village Green'.

BEGIN PROGRAM.
import trans, extendedTransforms, spssaux
from trans import const
varDict = spssaux.VariableDict()
storeList = varDict.Variables(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 available from SPSS Developer Central) to obtain the list of SPSS 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 Getting Started with the VariableDict Class in Chapter 14 on p. 263.
- The `for` loop iterates through the list of SPSS 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 *$loc*, casewise values of the SPSS variable specified by *store* will be substituted for *$type*, and casewise values of the SPSS variable *weekday* will be substituted for *$day*. The resulting string is stored to a new SPSS variable whose name is dynamically created by the expression `store+'_news'`—for example, *store1_news*.

Figure 18-5
Resulting dataset



	store1	store2	weekday	store2_news
1	gorgonzola	jarlsberg	mondays	The Village Green store is out of jarlsberg on mondays.
2	roquefort	cheddar	tuesdays	The Village Green store is out of cheddar on tuesdays.
3	stilton	gouda	wednesday	The Village Green store is out of gouda on wednesdays.
4	brie	edam	thursdays	The Village Green store is out of edam on thursdays.
5	camembert	parmesan	fridays	The Village Green store is out of parmesan on fridays.

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 SPSS 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

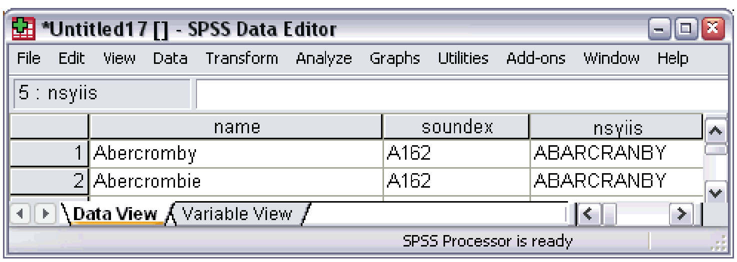
The `soundex` and `nysiis` functions implement two popular phonetic algorithms for indexing names by their sound as pronounced in English. The purpose is to encode names having the same pronunciation to the same string so that matching can occur despite differences in spelling.

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 SPSS variables *soundex* and *nsyiis*.
- The two spellings *Abercromby* and *Abercrombie* are phonetically the same and are encoded to the same value.

Figure 18-6
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 an SPSS 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 SPSS 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 SPSS 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 an SPSS 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 SPSS 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 SPSS datetime value to convert. The second argument is the pattern describing the resulting string. In this example, we're converting an SPSS 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 SPSS 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 SPSS 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 SPSS 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 an SPSS 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.

Chapter

19

Modifying and Exporting Viewer Contents

The `viewer` module, a supplementary module that you can download from SPSS Developer Central at *http://www.spss.com/devcentral*, provides programmatic access to the SPSS Viewer (on Windows systems) from Python via OLE automation. This capability is available only when working in local mode and does not provide access to the Draft Viewer. That said, it includes features to export Viewer contents and modify pivot tables (change column or row labels, make totals bold, add footnotes, change fonts or colors) beyond the formatting available in the general TableLook facility.

If you need to create custom pivot tables, use the `BasePivotTable` class (see Creating Pivot Table Output in Chapter 17 on p. 335). You may also want to consider the `tables` module (also available from SPSS Developer Central), which uses the `viewer` module and allows you to merge one pivot table into another based on matching row and column labels.

Tasks such as exporting the Viewer contents are accomplished using methods in the `viewer` module that call the necessary OLE automation interfaces for you. Modifying items in the Viewer, however, requires explicit use of the SPSS OLE automation interfaces. The example on Modifying Pivot Tables on p. 366 illustrates some of the OLE properties and methods needed to modify pivot tables.

For information on OLE automation in SPSS, see the *SPSS Base User's Guide* or the SPSS scripting and automation topics in the Help system. The examples presented in those sources use the Sax Basic language, but the object methods and properties used are part of SPSS OLE automation and are not specific to Sax Basic.

If you are familiar with scripting in Sax Basic, the transition to OLE automation with Python is relatively simple, but there are a few differences. For more information, see Migrating Sax Basic Scripts to Python in Chapter 20 on p. 379.

If you use the PythonWin IDE (a freely available IDE for working with Python on Windows), you can obtain a listing of the available OLE automation methods by choosing the COM Browser option from the Tools menu (OLE automation methods are also referred to as COM methods). The methods are listed in the SPSS Type Library and SPSS Pivot Table Type Library folders under the Registered Type Libraries folder. A listing of the COM methods is also available from any COM-aware software, such as Visual Studio or the Visual Basic environment accessed from any Microsoft Office application.

Note: To run the examples in this section, you need to download the `viewer` module from SPSS Developer Central and save it to your Python "site-packages" directory, which is typically *C:\Python24\Lib\site-packages*. You'll also need the two publicly available modules (not provided by SPSS)—`pythoncom` and `win32com.client`—that enable OLE automation with Python. These are installed with the *pywin32* package for Python 2.4, available at *http://sourceforge.net/projects/pywin32* (for example, *pywin32-205.win32-py2.4.exe* from that site). This package should be installed to your "site-packages" directory. As an added benefit, it includes installation of the PythonWin IDE.

Getting Started with the viewer Module

As an introduction to the `viewer` module, we will show a simple example of exporting the contents of the designated (current) Viewer window to a PDF file.

```
*python_viewer_export.sps.
BEGIN PROGRAM.
import spss,viewer,sys
spss.Submit(r"""
OUTPUT NEW TYPE=VIEWER.
GET FILE='c:/examples/data/Employee data.sav'.
DESCRIPTIVES ALL.
""")
spssappObj=viewer.spssapp()
try:
   spssappObj.ExportDesignatedOutput("c:/temp/myoutput.pdf",format="Pdf")
except:
   print sys.exc_info()[1]
else:
   spssappObj.CloseDesignatedOutput()
END PROGRAM.
```

- The program block utilizes the `spss` and `viewer` modules, as well as the built-in module `sys` (used here to extract information about an exception), so it includes the statement `import spss,viewer,sys`.

- The `OUTPUT NEW` command is used to create a new output window, which becomes the designated output window.
- To access the Viewer, you first create an instance of the `spssapp` class from the `viewer` module and assign it to a variable, as in `spssappObj=viewer.spssapp()`. The variable *spssappObj* contains a reference to the SPSS Application object, which enables access to the contents of the Viewer windows.
- The `ExportDesignatedOutput` method of the `spssapp` class exports the contents of the designated Viewer to the specified file in the specified format. You can export to the following formats: HTML (default), text, Excel, Word, PowerPoint, or PDF. For more information, along with a complete list of available arguments, include the statement `help(viewer.spssapp.ExportDesignatedOutput)` in a program block.
- If the save attempt fails for any reason, the `except` clause is invoked. `sys.exc_info()` returns a **tuple** of three values that provide information about the current exception. The value with an index of 1 contains the most descriptive information.
- If the export is successful, the `else` clause is executed. It calls the `CloseDesignatedOutput` method to close the designated Viewer window and open (designate) a new one.

Persistence of Objects

This section describes issues, related to the persistence of objects, that are important to be aware of when working with the `viewer` module.

Reference to the Designated Viewer Window

Working with the designated Viewer window requires having an object reference to it. This is provided by the `GetDesignatedOutput` method, from the `viewer` module. The `SaveDesignatedOutput`, `ExportDesignatedOutput`, and `CloseDesignatedOutput` methods take care of calling this method for you.

When you explicitly call `GetDesignatedOutput` to get a reference to the designated Viewer—for example, when you want to modify a pivot table—you typically store the reference to a variable for later use. If a different window becomes the designated one and you want to access the new window's contents, you'll have to call `GetDesignatedOutput` again, since the stored reference provides access to the original window, not the new one.

Working with Multiple Program Blocks

Sometimes you may have occasion to create a command syntax job that contains more than one `BEGIN PROGRAM` block. A subtlety in the way that OLE automation works, however, requires that each program block be properly initialized before OLE automation methods will work in that block. This is done automatically when you create an instance of the `spssapp` class or call any of the methods in that class, but it is not done for you by other classes in the `viewer` module. To work with OLE automation in subsequent program blocks, create a new instance of `spssapp` with something like `spssappObj=viewer.spssapp()`. This is not necessary if the code in the subsequent program block calls a method from the `spssapp` class before calling any from another class in the `viewer` module.

Modifying Pivot Tables

The `spssapp` class in the `viewer` module provides access to the SPSS application object, enabling you to modify items in the Viewer. This requires the explicit use of SPSS OLE automation objects. We will illustrate this capability with a Python user-defined function that changes the text style of specified column labels to bold for a chosen set of pivot tables.

```
def MakeColLabelBold(collabel,itemlabel):
    """Change all column labels that match a specified string
    to bold, and make this change for all pivot tables whose
    item label (title) matches a specified string.
    collabel is the string that specifies the column label
    to modify; for example "Total".
    itemlabel is the string that specifies the item label (title)
    of the pivot tables to modify; for example "Coefficients".
    """
    spssappObj = viewer.spssapp()
    objItems = spssappObj.GetDesignatedOutput().Items
    for i in range(objItems.Count):
       objItem = objItems.GetItem(i)
       if objItem.SPSSType == 5 and objItem.Label == itemlabel:
          objPivotTable = objItem.Activate()
          try:
             objColumnLabels = objPivotTable.ColumnLabelArray()
             for j in range(objColumnLabels.NumColumns):
                for k in range(objColumnLabels.NumRows):
                   if objColumnLabels.ValueAt(k,j) == collabel:
                      objColumnLabels.SelectLabelAt(k,j)
             try:
                objPivotTable.TextStyle=2
             except:
                pass
          finally:
             objItem.Deactivate()
```

- `MakeColLabelBold` is a Python user-defined function that requires two arguments, *collabel* and *itemlabel*.
- The code `spssappObj = viewer.spssapp()` creates an instance of the `spssapp` class and assigns it to the variable *spssappObj*.
- The `GetDesignatedOutput` method of the `spssapp` class returns a reference to the designated (current) Viewer window. This method is a wrapper for the OLE automation method `GetDesignatedOutputDoc` that provides some necessary initialization. Other than `GetDesignatedOutput`, the methods and properties used in `MakeColLabelBold` belong to SPSS OLE automation objects.
- The `Items` property contains a collection of all items in the designated output document (Viewer). You have to access this collection before you can access individual output items, such as pivot tables. The Python variable *objItems* contains a reference to this collection object.
- The outermost `for` loop iterates over all of the items in the designated Viewer. Each item is accessed using the `GetItem` method and tested to see if it is a pivot table (type 5) and if the object's label (table title for a pivot table) matches the

string passed in as the argument *itemlabel*. If the test expression evaluates to *true*, the item is activated using the `Activate` method.

- Whenever you activate an object and intend to deactivate it when you are done, use a `try:…finally:` block, as shown here, to ensure that if an exception is raised, the `Deactivate` method is always called. The `try` clause contains all of the code to execute against the object and the `finally` clause calls the `Deactivate` method.
- The `ColumnLabelArray` method obtains a reference to the Column Labels object. This object is a collection of column labels contained in the pivot table object.
- The inner `for` loops indexed by *j* and *k* iterate through the elements in the Column Labels object. The `ValueAt` method is used to access the value of a specified column label. If the value matches the string passed in as the argument *collabel* it is selected using the `SelectLabelAt` method.
- The inner `try` clause is executed once for each pivot table whose label (title) matches the value specified in *itemlabel*. The code `objPivotTable.TextStyle=2` sets the text style property of all currently selected cells to 2 (the value for bold type). An exception occurs when attempting to set the `TextStyle` property if there are no selected cells, meaning that no column labels in the current pivot table match the specified string. In that case, control passes to the `except` clause. Since there's no action to take, the clause contains only a `pass` statement.

Example

As an example, we will generate output from the `DESCRIPTIVES` procedure and call `MakeColLabelBold` to change the column label *Sum* to bold in the Descriptive Statistics table.

```
*python_modify_pivot_table.sps.
BEGIN PROGRAM.
import spss,samplelib_supp
spss.Submit(r"""
GET FILE='c:/examples/data/Employee data.sav'.
DESCRIPTIVES
  VARIABLES=salary,salbegin,jobtime,prevexp
  /STATISTICS=MEAN SUM STDDEV MIN MAX.
""")
samplelib_supp.MakeColLabelBold("Sum","Descriptive Statistics")
END PROGRAM.
```

- The `BEGIN PROGRAM` block starts with a statement to import the `samplelib_supp` module, which contains the definition for the `MakeColLabelBold` function.

  Note: To run this program block, you need to copy the module file *samplelib_supp.py* from *\examples\python* on the accompanying CD to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*. The `samplelib_supp` module uses functions in the `spssaux`, `viewer`, `spssdata`, and `namedtuple` modules, so you will also need copies of these modules in your "site-packages" directory. You can download them from SPSS Developer Central at *http://www.spss.com/devcentral*.

Using the viewer Module from a Python IDE

The `viewer` module is designed for optional use with a Python IDE (Integrated Development Environment). This allows you to develop and test code that operates on Viewer objects while still taking full advantage of the benefits that IDEs have to offer. The steps to enable this are as follows:

▶ Start up SPSS as you normally would.

▶ From a Python IDE, create an instance of the `spssapp` class with the argument *standalone* set to *true*, as in:

```
import viewer
spssappObj=viewer.spssapp(standalone=True)
```

With `standalone=True`, the `spssapp` instance attaches to the SPSS instance that you started up manually, as opposed to the SPSS instance that is automatically created when you run `import spss` from a Python IDE (an instance of SPSS that has no Viewer). Subsequent OLE automation code run from the IDE will act on the objects in the designated Viewer. Note, however, that in this mode of operation, output from SPSS commands submitted from Python is not directed to the Viewer but rather to the IDE's output window. In this regard, the mode with `standalone=True` is intended primarily for testing code that manipulates Viewer objects. When the code is ready for use, it should be included in a `BEGIN PROGRAM` block.

Note: Although the mode with `standalone=True` is primarily intended for use with a Python IDE, it can be used with any separate Python process, like the Python interpreter.

Chapter

20

Tips on Migrating Command Syntax, Macro, and Scripting Jobs to Python

Exploiting the power that the SPSS-Python Integration Plug-In offers may mean converting an existing command syntax job, macro, or Sax Basic script to Python. This is particularly straightforward for command syntax jobs, since you can run SPSS command syntax from Python using a function from the `spss` module (available once you install the plug-in). Converting macros and Sax Basic scripts is more complicated, since you need to translate from either the macro language or Sax Basic to Python, 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="c:\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='c:/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 SPSS command syntax and submits the syntax to SPSS for processing. By wrapping the command syntax in triple quotes, you can specify blocks of SPSS 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. 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, a standard practice for Windows and DOS. That said, it is a good practice to use forward slashes (/) in file paths, since you may at times forget to use raw mode, and SPSS accepts a forward slash (/) for any backslash in a file specification. For more information, see Using Raw Strings in Python in Chapter 13 on p. 241.

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, 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 Using a

Python IDE in Chapter 12 on p. 226. To run your job from SPSS, 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='c:/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 an SPSS command syntax job and converted it into a Python job. As it stands, the Python job does exactly what the SPSS 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 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'c:/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 Dynamically Specifying Command Syntax Using String Substitution in Chapter 13 on p. 238.
- 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. For more information, see Appendix A on p. 424.
- 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 SPSS command syntax renders SPSS macros obsolete for most purposes. Macros are still important, however, for passing information from a `BEGIN PROGRAM` block so that it is available to SPSS command syntax outside of the block. For more information, see Mixing Command Syntax and Program Blocks in Chapter 12 on p. 223. 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 an SPSS macro to Python, but a few general rules will help get you started:

- The analog of an SPSS macro 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 Creating User-Defined Functions in Python in Chapter 13 on p. 243.
- A block of SPSS 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 an SPSS 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= 'c:\examples\data\employee data.sav'
        RPath= 'c:\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 SPSS command syntax and submits the syntax to SPSS for processing. Enclosing the command syntax in triple quotes allows you to specify a block of SPSS 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 Dynamically Specifying Command Syntax Using String Substitution in Chapter 13 on p. 238.

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, 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. 243. To learn how to run code from a Python IDE, see Using a Python IDE on p. 226.

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'c:/examples/data/Employee data.sav',
                 r'c:/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, you need to copy the module file *samplelib.py* from *\examples\python* on the accompanying CD to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*. Because the `samplelib` module uses functions in the `spss` module, it includes an `import spss` statement.

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 dictionary that occur during the course of its execution, while a macro cannot.

Migrating Sax Basic Scripts to Python

With the functionality provided by the SPSS-Python Integration Plug-In, you can accomplish many tasks that previously required the SPSS scripting facility for accessing dictionary information or case data. When extended with two publicly available modules (not provided by SPSS), Python can access OLE automation (COM) objects, allowing you to manipulate output that appears in the SPSS Viewer. You'll still want to make use of the scripting facility for autoscripts, but you may want to consider migrating other scripts to Python. In this section, we'll focus on migrating scripts that manipulate Viewer objects, since there are some general considerations to be aware of.

In order to access SPSS Viewer objects from Python, you'll need the two publicly available modules `pythoncom` and `win32com.client`. These are installed with the *pywin32* package for Python 2.4, available at*http://sourceforge.net/projects/pywin32* (for example, *pywin32-205.win32-py2.4.exe* from that site). This package should be installed to your "site packages" directory, typically *C:\Python24\Lib\site-packages*. As an added benefit, it includes installation of the PythonWin IDE.

These extension modules allow you to access the SPSS Application object, from which you can access the full suite of SPSS OLE automation methods. In practice, using OLE automation from Python is most useful for accessing and manipulating objects in the Viewer, since other tasks, such as accessing dictionary information or case data, are better accomplished using Python functions from the `spss`, `spssaux`, and `spssdata` modules. If you're interested in learning how to work with dictionary information or case data in Python, see Working with Variable Dictionary Information on p. 254 or Working with Case Data in the Active Dataset on p. 275.

To facilitate using OLE automation to manipulate Viewer objects from Python, SPSS has provided the `viewer` module, a supplementary module available for download from SPSS Developer Central at *http://www.spss.com/devcentral*. This capability is available only when you are working in local mode and does not provide access to the Draft Viewer. That said, when converting Sax Basic scripts that manipulate Viewer objects, you'll want to make use of the functionality in this module, as described in the example that follows. Once you've downloaded the `viewer` module from SPSS Developer Central, save it to your Python "site-packages" directory. You'll need the `viewer` module to run the example in this section. For more information about using the `viewer` module than is provided here, see Modifying and Exporting Viewer Contents on p. 363.

As an example of converting a Sax Basic script to Python, we'll consider a modified version of the Traffic Light script that is distributed with the SPSS product. After running this popular script, cells in a selected pivot table are green if their value exceeds a specified limit, red if their value is less than a specified minimum, and yellow if their value lies between the specified minimum and maximum. In this version, we'll reverse the coloring scheme so that green specifies that a cell is less than a threshold and red specifies that it's above a threshold. We won't use yellow for cells with intermediate values. The Sax Basic script follows.

```
'BEGIN DESCRIPTION
'This is a modified version of the Traffic Light script by Bernhard Witt.
'Cells with values greater than high-margin will be colored red.
'Cells with values less than low-margin will be colored green.
'High-margin is set at 20 and low-margin is set at 10.
'Requirements: The pivot table to color should be selected.
'END DESCRIPTION

Option Explicit
Const TotalStr ="Total"
Const red = RGB(178,34,34)
Const green = RGB(60, 179, 113)
Const white = RGB(255,255,255)

Sub Main
Dim objItems As ISpssItems
Dim objItem As ISpssItem
Dim objPivotTable As PivotTable
Dim objDataCells As ISpssDataCells
Dim objRowLabels As ISpssLabels
Dim objColLabels As ISpssLabels
Dim lngNumRows As Long, lngNumColumns As Long
Dim lowMargin As Single, highMargin As Single
Dim I As Integer, J As Integer

lowMargin = 10
highMargin = 20
Set objItems = objSpssApp.GetDesignatedOutputDoc.Items

For I = 0 To objItems.Count - 1
  Set objItem = objItems.GetItem(I)
  If objItem.SPSSType = 5 And objItem.Selected = True Then
    Set objPivotTable = objItem.Activate
    Exit For
  End If
Next I

Set objDataCells = objPivotTable.DataCellArray
Set objRowLabels = objPivotTable.RowLabelArray
Set objColLabels = objPivotTable.ColumnLabelArray
lngNumRows = objDataCells.NumRows
lngNumColumns = objDataCells.NumColumns

For I=0 To lngNumRows-1
  If InStr (objRowLabels.ValueAt(I,objRowLabels.NumColumns-1),TotalStr)=0 Then
    For J=0 To lngNumColumns-1
      If InStr (objColLabels.ValueAt(objColLabels.NumRows-1,J),TotalStr)=0 Then
        If Len(objDataCells.ValueAt(I,J)) > 0 Then
          If objDataCells.ValueAt(I,J) <= lowMargin Then
            objDataCells.BackgroundColorAt(I,J) = green
          ElseIf objDataCells.ValueAt(I,J) >= highMargin Then
            objDataCells.BackgroundColorAt(I,J) = red
          End If
        Else
          objDataCells.BackgroundColorAt(I,J) = white
        End If
      End If
    Next
  End If
Next
objItem.Deactivate
End Sub
```

We'll assume that you're familiar with the Sax Basic language and will focus on the details of the conversion to Python. When you're converting from Sax Basic to Python, keep in mind that Sax Basic is not case sensitive, but Python is. The above script translates into the following Python code, which is shown here enclosed within a `BEGIN PROGRAM-END PROGRAM` block that can be run from SPSS. You can also run the code from a Python IDE. For more information, see Using a Python IDE in Chapter 12 on p. 226.

```
*python_color_cells.sps.
BEGIN PROGRAM.
import viewer
TotalStr ="Total"
red = 178+34*2**8+34*2**16
green = 60+179*2**8+113*2**16
white = 255+255*2**8+255*2**16
lowMargin = 10
highMargin = 20

spssappObj = viewer.spssapp()
objItems = spssappObj.GetDesignatedOutput().Items

for I in range(objItems.Count):
  objItem = objItems.GetItem(I)
  if objItem.SPSSType==5 and objItem.Selected:
      objPivotTable=objItem.Activate()
      break
try:
  objDataCells=objPivotTable.DataCellArray()
  objRowLabels=objPivotTable.RowLabelArray()
  objColLabels=objPivotTable.ColumnLabelArray()
  lngNumRows=objDataCells.NumRows
  lngNumColumns=objDataCells.NumColumns

  for I in range(lngNumRows):
    if objRowLabels.ValueAt(I,objRowLabels.NumColumns-1).find(TotalStr)==-1:
      for J in range(lngNumColumns):
        if objColLabels.ValueAt(objColLabels.NumRows-1,J).find(TotalStr)==-1:
          if objDataCells.ValueAt(I,J):
            if objDataCells.ValueAt(I,J) <= lowMargin:
              objDataCells.SetBackgroundColorAt(I,J,green)
            elif objDataCells.ValueAt(I,J) >= highMargin:
              objDataCells.SetBackgroundColorAt(I,J,red)
          else:
            objDataCells.SetBackgroundColorAt(I,J,white)
finally:
    objItem.Deactivate()
END PROGRAM.
```

- Since the Python code makes use of functions and methods in the `viewer` module, it is included on the `import` statement.
- Unlike Sax Basic, Python doesn't have an RGB function, so you have to provide integer representations of the RGB values you want. For example, the integer for RGB(178,34,34) is calculated from the expression: 178+34*2**8+34*2**16.

- To access the SPSS OLE automation methods, you create an instance of the `spssapp` class from the viewer module, as in `viewer.spssapp()`. We've assigned the instance to the variable *spssappObj*, which then contains a reference to the SPSS Application object. Since Python is a dynamically typed language, you don't have to declare variables before assigning values to them or use special instructions, such as `Set` in Sax Basic, for object references.
- The `GetDesignatedOutput` method of the `spssapp` class returns a reference to the designated (current) Viewer window. In Sax Basic, you would use the `GetDesignatedOutputDoc` method. The `GetDesignatedOutput` method is a wrapper for `GetDesignatedOutputDoc` that provides some necessary initialization.
- Object properties (at least properties that don't require arguments) are read and set the same in Python as in Sax Basic. For example, `objItems.Count` returns the `Count` property of the `objItems` collection.
- Object methods, for methods called with arguments, are invoked the same in Python as in Sax Basic. For example, `objItem = objItems.GetItem(I)` calls the `GetItem` method of the `objItems` collection and returns a reference to the *i*th item in the collection.
- When calling a method that doesn't take arguments, Python requires an empty set of parentheses. The parentheses let Python know that you're referring to a function and not a property of an object. For example, `objPivotTable = objItem.Activate()` calls the `Activate` method of a Viewer item. The same code in Sax Basic would look like `objPivotTable = objItem.Activate`, with no parentheses. If you omit the parentheses in Python, you get a reference to the `Activate` method instead of calling the method.
- Whenever you activate an object, intending to deactivate it when you are done, use a `try:…finally:` block, as is done here, to ensure that if an exception is raised, the `Deactivate` method is always called. The `try` clause contains all of the code to execute against the object, and the `finally` clause calls the `Deactivate` method.
- Object properties requiring arguments, such as the row and column indices of a cell, become methods in Python so that the arguments can be properly passed. The code for retrieving values of such properties looks the same in Python as in Sax Basic, but you're actually invoking a method in Python as opposed to simply accessing a property in Sax Basic. For example, `objDataCells.ValueAt(I,J)` retrieves the `ValueAt` property for the `(I,J)` element of an array—in this case, the data cell array of the pivot table.

Setting the value of such a property in Python requires special handling. Consider the `BackgroundColorAt` property for data cells used in the Sax Basic code sample above. It requires the row and column indices of the associated cell as arguments. In Python, if you try to set the `BackgroundColorAt` property of a cell with code such as `objDataCells.BackgroundColorAt(I,J) = value`, you will cause an exception because Python thinks you're trying to set the function `BackgroundColorAt` to a value, which is not allowed. (Remember, Python treats a name followed by a pair of parentheses as a function.) To set a property that has arguments, use the `Set` form of the method associated with the property and pass the value to set as an additional parameter. In the present example, `objDataCells.SetBackgroundColorAt(I,J,green)` sets the `BackgroundColorAt` property of the `(I,J)` cell of the data cell array of the pivot table to the value specified by the variable *green* (the integer representation for the desired shade of green). This is accomplished using the `SetBackgroundColorAt` method.

Note: In order for the `Set` form of a method (such as `SetBackgroundColorAt`) to work, you may need to run the utility program *makepy.py*, located in the client subdirectory where you installed the `win32com.client` module—for example, *C:\Python24\Lib\site-packages\win32com\client\makepy.py*. If you use the PythonWin IDE, you can launch *makepy.py* from the *COM Makepy utility* item on the Tools menu. Once the *makepy* utility is launched, select the most recent version of the SPSS Type Library that you need (such as the SPSS Pivot Table Type Library, used in the present example), and click OK. Repeat for each SPSS Type Library that you need. If you don't know which libraries you need, you can simply repeat the process for each SPSS Type Library listed. You need run the utility only once for each library.

Example

As a concrete example of coloring a pivot table using the Python code presented above, do the following:

- ▶ Run the command syntax file *python_color_cells_data.sps*, located in the *\examples\commands* folder on the accompanying CD. It prepares a dataset and then runs a `SUMMARIZE` command to generate a suitable pivot table.
- ▶ Select the Percentage Responding Strongly Negative table from the results of the `SUMMARIZE` command.

▶ Run the command syntax file *python_color_cells.sps*, located in the *\examples\commands* folder on the accompanying CD.

After running this command syntax file, you should notice that cells whose values are less than 10 are green and those with values greater than 20 are red. The dataset used to generate this table contains responses from a customer satisfaction survey conducted by a local store chain. Customers were asked to rate satisfaction in a number of categories, such as price, variety, and service. In one analysis of the data, management was interested in the percentage of respondents in each category who indicated strong dissatisfaction, with results presented by store. Values greater than 20% were deemed high enough to warrant attention, but they also wanted to highlight values less than 10% as representing good performance. Since high values have a negative connotation and low values, a positive one, it made sense to reverse the color scheme used in the standard Traffic Light script.

Chapter

21

Special Topics

Using Regular Expressions

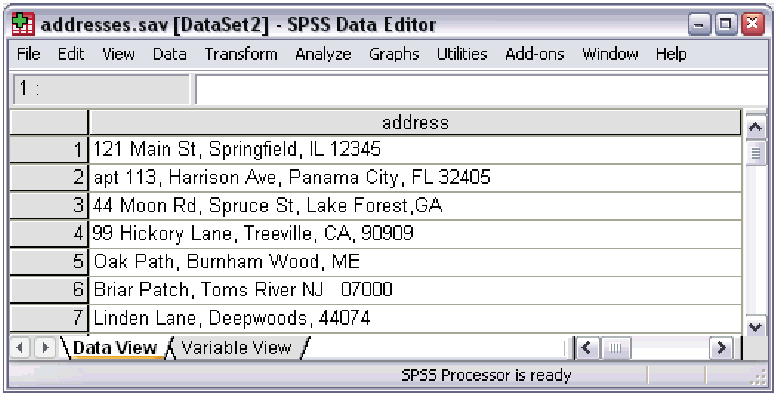
Regular expressions define patterns of characters that are matched against a string to determine if the string contains the pattern. In addition to identifying matches, you can extract the part of a string matching the pattern, replace the matched part with a specified string, or split the string apart 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 21-1
Dataset with addresses containing state codes




```
*python_re_state_lookup.sps.
BEGIN PROGRAM.
import spss, spssaux, spssdata, re
spssaux.OpenDataFile('c:/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 available for download from SPSS Developer Central at *http://www.spss.com/devcentral*.

- The `OpenDataFile` function from the `spssaux` module opens an SPSS 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 upper case 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 upper case 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 upper case 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 Using the spssdata Module in Chapter 15 on p. 291.
- 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 SPSS 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 SPSS 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, i.e., `([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 SPSS 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 instance, `stateCodes['GA']` is `'Georgia'`.

For an example of using regular expressions to select a subset of SPSS variables in the active dataset, see Using Regular Expressions to Select Variables on p. 273. For examples of using regular expressions to search for and replace patterns of characters in SPSS case data, see The search and subs Functions on p. 350.

Locale Issues

For users who need to pay attention to locale issues, a few initial points are noteworthy.

- When used with SPSS, the Python interpreter runs in the same locale as SPSS.
- Although the Python language provides the built-in module `locale` for dealing with locale issues, you should only change the locale with SPSS `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 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. This simple example illustrates the point with some accented characters used in French.

```
BEGIN PROGRAM.
import spss
spss.Submit("SET LOCALE='english'.")
list=["a","ô","é"]
print list
print " ".join(list)
END PROGRAM.
```

Result

```
['a', '\xf4', '\xe9']
a ô é
```

- 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`, and `match` functions from the Python `re` module to specify that all alphanumeric characters specific to the current locale be included in matches.

Chapter

22

SPSS for SAS Programmers

This chapter shows the SPSS code and SAS equivalents for a number of basic data management tasks. This 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; it is not intended to demonstrate how one application is better or worse than the other.

Reading Data

Both SPSS 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 examples presented in this section are discussed in greater detail in Chapter 3.

Reading Database Tables

Both SAS and SPSS 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 an SPSS data file or SAS dataset: records (rows) are cases, and fields (columns) are variables.

```
access1.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.
```

```
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 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 *c:\examples\data\dm_demo.mdb* is defined as *dm_demo*.
- Another difference that you will notice is that SPSS does not use a dataset name. This is because once the data is read, it is immediately the active dataset in SPSS. For this example, the SAS dataset is given the name *sasdata1*.
- In SPSS, 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 removes the spaces without substituting any characters. Where SAS uses all of the original variable names as labels, SPSS provides labels for only the variables not conforming to SPSS standards. So, in this example, the variable *ID* will be named *ID* in SPSS 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, with a label of *Marital Status* in both SPSS and SAS.

Reading Multiple Tables

Both SPSS 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=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.
```

```
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 both 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 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 requires that each line of SQL be quoted, square brackets are used here for clarity.

Reading Excel Files

SPSS 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='c:\examples\data\sales.xls'
   /SHEET=NAME 'Gross Revenue'
   /CELLRANGE=RANGE 'A2:I15'
   /READNAMES=on .

proc import datafile='c:\examples\data\sales.xls'
  dbms=excel2000 replace out=SASdata4;
   sheet="Gross Revenue";
   range="A2:I15";
   getnames=yes;
   run;
```

- Both languages require the name of the Excel file, worksheet name, and range of cells.
- Both provide the choice of reading the top row of the range as variable names. SPSS 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 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 provides labels for the variable names not conforming to SPSS 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 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 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 SPSS 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 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 code and vice versa for the SAS code.

Reading Text Data

Both SPSS and SAS can read a wide variety of text-format data files. This example shows how the two applications read comma-separated values (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 = 'C:\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 "c:\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 `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 uses the `QUALIFIER` subcommand.
- SPSS uses the format `ADATE10`, and SAS uses the format `mmddyy10` to properly read the date variable.
- The SPSS `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 SPSS and SAS can merge two or more datasets together. All of the SPSS examples presented in this section are discussed in greater detail in “Merging Data Files” on p. 88 in Chapter 4.

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='C:\examples\data\match_response1.sav'.
SORT CASES BY id.
DATASET NAME response1
GET FILE='C:\examples\data\match_response2.sav'.
SORT CASES BY id.
DATASET NAME response2.
GET FILE='C:\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 "c:\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;
```

- SPSS 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, 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 (but 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 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.

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 = 'c:\examples\data\catalog.sav'
  /FILE =' c:\examples\data\retail.sav'
    /IN = Division.
EXECUTE.
VALUE LABELS Division 0 'Catalog' 1 'Retail Store'.

libname in "c:\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 SPSS 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 in 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 `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.

Aggregating Data

SPSS 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 has a specific command for aggregating data: `AGGREGATE`.
- The SPSS `BREAK` subcommand is equivalent to the SAS `By Variable` command.
- In SPSS, 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 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. All of the SPSS examples provided here are discussed in greater detail in "Variable Properties" on p. 73 in Chapter 4.

Variable Labels

Both SPSS 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";
label Income_category = "Income category";
label opinion1="Would buy this product";
label opinion2="Would recommend this product to others";
label opinion3="Price is reasonable";
label opinion4="Better than a poke in the eye with a sharp stick";
```

- In SPSS, all of the variable labels can be defined in a single `VARIABLE LABELS` command. In SAS, a separate `label` statement is required for each variable.
- In SPSS, `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, the `label` statements must be contained in the data step.

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 relftm.
    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 SPSS, 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 need to define a format and then apply the format to specified variables within the data step.

Cleaning and Validating Data

Real data frequently contain real errors—and SPSS and SAS both have features that can help identify invalid or suspicious values. All of the SPSS 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 SPSS is equivalent to `if pregnant then do` in SAS. As in the SAS example, you could simplify the SPSS code to `DO IF pregnant`, since this resolves to Boolean *true* if the value of *pregnant* is 1.

- END IF in SPSS is equivalent to end in SAS in this example.
- To display a frequency table of *valueCheck*, SPSS 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 SPSS 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='c:\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 "c:\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 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 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 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 retains the duplicates but does not include them in reports or analyses unless you turn off filtering (but you could use `SELECT IF` to delete instead of filter unselected cases). SPSS displays these records in the Data Editor with a slash through the row number.

Transforming Data Values

In both SPSS 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 examples presented in this section are discussed in greater detail in “Transforming Data Values” on p. 112 in Chapter 4.

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!'
        ;
  run;
data recode;
  format opinion1 opinion2_new opfmt.;
  input opinion1 opinion2;
  if opinion2=1 then opinion2_new=5;
  else if opinion2=2 then opinion2_new=4;
  else if opinion2=4 then opinion2_new=2;
  else if opinion2=5 then opinion2_new=1;
  else opinion2_new=opinion2;
cards;
1 5
2 4
3 3
4 2
5 1
;
run;
```

- SPSS uses a single `RECODE` command to create a new variable *opinion2_new* with the recoded values of the original variable *opinion2*.
- SAS uses a series of `if/else if/else` statements to assign the recoded values, which requires a separate conditional statement for each value.
- `ELSE = COPY` in the SPSS `RECODE` command covers any values not explicitly specified and copies the original values to the new variable. This is equivalent to the last `else` statement in the SAS code.

Banding Data

Creating a small number of discrete categories from a continuous scale variable is sometimes referred to as **banding**. For example, you can band salary data into a few salary range categories.

Although it is not difficult to write code in SPSS or SAS to band a scale variable into range categories, in SPSS we recommend that you use the Visual Bander, 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 banding methods and can automatically generate descriptive labels for the banded categories. The SPSS command syntax in this example was generated by the Visual Bander.

```
*visual_bander.sps.
GET FILE = 'c:\examples\data\employee data.sav'.
***commands generated by Visual Bander***.
RECODE salary
   ( MISSING = COPY ) ( LO THRU 25000 =1 ) ( LO THRU 50000 =2 )
   ( LO THRU 75000 =3 ) ( LO THRU HI = 4 )
   INTO salary_category.
VARIABLE LABELS salary_category 'Current Salary (Banded)'.
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 "c:\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 (Banded)";
  if 0<salary and salary<=25000 then salary_category=1;
  else if 25000<salary and salary<=50000 then salary_category=2;
  else if 50000<salary and salary<=75000 then salary_category=3;
  else if 75000<salary then salary_category=4;
  else salary_category=salary;
run;
```

- The SPSS Visual Bander 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 banded category.

- As in the recoding example, SAS uses a series of `if/else if/else` statements to accomplish the same thing.
- The SPSS `RECODE` command supports the keywords `LO` and `HI` to ensure that no values are left out of the banding 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 SPSS 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 var2 var3 var4;
  Square_Root=sqrt(var4);
  Remainder=mod(var4,3);
  x=nmiss(var1,var2,var3,var4);
  if x<=1 then Average=mean(var1,var2,var3,var4);
  Valid_Values=4-x;
  Trunc_Mean=int(mean(var1,var2,var3,var4));
cards;
1 . 3 4
5 6 7 8
9 . . 12
;
run;
```

- SPSS and SAS use the same function names for the square root (`SQRT`) and remainder (`MOD`) functions.
- SPSS 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 `nmiss` and then use this information in an `if` statement prior to calculating the function.
- The SPSS `NVALID` function returns the number of nonmissing values in an argument list. To achieve comparable functionality with SAS, you need to use the `nmiss` function to calculate the number of missing values and then subtract that value from the total number of arguments.
- The SAS `int` function is equivalent to the SPSS `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 SPSS 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 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 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 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.

data new;
   input tel1 4. tel2 4. tel3 4.;
   telephone=
     (translate(right(put(tel1,$3.)),'0',' '))||"-"||
     (translate(right(put(tel2,$3.)),'0',' '))||"-"||
     (translate(right(put(tel3,$4.)),'0',' '))
  ;
cards;
111 222 3333
222 333 4444
333 444 5555
;
run;
```

- SPSS uses the `CONCAT` function to concatenate strings, and SAS uses “||” for concatenation.
- The SPSS `STRING` function converts a numeric value to a character value, like the SAS `put` function.
- The SPSS `N` format converts spaces to zeroes, like the SAS `translate` function.

String Parsing

In addition to being able to combine strings, you can take them apart. For example, you could 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, INDEX(telephone, "-")-1), F5).
COMPUTE tel2 =
  NUMBER(SUBSTR(telephone, INDEX(telephone, "-")+1,
  RINDEX(telephone, "-")-(INDEX(telephone, "-")+1)), F5).
COMPUTE tel3 =
  NUMBER(SUBSTR(telephone, RINDEX(telephone, "-")+1), F5).
EXECUTE.
FORMATS tel1 tel2 (N3) tel3 (N4).
```

```
data new;
   input telephone $16.;
   format tel1 tel2 3. tel3 z4.;
   tel1=substr(compress(telephone,'- '),1,3);
   tel2=substr(compress(telephone,'- '),4,3);
   tel3=substr(compress(telephone,'- '),7,4);
cards;
111-222-3333
222 - 333 - 4444
333-444-5555
444 - 555-6666
555-666-0707
;
```

```
run;
```

- SPSS uses substring (`SUBSTR`) and index (`INDEX`, `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 `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 SPSS 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, SPSS 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=(EndDateTime-StartDateTime)/60/60;
      minutes=(EndTime-StartTime)/60;
cards;
3/01/2003, 4/10/2003
01-MAR-2003 12:00, 02-MAR-2003 12:00
09:30, 10:15
;
run;
```

- SPSS 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, and if you want the difference in some other time measurement unit, you must provide the necessary calculations.

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—and 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 SPSS 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 `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 mmddyyy10.;
     input StartDateTime & : DATETIME25. ;
```

```
      dateonly=datepart(StartDateTime);
      hour=hour(StartDateTime);

      DayofWeek=weekday(dateonly);
      quarter=qtr(dateonly);
cards;
29-OCT-2003 11:23:02
;
run;
```

- SPSS 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 `XDATE.DATE` function is equivalent to the SAS `datepart` function. The SPSS `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 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 SPSS-Python Integration Plug-In 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 SPSS-Python Integration Plug-In works with SPSS release 14.0.1 or later and requires only SPSS Base. The SPSS examples in this section assume some familiarity with Python and the way it can be used with SPSS command syntax. For more information, see Getting Started with Python Programming in SPSS in Chapter 12 on p. 213.

Creating Custom Functions

Both SPSS 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, 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 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 'c:\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 SPSS commands using the `Submit` function from the `spss` module. The function accepts a quoted string representing an SPSS command and submits the command text to SPSS 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 or the `means` procedure in SAS, and *dataset* specifies the dataset. The expressions `%(product)s` and `%(dataset)s` in the SPSS code specify to substitute a string representation of the value of *product* and the value of *dataset*, respectively. For more information, see Dynamically Specifying Command Syntax Using String Substitution in Chapter 13 on p. 238.

- In SPSS, 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, 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("c:/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, see Creating User-Defined Functions in Python on p. 243. For more on the `Submit` function, see Submitting Commands to SPSS on p. 215, or see Appendix A.

Job Flow Control

Both SPSS 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 is to drive SPSS command syntax from Python using the `Submit` function from the `spss` module. Information needed to determine the flow is retrieved from SPSS 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='c:/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 'c:\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 and SAS use a conditional expression to determine whether to generate the report. In the case of SPSS, this is a Python `if` statement, since the execution is being controlled from Python. In SPSS, 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 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 available for download from SPSS Developer Central at *http://www.spss.com/devcentral*. It builds on the functionality available in the

SPSS-Python Integration Plug-In to provide a number of features that simplify the task of working with case data. For more information, see Using the spssdata Module in Chapter 15 on p. 291.

- 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 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 SPSS and SAS have the ability to create global macro variables. In SAS, this is done with `symput`, whereas in SPSS, 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='c:/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 'c:\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 an SPSS 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 an SPSS macro that is then available to SPSS command syntax outside of a BEGIN PROGRAM block. The convention in SPSS—followed in this example—is to prefix the name of a macro with the ! character, although this is not required. For more information on the SetMacroValue function, see Appendix A.
- 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 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 version doesn't need to create a dataset. It uses the spssdata module to read the data in *regionsales* and create a separate SPSS macro for each case read. For more information on the spssdata module, see Using the spssdata Module on p. 291.

Setting Global Macro Variables to Values from the Environment

SPSS 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, 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 C:\Work\SAS\prog1.sas -sysparm 4.5
```

- In the SPSS 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, 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 an SPSS command syntax job using values retrieved from the environment. And you can use the `SetMacroValue` function (discussed in the previous example) to create an SPSS macro that resolves to the retrieved value and can be used outside of a `BEGIN PROGRAM` block. In the current example, an SPSS macro named `!rate` is created from the value of an environment variable named *rate*.

Appendix

A

Python Functions and Classes

The SPSS-Python package contains functions and classes that facilitate the process of using Python programming features with SPSS, including those that:

Build and run SPSS command syntax

- spss.Submit

Get information about data files in the current SPSS session

- spss.CreateXPathDictionary
- spss.EvaluateXPath
- spss.GetCaseCount
- spss.GetVarAttributeNames
- spss.GetVarAttributes
- spss.GetVariableCount
- spss.GetVariableFormat
- spss.GetVariableLabel
- spss.GetVariableMeasurementLevel
- spss.GetVariableName
- spss.GetVariableType
- spss.GetVarMissingValues
- spss.GetWeightVar
- spss.GetXmlUtf16

Get data, add new variables, and append cases to the active dataset

- spss.Cursor

Get output results

- spss.EvaluateXPath
- spss.GetXmlUtf16

Create custom pivot tables and text blocks

- spss.BasePivotTable
- spss.TextBlock

Create macro variables

- spss.SetMacroValue

Get error information

- spss.GetLastErrorLevel
- spss.GetLastErrorMessage

Manage multiple versions of the SPSS-Python Integration Plug-in

- spss.GetDefaultPlugInVersion
- spss.SetDefaultPlugInVersion
- spss.ShowInstalledPlugInVersions

To display a list of all available SPSS Python functions, with brief descriptions, use the Python `help` function, as in:

```
BEGIN PROGRAM.
import spss
help(spss)
END PROGRAM.
```

spss.BasePivotTable Class

spss.BasePivotTable(title,templateName,outline,isSplit,caption). *Provides the ability to create custom pivot tables that can be displayed in the SPSS Viewer or written to an external file using the SPSS Output Management System.*

- The argument *title* is a string that specifies the title that appears with the table. Each table associated with a set of output (as specified in a `StartProcedure-EndProcedure` block) should have a unique *title*.

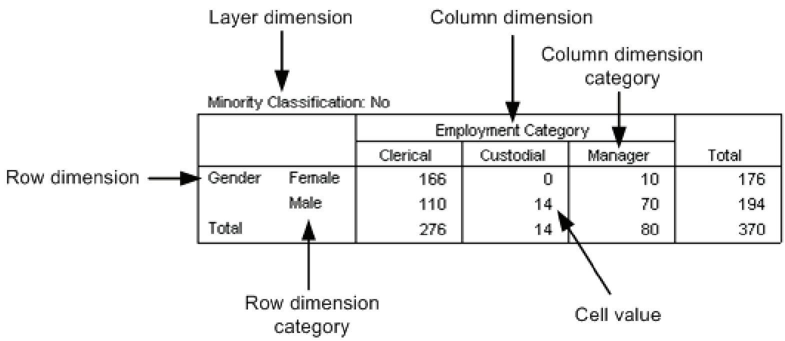
- The argument *templateName* is a string that specifies the OMS (Output Management System) table subtype for this table. It must begin with a letter and have a maximum of 64 characters. Unless you are routing this pivot table with OMS or need to write an autoscript for this table, you will not need to keep track of this value, although you do have to provide a value that meets the stated requirements.
- The optional argument *outline* is a 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.
- The optional Boolean argument *isSplit* specifies whether to enable split processing when creating pivot tables from data that have splits. By default, split processing is enabled. To disable split processing for pivot tables, specify `isSplit=False`. If you are creating a pivot table from data that has splits and you want separate results displayed for each split group, you will want to make use of the spss.SplitChange function. In the absence of calls to `spss.SplitChange`, *isSplit* has no effect.
- The optional argument *caption* is a string that specifies a table caption.

An instance of the `BasePivotTable` class can only be used within a `StartProcedure-EndProcedure` block or within a custom procedure class based on the `spss.BaseProcedure` class. For an example of creating a pivot table using `spss.StartProcedure-spss.EndProcedure`, see Creating Pivot Tables with the SimplePivotTable Method on p. 427. For an example of creating a pivot table using a class based on the `spss.BaseProcedure` class, see spss.BaseProcedure Class on p. 456.

Figure A-1 shows the basic structural components of a pivot table. Pivot tables consists of one or more dimensions, each of which can be of the type row, column, or layer. In this example, there is one dimension of each type. 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.

Figure A-1
Pivot table structure



Creating Pivot Tables with the SimplePivotTable Method

For creating a pivot table with a single row dimension and a single column dimension, the `BasePivotTable` class provides the SimplePivotTable method. The arguments to the method provide the dimensions, categories, and cell values. No other methods are necessary in order to create the table structure and populate the cells. If you require more functionality than the `SimplePivotTable` method provides, there are a variety of methods to create the table structure and populate the cells. For more information, see General Approach to Creating Pivot Tables on p. 429.

Example

```
import spss
spss.StartProcedure("mycompany.com.demoProc")
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

table.SimplePivotTable(rowdim = "row dimension",
                       rowlabels = ["first row","second row"],
                       coldim = "column dimension",
                       collabels = ["first column","second column"],
                       cells = [11,12,21,22])
spss.EndProcedure()
```

Result

Figure A-2
Simple pivot table

Table Title

row dimension	column dimension	
	first column	second column
first row	11	12
second row	21	22

- This example shows how to generate a pivot table within a `spss.StartProcedure-spss.EndProcedure` block. The argument to the `StartProcedure` function specifies a name to associate with the output. This is the name that appears in the outline pane of the Viewer associated with the output—in this case, *mycompany.com.demoProc*. It is also the command name associated with this output when routing output with OMS, as well as the name associated with this output for use with autoscripts.

  Note: In order that names associated with output do not conflict with names of existing SPSS commands (when working with OMS or autoscripts), SPSS recommends that they have the form *yourcompanyname.com.procedurename*. For more information, see spss.StartProcedure Function on p. 506.

- You create a pivot table by first creating an instance of the `BasePivotTable` class and storing it to a variable—in this case, the variable *table*.
- The `SimplePivotTable` method of the `BasePivotTable` instance is called to create the structure of the table and populate its cells. Row and column labels and cell values can be specified as character strings or numeric values. They can also be specified as a `CellText` object. `CellText` objects allow you to specify that category labels be treated as variable names or variable values, or that cell values be displayed in one of the numeric formats used in SPSS pivot tables, such as the format for a mean. When you specify a category as a variable name or variable value, pivot table display options such as display variable labels or display value labels are honored.
- Numeric values specified for cell values, row labels, or column labels, are displayed using the default format for the pivot table. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer. You can change the default format using the SetDefaultFormatSpec method.
- `spss.EndProcedure` marks the end of output creation.

General Approach to Creating Pivot Tables

The `BasePivotTable` class provides a variety of methods for creating pivot tables that cannot be created with the SimplePivotTable method. The basic steps for creating a pivot table are:

- Create an instance of the `BasePivotTable` class.
- Add dimensions.
- Define categories.
- Set cell values.

Once a cell value has been set, you can access its value. This is convenient for cell values that depend on the value of another cell. For more information, see Using Cell Values in Expressions on p. 435.

Step 1: Adding Dimensions

You add dimensions to a pivot table with the Append or Insert method.

Example: Using the Append Method

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
coldim=table.Append(spss.Dimension.Place.column,"coldim")
rowdim1=table.Append(spss.Dimension.Place.row,"rowdim-1")
rowdim2=table.Append(spss.Dimension.Place.row,"rowdim-2")
```

- The first argument to the `Append` method specifies the type of dimension, using one member from a set of built-in object properties: `spss.Dimension.Place.row` for a row dimension, `spss.Dimension.Place.column` for a column dimension, and `spss.Dimension.Place.layer` for a layer dimension.
- The second argument to `Append` is a string that specifies the name used to label this dimension in the displayed table.
- Although not required to append a dimension, it's good practice to store a reference to the newly created dimension object in a variable. For instance, the variable *rowdim1* holds a reference to the object for the row dimension named *rowdim-1*. Depending on which approach you use for setting categories, you may need this object reference.

Figure A-3
Resulting table structure

		coldim
rowdim-1	rowdim-2	

The order in which the dimensions are appended determines how they are displayed in the table. Each newly appended dimension of a particular type (row, column, or layer) becomes the current innermost dimension in the displayed table. In the example above, *rowdim-2* is the innermost row dimension since it is the last one to be appended. Had *rowdim-2* been appended first, followed by *rowdim-1*, *rowdim-1* would be the innermost dimension.

Note: Generation of the resulting table requires more code than is shown here.

Example: Using the Insert Method

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
rowdim1=table.Append(spss.Dimension.Place.row,"rowdim-1")
rowdim2=table.Append(spss.Dimension.Place.row,"rowdim-2")
rowdim3=table.Insert(2,spss.Dimension.Place.row,"rowdim-3")
coldim=table.Append(spss.Dimension.Place.column,"coldim")
```

- The first argument to the `Insert` method specifies the position within the dimensions of that type (row, column, or layer). The first position has index 1 (unlike typical Python indexing that starts with 0) and defines the innermost dimension of that type in the displayed table. Successive integers specify the next innermost dimension and so on. In the current example, *rowdim-3* is inserted at position 2 and *rowdim-1* is moved from position 2 to position 3.
- The second argument to `Insert` specifies the type of dimension, using one member from a set of built-in object properties: `spss.Dimension.Place.row` for a row dimension, `spss.Dimension.Place.column` for a column dimension, and `spss.Dimension.Place.layer` for a layer dimension.
- The third argument to `Insert` is a string that specifies the name used to label this dimension in the displayed table.
- Although not required to insert a dimension, it is good practice to store a reference to the newly created dimension object to a variable. For instance, the variable *rowdim3* holds a reference to the object for the row dimension named *rowdim-3*. Depending on which approach you use for setting categories, you may need this object reference.

Figure A-4
Resulting table structure

			coldim
rowdim-1	rowdim-3	rowdim-2	

Note: Generation of the resulting table requires more code than is shown here.

Step 2: Defining Categories

There are two ways to define categories for each dimension: explicitly, using the SetCategories method, or implicitly when setting values. The explicit method is shown here. The implicit method is shown in Step 3: Setting Cell Values on p. 432.

Example

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

coldim=table.Append(spss.Dimension.Place.column,"coldim")
rowdim1=table.Append(spss.Dimension.Place.row,"rowdim-1")
rowdim2=table.Append(spss.Dimension.Place.row,"rowdim-2")

cat1=CellText.String("A1")
cat2=CellText.String("B1")
cat3=CellText.String("A2")
cat4=CellText.String("B2")
cat5=CellText.String("C")
cat6=CellText.String("D")
cat7=CellText.String("E")

table.SetCategories(rowdim1,[cat1,cat2])
table.SetCategories(rowdim2,[cat3,cat4])
table.SetCategories(coldim,[cat5,cat6,cat7])
```

- The statement `from spss import CellText` allows you to omit the `spss` prefix when specifying `CellText` objects (discussed below), once you have imported the `spss` module.
- You set categories after you add dimensions, so the `SetCategories` method calls follow the `Append` or `Insert` method calls.

- The first argument to `SetCategories` is an object reference to the dimension for which the categories are being defined. This underscores the need to save references to the dimensions you create with `Append` or `Insert`, as discussed in the previous topic.
- The second argument to `SetCategories` is a single category or a sequence of categories, each expressed as a CellText object (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`). When you specify a category as a variable name or variable value, pivot table display options such as display variable labels or display value labels are honored. In the present example, we use string objects whose single argument is the string specifying the category.
- It is a good practice to assign variables to the `CellText` objects representing the category names, since each category will often need to be referenced more than once when setting cell values.

Figure A-5
Resulting table structure

		coldim		
rowdim-1	rowdim-2	C	D	E
A1	A2			
	B2			
B1	A2			
	B2			

Note: Generation of the resulting table requires more code than is shown here.

Step 3: Setting Cell Values

There are two primary methods for setting cell values: setting values one cell at a time by specifying the categories that define the cell, or using the SetCellsByRow or SetCellsByColumn method.

Example: Specifying Cells by Their Category Values

This example reproduces the table created in the SimplePivotTable example.

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

table.Append(spss.Dimension.Place.row,"row dimension")
table.Append(spss.Dimension.Place.column,"column dimension")
```

```
row_cat1 = CellText.String("first row")
row_cat2 = CellText.String("second row")
col_cat1 = CellText.String("first column")
col_cat2 = CellText.String("second column")

table[(row_cat1,col_cat1)] = CellText.Number(11)
table[(row_cat1,col_cat2)] = CellText.Number(12)
table[(row_cat2,col_cat1)] = CellText.Number(21)
table[(row_cat2,col_cat2)] = CellText.Number(22)
```

- The Append method is used to add a row dimension and then a column dimension to the structure of the table. The table specified in this example has one row dimension and one column dimension. Notice that references to the dimension objects created by the `Append` method are not saved to variables, contrary to the recommendations in the topic on adding dimensions. When setting cells using the current approach, these object references are not needed.
- For convenience, variables consisting of `CellText` objects are created for each of the categories in the two dimensions.
- Cells are specified by their category values in each dimension. In the tuple (or list) that specifies the category values—for example, `(row_cat1,col_cat1)`—the first element corresponds to the first appended dimension (what we have named "row dimension") and the second element to the second appended dimension (what we have named "column dimension"). The tuple `(row_cat1,col_cat1)` then specifies the cell whose "row dimension" category is "first row" and "column dimension" category is "first column."
- You may notice that the example does not make use of the `SetCategories` method to define the row and column dimension category values. When you assign cell values in the manner done here—`table[(category1,category2)]`—the values provided to specify the categories for a given cell are used by the `BasePivotTable` object to build the set of categories for the table. Values provided in the first element of the tuple (or list) become the categories in the dimension created by the first method call to `Append` or `Insert`. Values in the second element become the categories in the dimension created by the second method call to `Append` or `Insert`, and so on. The order of the categories, as displayed in the table, is the order in which they are created from `table[(category1,category2)]`. In the example shown above, the row categories will be displayed in the order *"first row," "second row."*

- Cell values must be specified as CellText objects (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`).
- In this example, Number objects are used to specify numeric values for the cells. Values will be formatted using the table's default format. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer. You can change the default format using the SetDefaultFormatSpec method, or you can override the default by explicitly specifying the format, as in: `CellText.Number(22,spss.FormatSpec.GeneralStat)`. For more information, see Number Class on p. 450.

Example: Setting Cell Values by Row or Column

The SetCellsByRow and SetCellsByColumn methods allow you to set cell values for entire rows or columns with one method call. To illustrate the approach, we will use the `SetCellsByRow` method to reproduce the table created in the SimplePivotTable example. It is a simple matter to rewrite the example to set cells by column.

Note: You can only use the `SetCellsByRow` method with pivot tables that have one column dimension and you can only use the `SetCellsByColumn` method with pivot tables that have one row dimension.

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

rowdim = table.Append(spss.Dimension.Place.row,"row dimension")
coldim = table.Append(spss.Dimension.Place.column,"column dimension")

row_cat1 = CellText.String("first row")
row_cat2 = CellText.String("second row")
col_cat1 = CellText.String("first column")
col_cat2 = CellText.String("second column")

table.SetCategories(rowdim,[row_cat1,row_cat2])
table.SetCategories(coldim,[col_cat1,col_cat2])

table.SetCellsByRow(row_cat1,[CellText.Number(11),
                              CellText.Number(12)])
table.SetCellsByRow(row_cat2,[CellText.Number(21),
                              CellText.Number(22)])
```

- The `SetCellsByRow` method is called for each of the two categories in the row dimension.

- The first argument to the `SetCellsByRow` method is the row category for which values are to be set. The argument must be specified as a CellText object (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`). When setting row values for a pivot table with multiple row dimensions, you specify a list of category values for the first argument to `SetCellsByRow`, where each element in the list is a category value for a different row dimension.
- The second argument to the `SetCellsByRow` method is a list or tuple of CellText objects (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`) that specify the elements of the row, one element for each column category in the single column dimension. The first element in the list or tuple will populate the first column category (in this case, *col_cat1*), the second will populate the second column category, and so on.
- In this example, Number objects are used to specify numeric values for the cells. Values will be formatted using the table's default format. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer. You can change the default format using the SetDefaultFormatSpec method, or you can override the default by explicitly specifying the format, as in: `CellText.Number(22,spss.FormatSpec.GeneralStat)`. For more information, see Number Class on p. 450.

Using Cell Values in Expressions

Once a cell's value has been set, it can be accessed and used to specify the value for another cell. Cell values are stored as `CellText.Number` or `CellText.String` objects. To use a cell value in an expression, you obtain a string or numeric representation of the value using the toString or toNumber method.

Example: Numeric Representations of Cell Values

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

table.Append(spss.Dimension.Place.row,"row dimension")
table.Append(spss.Dimension.Place.column,"column dimension")

row_cat1 = CellText.String("first row")
row_cat2 = CellText.String("second row")
col_cat1 = CellText.String("first column")
```

```
col_cat2 = CellText.String("second column")

table[(row_cat1,col_cat1)] = CellText.Number(11)
cellValue = table[(row_cat1,col_cat1)].toNumber()
table[(row_cat2,col_cat2)] = CellText.Number(2*cellValue)
```

- The `toNumber` method is used to obtain a numeric representation of the cell with category values `("first row","first column")`. The numeric value is stored in the variable *cellValue* and used to specify the value of another cell.
- Character representations of numeric values stored as `CellText.String` objects, such as `CellText.String("11")`, are converted to a numeric value by the `toNumber` method.

Example: String Representations of Cell Values

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

table.Append(spss.Dimension.Place.row,"row dimension")
table.Append(spss.Dimension.Place.column,"column dimension")

row_cat1 = CellText.String("first row")
row_cat2 = CellText.String("second row")
col_cat1 = CellText.String("first column")
col_cat2 = CellText.String("second column")

table[(row_cat1,col_cat1)] = CellText.String("abc")
cellValue = table[(row_cat1,col_cat1)].toString()
table[(row_cat2,col_cat2)] = CellText.String(cellValue + "d")
```

- The `toString` method is used to obtain a string representation of the cell with category values `("first row","first column")`. The string value is stored in the variable *cellValue* and used to specify the value of another cell.
- Numeric values stored as `CellText.Number` objects are converted to a string value by the `toString` method.

spss.BasePivotTable Methods

Append Method

.Append(place,dimName,hideName, hideLabels). *Appends row, column, and layer dimensions to a pivot table.* You use this method, or the Insert method, to create the dimensions associated with a custom pivot table. The argument *place* specifies the type of dimension: `spss.Dimension.Place.row` for a row dimension, `spss.Dimension.Place.column` for a column dimension, and `spss.Dimension.Place.layer` for a layer dimension. The argument *dimName* is a string that specifies the name used to label this dimension in the displayed table. The argument *hideName* specifies whether the dimension name is hidden—by default, it is displayed. Use `hideName=True` to hide the name. The argument *hideLabels* specifies whether category labels for this dimension are hidden—by default, they are displayed. Use `hideLabels=True` to hide category labels.

- The order in which dimensions are appended affects how they are displayed in the resulting table. Each newly appended dimension of a particular type (row, column, or layer) becomes the current innermost dimension in the displayed table, as shown in the example below.
- The order in which dimensions are created (with the `Append` or `Insert` method) determines the order in which categories should be specified when providing the dimension coordinates for a particular cell (used when Setting Cell Values or adding Footnotes). For example, when specifying coordinates using an expression such as `(category1,category2)`, *category1* refers to the dimension created by the first call to `Append` or `Insert`, and *category2* refers to the dimension created by the second call to `Append` or `Insert`.

Example

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
coldim=table.Append(spss.Dimension.Place.column,"coldim")
rowdim1=table.Append(spss.Dimension.Place.row,"rowdim-1")
rowdim2=table.Append(spss.Dimension.Place.row,"rowdim-2")
```

Figure A-6
Resulting table structure

		coldim
rowdim-1	rowdim-2	

Examples of using the `Append` method are most easily understood in the context of going through the steps to create a pivot table. For more information, see General Approach to Creating Pivot Tables on p. 429.

Caption Method

.Caption(caption). *Adds a caption to the pivot table.* The argument *caption* is a string specifying the caption.

Example

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
table.Caption("A sample caption")
```

Footnotes Method

.Footnotes(categories,footnote). *Used to add a footnote to a table cell.* The argument *categories* is a list or tuple of categories specifying the cell for which a footnote is to be added. Each element in the list or tuple must be a CellText object (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`). The argument *footnote* is a string specifying the footnote.

Example

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

rowdim=table.Append(spss.Dimension.Place.row,"rowdim")
coldim=table.Append(spss.Dimension.Place.column,"coldim")

table.SetCategories(rowdim,spss.CellText.String("row1"))
table.SetCategories(coldim,spss.CellText.String("column1"))

table[(spss.CellText.String("row1"),spss.CellText.String("column1"))] = \
       spss.CellText.String("cell value")
table.Footnotes((spss.CellText.String("row1"),
                 spss.CellText.String("column1")),
     "Footnote for the cell specified by the categories row1 and column1")
```

- The order in which dimensions are added to the table, either through a call to `Append` or to `Insert`, determines the order in which categories should be specified when providing the dimension coordinates for a particular cell. In the present example, the dimension *rowdim* is added first and *coldim* second, so the first element of `(spss.CellText.String("row1"),spss.CellText.String("column1"))` specifies a category of *rowdim* and the second element specifies a category of *coldim*.

GetDefaultFormatSpec

.GetDefaultFormatSpec(). *Returns the default format for CellText.Number objects.* The returned value is a list with two elements. The first element is the integer code associated with the format. Codes and associated formats are listed in Table A-1 on p. 451. For formats with codes `5` (`Mean`), `12` (`Variable`), `13` (`StdDev`), `14` (`Difference`), and `15` (`Sum`), the second element of the returned value is the index of the variable in the active dataset whose format is used to determine details of the resulting format. For all other formats, the second element is the Python data type *None*. You can set the default format with the SetDefaultFormatSpec method.

- Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays cell values rounded to the nearest integer.

Example

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
print "Default format: ", table.GetDefaultFormatSpec()
```

HideTitle Method

.HideTitle(). *Used to hide the title of a pivot table.* By default, the title is shown.

Example

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
table.HideTitle()
```

Insert Method

.Insert(i,place,dimName,hideName, hideLabels). *Inserts row, column, and layer dimensions into a pivot table.* You use this method, or the Append method, to create the dimensions associated with a custom pivot table. The argument *i* specifies the position within the dimensions of that type (row, column, or layer). The first position has index 1 and defines the innermost dimension of that type in the displayed table. Successive integers specify the next innermost dimension and so on. The argument *place* specifies the type of dimension: `spss.Dimension.Place.row` for a row dimension, `spss.Dimension.Place.column` for a column dimension, and `spss.Dimension.Place.layer` for a layer dimension. The argument *dimName* is a string that specifies the name used to label this dimension in the displayed table. The argument *hideName* specifies whether the dimension name is hidden—by default, it is displayed. Use `hideName=True` to hide the name. The argument *hideLabels* specifies whether category labels for this dimension are hidden—by default, they are displayed. Use `hideLabels=True` to hide category labels.

- The argument *i* can take on the values 1, 2, ... , *n*+1 where *n* is the position of the outermost dimension (of the type specified by *place*) created by any previous calls to `Append` or `Insert`. For example, after appending two row dimensions, you can insert a row dimension at positions 1, 2, or 3. You cannot, however, insert a row dimension at position 3 if only one row dimension has been created.
- The order in which dimensions are created (with the `Append` or `Insert` method) determines the order in which categories should be specified when providing the dimension coordinates for a particular cell (used when Setting Cell Values or adding Footnotes). For example, when specifying coordinates using an expression such as `(category1,category2)`, *category1* refers to the dimension created by the first call to `Append` or `Insert`, and *category2* refers to the dimension created by the second call to `Append` or `Insert`.

  Note: The order in which categories should be specified is not determined by dimension positions as specified by the argument *i*.

Example

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
rowdim1=table.Append(spss.Dimension.Place.row,"rowdim-1")
rowdim2=table.Append(spss.Dimension.Place.row,"rowdim-2")
rowdim3=table.Insert(2,spss.Dimension.Place.row,"rowdim-3")
coldim=table.Append(spss.Dimension.Place.column,"coldim")
```

Figure A-7
Resulting table structure

			coldim
rowdim-1	rowdim-3	rowdim-2	

Examples of using the `Insert` method are most easily understood in the context of going through the steps to create a pivot table. For more information, see General Approach to Creating Pivot Tables on p. 429.

SetCategories Method

.SetCategories(dim,categories). *Sets categories for the specified dimension.* The argument *dim* is a reference to the dimension object for which categories are to be set. Dimensions are created with the Append or Insert method. The argument *categories* is a single value or a sequence of values, each of which is a CellText object (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`).

- In addition to defining category values for a specified dimension, `SetCategories` sets the pivot table object's value of the currently selected category for the specified dimension. In other words, calling `SetCategories` also sets a pointer to a category in the pivot table. When a sequence of values is provided, the currently selected category (for the specified dimension) is the last value in the sequence. For an example of using currently selected dimension categories to specify a cell, see the SetCell method.
- Once a category has been defined, a subsequent call to `SetCategories` (for that category) will set that category as the currently selected one for the specified dimension.

Example

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

rowdim=table.Append(spss.Dimension.Place.row,"rowdim")
coldim=table.Append(spss.Dimension.Place.column,"coldim")

table.SetCategories(rowdim,[spss.CellText.String("row1"),
                            spss.CellText.String("row2")])
table.SetCategories(coldim,[spss.CellText.String("column1"),
                            spss.CellText.String("column2")])
```

Examples of using the `SetCategories` method are most easily understood in the context of going through the steps to create a pivot table. For more information, see General Approach to Creating Pivot Tables on p. 429.

SetCell Method

.SetCell(cell). *Sets the value for the cell specified by the currently selected set of category values.* The argument *cell* is the value, specified as a CellText object (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`). Category values are selected using the SetCategories method as shown in the following example.

Example

```
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

rowdim = table.Append(spss.Dimension.Place.row,"rowdim")
coldim = table.Append(spss.Dimension.Place.column,"coldim")

# Define category values and set the currently selected set of
# category values to "row1" for rowdim and "column1" for coldim.
table.SetCategories(rowdim,spss.CellText.String("row1"))
table.SetCategories(coldim,spss.CellText.String("column1"))

# Set the value for the current cell specified by the currently
# selected set of category values.
table.SetCell(spss.CellText.Number(11))

table.SetCategories(rowdim,spss.CellText.String("row2"))
table.SetCategories(coldim,spss.CellText.String("column2"))

# Set the value for the current cell. Its category values are "row2"
# for rowdim and "column2" for coldim.
table.SetCell(spss.CellText.Number(22))

# Set the currently selected category to "row1" for rowdim.
table.SetCategories(rowdim,spss.CellText.String("row1"))

# Set the value for the current cell. Its category values are "row1"
# for rowdim and "column2" for coldim.

table.SetCell(spss.CellText.Number(12))
```

- In this example, Number objects are used to specify numeric values for the cells. Values will be formatted using the table's default format. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer. You can change the default format using the SetDefaultFormatSpec method, or you can override the default by explicitly specifying the format, as in:

`CellText.Number(22,spss.FormatSpec.GeneralStat)`. For more information, see Number Class on p. 450.

Figure A-8
Resulting table

rowdim	coldim	
	column1	column2
row1	11	12
row2		22

SetCellsByRow Method

.SetCellsByRow(rowlabels,cells). *Sets cell values for the row specified by a set of categories, one for each row dimension.* The argument *rowlabels* specifies the set of categories that defines the row—a single value, or a list or tuple. The argument *cells* is a tuple or list of cell values. Row categories and cell values must be specified as CellText objects (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`).

- For tables with multiple row dimensions, the order of categories in the *rowlabels* argument is the order in which their respective dimensions were added (appended or inserted) to the table. For example, given two row dimensions *rowdim1* and *rowdim2* added in the order *rowdim1* and *rowdim2*, the first element in *rowlabels* should be the category for *rowdim1* and the second the category for *rowdim2*.
- You can only use the `SetCellsByRow` method with pivot tables that have one column dimension.

Example

```
from spss import CellText

table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

coldim=table.Append(spss.Dimension.Place.column,"coldim")
rowdim1=table.Append(spss.Dimension.Place.row,"rowdim-1")
rowdim2=table.Append(spss.Dimension.Place.row,"rowdim-2")

cat1=CellText.String("rowdim1:A")
cat2=CellText.String("rowdim1:B")
cat3=CellText.String("rowdim2:A")
cat4=CellText.String("rowdim2:B")
cat5=CellText.String("C")
cat6=CellText.String("D")
```

```
table.SetCategories(rowdim1,[cat1,cat2])
table.SetCategories(rowdim2,[cat3,cat4])
table.SetCategories(coldim,[cat5,cat6])

table.SetCellsByRow((cat1,cat3),
                    [CellText.Number(11),
                     CellText.Number(12)])
table.SetCellsByRow((cat1,cat4),
                    [CellText.Number(21),
                     CellText.Number(22)])
table.SetCellsByRow((cat2,cat3),
                    [CellText.Number(31),
                     CellText.Number(32)])
table.SetCellsByRow((cat2,cat4),
                    [CellText.Number(41),
                     CellText.Number(42)])
```

- In this example, Number objects are used to specify numeric values for the cells. Values will be formatted using the table's default format. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer. You can change the default format using the SetDefaultFormatSpec method, or you can override the default by explicitly specifying the format, as in: `CellText.Number(22,spss.FormatSpec.GeneralStat)`. For more information, see Number Class on p. 450.

Figure A-9
Resulting table

		coldim	
rowdim-1	rowdim-2	C	D
rowdim1:A	rowdim2:A	11	12
	rowdim2:B	21	22
rowdim1:B	rowdim2:A	31	32
	rowdim2:B	41	42

Examples of using the `SetCellsByRow` method are most easily understood in the context of going through the steps to create a pivot table. For more information, see General Approach to Creating Pivot Tables on p. 429.

SetCellsByColumn Method

.SetCellsByColumn(collabels,cells). *Sets cell values for the column specified by a set of categories, one for each column dimension.* The argument *collabels* specifies the set of categories that defines the column—a single value, or a list or tuple. The argument *cells* is a tuple or list of cell values. Column categories and cell values must be specified as CellText objects (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`).

- For tables with multiple column dimensions, the order of categories in the *collabels* argument is the order in which their respective dimensions were added (appended or inserted) to the table. For example, given two column dimensions *coldim1* and *coldim2* added in the order *coldim1* and *coldim2*, the first element in *collabels* should be the category for *coldim1* and the second the category for *coldim2*.
- You can only use the `SetCellsByColumn` method with pivot tables that have one row dimension.

Example

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
rowdim=table.Append(spss.Dimension.Place.row,"rowdim")
coldim1=table.Append(spss.Dimension.Place.column,"coldim-1")
coldim2=table.Append(spss.Dimension.Place.column,"coldim-2")

cat1=CellText.String("coldim1:A")
cat2=CellText.String("coldim1:B")
cat3=CellText.String("coldim2:A")
cat4=CellText.String("coldim2:B")
cat5=CellText.String("C")
cat6=CellText.String("D")

table.SetCategories(coldim1,[cat1,cat2])
table.SetCategories(coldim2,[cat3,cat4])
table.SetCategories(rowdim,[cat5,cat6])

table.SetCellsByColumn((cat1,cat3),
                       [CellText.Number(11),
                        CellText.Number(21)])
table.SetCellsByColumn((cat1,cat4),
                       [CellText.Number(12),
                        CellText.Number(22)])
table.SetCellsByColumn((cat2,cat3),
                       [CellText.Number(13),
                        CellText.Number(23)])
table.SetCellsByColumn((cat2,cat4),
                       [CellText.Number(14),
                        CellText.Number(24)])
```

- In this example, Number objects are used to specify numeric values for the cells. Values will be formatted using the table's default format. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer. You can change the default format using the SetDefaultFormatSpec method, or you can override the default by explicitly specifying the format, as in: `CellText.Number(22,spss.FormatSpec.GeneralStat)`. For more information, see Number Class on p. 450.

Figure A-10
Resulting table structure

	coldim-1			
	coldim1:A		coldim1:B	
	coldim-2		coldim-2	
rowdim	coldim2:A	coldim2:B	coldim2:A	coldim2:B
C	11	12	13	14
D	21	22	23	24

Examples of using the `SetCellsByColumn` method are most easily understood in the context of going through the steps to create a pivot table. For more information, see General Approach to Creating Pivot Tables on p. 429.

SetDefaultFormatSpec

.SetDefaultFormatSpec(formatSpec,varIndex). *Sets the default format for CellText.Number objects.* The argument *formatspec* is of the form `spss.FormatSpec.format` where `format` is one of those listed in Table A-1 on p. 451—for example, `spss.FormatSpec.Mean`. The argument *varIndex* is the index of a variable in the active dataset whose format is used to determine details of the resulting format. *varIndex* is only used for, and required by, the following subset of formats: `Mean`, `Variable`, `StdDev`, `Difference`, and `Sum`. Index values represent position in the active dataset, starting with 0 for the first variable in file order. The default format can be retrieved with the GetDefaultFormatSpec method.

- Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer.

Example

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
table.SetDefaultFormatSpec(spss.FormatSpec.Mean,2)
table.Append(spss.Dimension.Place.row,"rowdim")
table.Append(spss.Dimension.Place.column,"coldim")

table[(CellText.String("row1"),CellText.String("col1"))] = \
       CellText.Number(2.37)
table[(CellText.String("row2"),CellText.String("col1"))] = \
       CellText.Number(4.34)
```

- The call to `SetDefaultFormatSpec` specifies that the format for mean values is to be used as the default, and that it will be based on the format for the variable with index value 2 in the active dataset. Subsequent instances of `CellText.Number` will use this default, so the cell values `2.37` and `4.34` will be formatted as mean values.

SimplePivotTable Method

.SimplePivotTable(rowdim,rowlabels,coldim,collabels,cells). *Creates a pivot table with one row dimension and one column dimension.*

- **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 rows. Each item can be a character string, a numeric value, or a CellText object (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`). If provided, the length of this list determines the number of rows in the table. If 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 columns. Each item can be a character string, a numeric value, or a CellText object (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`). If provided, the length of this list determines the number of columns in the table. If omitted, the number of columns is equal to the length of the first element of *cells*. If *cells* is one-dimensional, this implies a table with one column and as many rows

as there are elements in *cells*. See the examples below for the case where *cells* is two-dimensional and *collabels* is omitted.

- **cells.** This argument specifies the values for the cells of the pivot table. It consists of a one- or two-dimensional sequence of items that can be indexed as `cells[i]` or `cells[i][j]`. For example, `[1,2,3,4]` is a one-dimensional sequence, and `[[1,2],[3,4]]` is a two-dimensional sequence. Each element in *cells* can be a character string, a numeric value, or a CellText object (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`). Examples showing how the rows and columns of the pivot table are populated from *cells* are provided below.
- The number of elements in *cells* must equal the product of the number of rows and the number of columns.
- Elements in the pivot table are populated in row-wise fashion from the elements of *cells*. For example, if you specify a table with two rows and two columns and provide `cells=[1,2,3,4]`, the first row will consist of the first two elements and the second row will consist of the last two elements.
- Numeric values specified in *cells*, *rowlabels*, or *collabels* will be converted to CellText.Number objects with a format given by the default. The default format can be set with the SetDefaultFormatSpec method and retrieved with the GetDefaultFormatSpec method. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer.
- String values specified in *cells*, *rowlabels*, or *collabels* will be converted to CellText.String objects.

Example: Creating a Table with One Column

```
import spss
spss.StartProcedure("mycompany.com.demoProc")

table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
table.SimplePivotTable(rowdim="row dimension",
                       rowlabels=["row 1","row 2","row 3","row 4"],
                       collabels=["column 1"],
                       cells = [1,2,3,4])
spss.EndProcedure()
```

Result

Figure A-11
Pivot table with a single column

row dimension	column 1
row 1	1
row 2	2
row 3	3
row 4	4

Example: Using a Two-Dimensional Sequence for Cells

```
import spss
spss.StartProcedure("mycompany.com.demoProc")

table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
table.SimplePivotTable(rowdim="row dimension",
                       coldim="column dimension",
                       rowlabels=["row 1","row 2","row 3","row 4"],
                       collabels=["column 1","column 2"],
                       cells = [[1,2],[3,4],[5,6],[7,8]])
spss.EndProcedure()
```

Result

Figure A-12
Table populated from two-dimensional sequence

	column dimension	
row dimension	column 1	column 2
row 1	1	2
row 2	3	4
row 3	5	6
row 4	7	8

Example: Using a Two-Dimensional Sequence for Cells and Omitting Column Labels

```
import spss
spss.StartProcedure("mycompany.com.demoProc")

table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
table.SimplePivotTable(rowdim="row dimension",
                       coldim="column dimension",
                       rowlabels=["row 1","row 2","row 3","row 4"],
                       cells = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
spss.EndProcedure()
```

Result

Figure A-13
Table populated from two-dimensional sequence without specifying column labels

	column dimension		
row dimension	col0	col1	col2
row 1	1	2	3
row 2	4	5	6
row 3	7	8	9
row 4	10	11	12

Auxiliary Classes for Use with spss.BasePivotTable

spss.CellText Class

spss.CellText. *A class of objects used to create a dimension category or a cell in a pivot table.* This class is only for use with the `spss.BasePivotTable` class. The `CellText` class is used to create the following object types:

- CellText.Number: Used to specify a numeric value.
- CellText.String: Used to specify a string value.
- CellText.VarName: Used to specify a variable name. Use of this object means that settings for the display of variable names in pivot tables (names, labels, or both) are honored.
- CellText.VarValue: Used to specify a variable value. Use of this object means that settings for the display of variable values in pivot tables (values, labels, or both) are honored.

Number Class

spss.CellText.Number(value,formatspec,varIndex). *Used to specify a numeric value for a category or a cell in a pivot table.* The argument *value* specifies the numeric value. The optional argument *formatspec* is of the form `spss.FormatSpec.format` where `format` is one of those listed in the table below—for example, `spss.FormatSpec.Mean`. You can also specify an integer code for *formatspec*, as in the value `5` for `Mean`. The argument *varIndex* is the index of a variable in the active dataset whose format is used to determine details of the resulting format. *varIndex* is only used in conjunction with *formatspec* and is required when specifying one of the following formats: `Mean`, `Variable`, `StdDev`, `Difference`, and `Sum`. Index values represent position in the active dataset, starting with 0 for the first variable in file order.

- When *formatspec* is omitted, the default format is used. You can set the default format with the SetDefaultFormatSpec method and retrieve the default with the GetDefaultFormatSpec method. Instances of the `BasePivotTable` class have an implicit default format of `Count`.
- You can obtain a numeric representation of a `CellText.Number` object using the toNumber method, and you can use the toString method to obtain a string representation.

Example

```
from spss import CellText
from spss import FormatSpec
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

table.Append(spss.Dimension.Place.row,"rowdim")
table.Append(spss.Dimension.Place.column,"coldim")

table[(CellText.String("row1"),CellText.String("col1"))] = \
       CellText.Number(25.632,FormatSpec.Mean,2)
table[(CellText.String("row2"),CellText.String("col1"))] = \
       CellText.Number(23.785,FormatSpec.Mean,2)
```

In this example, cell values are displayed in the format used for mean values. The format of the variable with index 2 in the active dataset is used to determine the details of the resulting format.

Table A-1
Numeric formats for use with FormatSpec

Format name	Code
Coefficient	0
CoefficientSE	1
CoefficientVar	2
Correlation	3
GeneralStat	4
Mean	5
Count	6
Percent	7

Format name	Code
PercentNoSign	8
Proportion	9
Significance	10
Residual	11
Variable	12
StdDev	13
Difference	14
Sum	15

Suggestions for Choosing a Format

- Consider using `Coefficient` for unbounded, unstandardized statistics; for instance, beta coefficients in regression.
- `Correlation` is appropriate for statistics bounded by –1 and 1 (typically correlations or measures of association).
- Consider using `GeneralStat` for unbounded, scale-free statistics; for instance, beta coefficients in regression.
- `Mean` is appropriate for the mean of a single variable, or the mean across multiple variables.
- `Count` is appropriate for counts and other integers such as integer degrees of freedom.
- `Percent` and `PercentNoSign` are both appropriate for percentages. `PercentNoSign` results in a value without a percentage symbol (%).
- `Significance` is appropriate for statistics bounded by 0 and 1 (for example, significance levels).
- Consider using `Residual` for residuals from cell counts.
- `Variable` refers to a variable's print format as given in the data dictionary and is appropriate for statistics whose values are taken directly from the observed data (for instance, minimum, maximum, and mode).
- `StdDev` is appropriate for the standard deviation of a single variable, or the standard deviation across multiple variables.
- `Sum` is appropriate for sums of single variables. Results are displayed using the specified variable's print format.

String Class

spss.CellText.String(value). *Used to specify a string value for a category or a cell in a pivot table.* The argument is the string value.

- You can obtain a string representation of a `CellText.String` object using the toString method. For character representations of numeric values stored as `CellText.String` objects, such as `CellText.String("11")`, you can obtain the numeric value using the toNumber method.

Example

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

table.Append(spss.Dimension.Place.row,"rowdim")
table.Append(spss.Dimension.Place.column,"coldim")

table[(CellText.String("row1"),CellText.String("col1"))] = \
       CellText.String("1")
table[(CellText.String("row2"),CellText.String("col1"))] = \
       CellText.String("2")
```

VarName Class

spss.CellText.VarName(index). *Used to specify that a category or cell in a pivot table is to be treated as a variable name.* `CellText.VarName` objects honor display settings for variable names in pivot tables (names, labels, or both). The argument is the index value of the variable. Index values represent position in the active dataset, starting with 0 for the first variable in file order.

Example

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
coldim=table.Append(spss.Dimension.Place.column,"coldim")
rowdim=table.Append(spss.Dimension.Place.row,"rowdim")
table.SetCategories(rowdim,[CellText.VarName(0),CellText.VarName(1)])
table.SetCategories(coldim,CellText.String("Column Heading"))
```

In this example, row categories are specified as the names of the variables with index values 0 and 1 in the active dataset. Depending on the setting of pivot table labeling for variables in labels, the variable names, labels, or both will be displayed.

VarValue Class

spss.CellText.VarValue(index,value). *Used to specify that a category or cell in a pivot table is to be treated as a variable value.* `CellText.VarValue` objects honor display settings for variable values in pivot tables (values, labels, or both). The argument *index* is the index value of the variable. Index values represent position in the active dataset, starting with 0 for the first variable in file order. The argument *value* is a number (for a numeric variable) or string (for a string variable) representing the value of the `CellText` object.

Example

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
coldim=table.Append(spss.Dimension.Place.column,"coldim")
rowdim=table.Append(spss.Dimension.Place.row,"rowdim")
table.SetCategories(rowdim,[CellText.VarValue(0,1),CellText.VarValue(0,2)])
table.SetCategories(coldim,CellText.String("Column Heading"))
```

In this example, row categories are specified as the values 1 and 2 of the variable with index value 0 in the active dataset. Depending on the setting of pivot table labeling for variable values in labels, the values, value labels, or both will be displayed.

toNumber Method

This method is used to obtain a numeric representation of a `CellText.Number` object or a `CellText.String` object that stores a character representation of a numeric value, as in `CellText.String("123")`. Values obtained from this method can be used in arithmetic expressions. You call this method on a `CellText.Number` or `CellText.String` object.

Example

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
```

```
table.Append(spss.Dimension.Place.row,"row dimension")
table.Append(spss.Dimension.Place.column,"column dimension")

row_cat1 = CellText.String("first row")
row_cat2 = CellText.String("second row")
col_cat1 = CellText.String("first column")
col_cat2 = CellText.String("second column")

table[(row_cat1,col_cat1)] = CellText.Number(11)
cellValue = table[(row_cat1,col_cat1)].toNumber()
table[(row_cat2,col_cat2)] = CellText.Number(2*cellValue)
```

- `table[(row_cat1,col_cat1)].toNumber()` returns a numeric representation of the `CellText` object (recall that table cells are stored as `CellText` objects) for the cell with category values `("first row","first column")`.

toString Method

This method is used to obtain a string representation of a `CellText.String` or `CellText.Number` object. Values obtained from this method can be used in string expressions. You call this method on a `CellText.String` or `CellText.Number` object.

Example

```
from spss import CellText
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")

table.Append(spss.Dimension.Place.row,"row dimension")
table.Append(spss.Dimension.Place.column,"column dimension")

row_cat1 = CellText.String("first row")
row_cat2 = CellText.String("second row")
col_cat1 = CellText.String("first column")
col_cat2 = CellText.String("second column")

table[(row_cat1,col_cat1)] = CellText.String("abc")
cellValue = table[(row_cat1,col_cat1)].toString()
table[(row_cat2,col_cat2)] = CellText.String(cellValue + "d")
```

- `table[(row_cat1,col_cat1)].toString()` returns a string representation of the `CellText` object (recall that table cells are stored as `CellText` objects) for the cell with category values `("first row","first column")`.

spss.BaseProcedure Class

The `spss.BaseProcedure` class is used to create classes that encapsulate procedures. Procedures can read the data, perform computations, add new variables and/or new cases to the active dataset, and produce pivot table output and text blocks in the SPSS Viewer. Procedures have almost the same capabilities as built-in SPSS procedures, such as `DESCRIPTIVES` and `REGRESSION`, but they are written in Python by users. Use of the `spss.BaseProcedure` class provides an alternative to encapsulating procedure code within a Python function and explicitly using an `spss.StartProcedure-spss.EndProcedure` block for the procedure output. All classes that encapsulate procedures must inherit from the `BaseProcedure` class.

The `spss.BaseProcedure` class has three methods: `__init__`, `execProcedure`, and `execUserProcedure`. When creating procedure classes you always override the `execUserProcedure` method, replacing it with the body of your procedure. You override the `__init__` method if you need to provide arguments other than the procedure name. You never override the `execProcedure` method. It is responsible for calling `execUserProcedure` to run your procedure as well as automatically making the necessary calls to `spss.StartProcedure` and `spss.EndProcedure`.

The rules governing procedure code contained within the `execUserProcedure` method are the same as those for `StartProcedure-EndProcedure` blocks. For more information, see spss.StartProcedure Function on p. 506.

Example

As an example, we will create a procedure class that calculates group means for a selected variable using a specified categorical variable to define the groups. The output of the procedure is a pivot table displaying the group means. For an alternative approach to creating the same procedure, but making explicit use of `spss.StartProcedure-spss.EndProcedure` and without the use of the `BaseProcedure` class, see the example for the spss.StartProcedure function.

```
class groupMeans(spss.BaseProcedure):

    #Overrides __init__ method to pass arguments
    def __init__(self, procName, groupVar, sumVar):
        self.procName = procName
        self.groupVar = groupVar
        self.sumVar = sumVar

    #Overrides execUserProcedure method of BaseProcedure
    def execUserProcedure(self):
        #Determine variable indexes from variable names
        varCount = spss.GetVariableCount()
        groupIndex = 0
        sumIndex = 0
        for i in range(varCount):
            varName = spss.GetVariableName(i)
            if varName == self.groupVar:
                groupIndex = i
                continue
            elif varName == self.sumVar:
                sumIndex = i
                continue

        varIndex = [groupIndex,sumIndex]
        cur = spss.Cursor(varIndex)
        Counts={};Salaries={}

        #Calculate group sums
        for i in range(cur.GetCaseCount()):
            row = cur.fetchone()
            cat=int(row[0])
            Counts[cat]=Counts.get(cat,0) + 1
            Salaries[cat]=Salaries.get(cat,0) + row[1]

        cur.close()

        #Create a pivot table
        table = spss.BasePivotTable("Group Means",
                                    "OMS table subtype")
        table.Append(spss.Dimension.Place.row,
                     spss.GetVariableLabel(groupIndex))
        table.Append(spss.Dimension.Place.column,
                     spss.GetVariableLabel(sumIndex))

        category2 = spss.CellText.String("Mean")
        for cat in sorted(Counts):
            category1 = spss.CellText.Number(cat)
            table[(category1,category2)] = \
                  spss.CellText.Number(Salaries[cat]/Counts[cat])
```

- `groupMeans` is a class based on the `spss.BaseProcedure` class.

- The procedure defined by the class requires two arguments, the name of the grouping variable (*groupVar*) and the name of the variable for which group means are desired (*sumVar*). Passing these values requires overriding the `__init__` method of `spss.BaseProcedure`. The values of the parameters are stored to the properties *groupVar* and *sumVar* of the class instance. The value passed in as the procedure name is stored to the *procName* property.
- The body of the procedure is contained within the `execUserProcedure` method, which overrides that method in `spss.BaseProcedure`. The procedure reads the data to calculate group sums and group case counts and creates a pivot table populated with the group means.
- The necessary calls to `spss.StartProcedure` and `spss.EndProcedure` are handled by `spss.BaseProcedure`.

Saving and Running Procedure Classes

Once you have written a procedure class, you will want to save it in a Python module on the Python search path so that you can call it. 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 classes in a single module. To be sure that Python can find your new module, you may want to save it to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*.

For the example procedure class described above, you might choose to save the class definition to a Python module named *myprocs.py*. And be sure to include an `import spss` statement in the module. Sample command syntax to instantiate this class and run the procedure is:

```
import spss, myprocs
spss.Submit("GET FILE='c:/program files/spss15/Employee data.sav'.")
proc = myprocs.groupMeans("mycompany.com.groupMeans","educ","salary")
proc.execProcedure()
```

- The import statement containing `myprocs` makes the contents of the Python module *myprocs.py* available to the current session (assuming that the module is on the Python search path).

- The call to `myprocs.groupMeans` creates an instance of the `groupMeans` class. The variables *educ* and *salary* in *c:/program files/spss15/Employee data.sav* are used as the grouping variable and the variable for which means are calculated.
- Output from the procedure is associated with the name *mycompany.com.groupMeans*. This is the name that appears in the outline pane of the Viewer associated with output produced by the procedure. It is also the command name associated with this procedure when routing output from this procedure with OMS (Output Management System), as well as the name associated with this procedure for use with autoscripts. In order that names associated with procedure output not conflict with names of existing SPSS commands (when working with OMS or autoscripts), SPSS recommends that they have the form *yourcompanyname.com.procedurename*. For more information, see spss.StartProcedure Function on p. 506.

Result

Figure A-14
Output from the groupMeans procedure

	Current Salary
Educational Level (years)	Mean
8	24399
12	25887
14	31625
15	31685
16	48226
17	59527
18	65128
19	72520
20	64313
21	65000

spss.CreateXPathDictionary Function

spss.CreateXPathDictionary(handle). *Creates an XPath dictionary DOM for the active dataset that can be accessed with XPath expressions.* The argument is a handle name, used to identify this DOM in subsequent `spss.EvaluateXPath` and `spss.DeleteXPathHandle` functions. It cannot be the name of an existing handle.

Example

```
handle='demo'
spss.CreateXPathDictionary(handle)
```

- The XPath dictionary DOM for the current active dataset is assigned the handle name *demo*. Any subsequent `spss.EvaluateXPath` or `spss.DeleteXPathHandle` functions that reference this dictionary DOM must use this handle name.

spss.Cursor Class

spss.Cursor(n, accessType). *Provides the ability to read cases, append cases, and add new variables to the active dataset.*

- The optional argument *accessType* specifies one of three usage modes: read ('r'), write ('w'), and append ('a'). The default is read mode.
- The optional argument *n* specifies a tuple or a list of variable index values representing position in the active dataset, starting with 0 for the first variable in file order. This argument is used in read or write mode to specify a subset of variables to return when reading case data from the active dataset. If the argument is omitted, all variables are returned. The argument has no effect if used in append mode.
- You cannot use the `spss.Submit` function while a data cursor is open. You must close or delete the cursor first.
- Only one data cursor can be open at any point in the program block. To define a new data cursor, you must first close or delete the previous one.
- Instances of the `Cursor` class are implicitly deleted at the end of a `BEGIN PROGRAM-END PROGRAM` block, and therefore they do not persist across `BEGIN PROGRAM-END PROGRAM` blocks.

Read Mode (accessType='r')

This is the default for the `Cursor` class and provides the ability to read case data from the active dataset. For users of a 14.0.x version of the plug-in who are upgrading to the 15.0 version, this mode is equivalent to `spss.Cursor(n)` in 14.0.x versions. No changes to your 14.0.x code for the `Cursor` class are required to run the code with the 15.0 version.

The `Cursor` methods fetchone, fetchmany, and fetchall are used to retrieve cases from the active dataset.

Example

```
*python_cursor.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()
oneRow=dataCursor.fetchone()
dataCursor.close()
i=[0]
dataCursor=spss.Cursor(i)
oneVar=dataCursor.fetchall()
dataCursor.close()
print "One row (case): ", oneRow
print "One column (variable): ", oneVar
END PROGRAM.
```

Result

```
One row (case):  (11.0, 'ab', 13.0)
One column (variable):  ((11.0,), (21.0,), (31.0,))
```

- Cases from the active dataset are returned as a single tuple for `fetchone` and a list of tuples for `fetchall`.
- Each tuple represents the data for one case. For `fetchall` the tuples are arranged in the same order as the cases in the active dataset.
- Each element in a tuple contains the data value for a specific variable. The order of variable values within a tuple is the order specified by the optional argument *n* to the `Cursor` class, or file order if *n* is omittted.

Example

```
*python_cursor_sysmis.sps.
*System- and user-missing values.
DATA LIST LIST (',') /numVar (f) stringVar (a4).
BEGIN DATA
1,a
,b
```

```
3,,
4,d
END DATA.
MISSING VALUES stringVar (' ').
BEGIN PROGRAM.
import spss
dataCursor=spss.Cursor()
print dataCursor.fetchall()
dataCursor.close()
END PROGRAM.
```

Result

```
((1.0, 'a   '), (None, 'b   '), (3.0, None), (4.0, 'd   '))
```

- String values are right-padded to the defined width of the string variable.
- System-missing values are always converted to the Python data type *None*.
- By default, user-missing values are converted to the Python data type *None*. You can use the SetUserMissingInclude method to specify that user-missing values be treated as valid.

Write Mode (accessType='w')

This mode is used to add new variables, along with their case values, to an existing dataset. It cannot be used to append cases to the active dataset.

- All of the methods available in read mode are also available in write mode.
- When adding new variables, the `CommitDictionary` method must be called after the statements defining the new variables and prior to setting case values for those variables. You cannot add new variables to an empty dataset.
- When setting case values for new variables, the `CommitCase` method must be called for each case that is modified. The `fetchone` method is used to advance the record pointer by one case, or you can use the `fetchmany` method to advance the record pointer by a specified number of cases.
- Changes to the active dataset do not take effect until the cursor is closed.
- Write mode supports multiple data passes and allows you to add new variables on each pass. In the case of multiple data passes where you need to add variables on a data pass other than the first, you must call the `AllocNewVarsBuffer` method to allocate the buffer size for the new variables. When used, `AllocNewVarsBuffer`

must be called before reading any data with `fetchone`, `fetchmany`, or `fetchall`.

- The `Cursor` methods SetVarNameAndType and SetOneVarNameAndType are used to add new variables to the active dataset, and the methods SetValueChar and SetValueNumeric are used to set case values.

Example

In this example, we create a new numeric variable and a new string variable and set their values for all cases.

```
*python_cursor_create_var.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
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['var4','strvar'],[0,8])
cur.SetVarFormat('var4',5,2,0)
cur.CommitDictionary()
for i in range(cur.GetCaseCount()):
   cur.fetchone()
   cur.SetValueNumeric('var4',4+10*(i+1))
   cur.SetValueChar('strvar','row' + str(i+1))
   cur.CommitCase()
cur.close()
END PROGRAM.
```

- An instance of the `Cursor` class in write mode is created and assigned to the variable *cur*.
- The SetVarNameAndType method is used to add two new variables to the active dataset. *var4* is a numeric variable and *strvar* is a string variable of width 8.
- SetVarFormat sets the display format for *var4*. The integers 5, 2, and 0 specify the format type (5 is standard numeric), the defined width, and the number of decimal digits respectively.
- The `CommitDictionary` method is called to commit the new variables to the cursor before populating their case values.

- The SetValueNumeric and SetValueChar methods are used to set the case values of the new variables. The CommitCase method is called to commit the changes for each modified case.
- fetchone advances the record pointer to the next case.

Example

In this example, we create new variables and set their values for specific cases. The fetchone method is used to advance the record pointer to the desired cases.

```
*python_cursor_move_pointer.sps.
DATA LIST FREE /code (A1) loc (A3) emp (F) dtop (F) ltop (F).
BEGIN DATA
H NY 151 127 24
W CHI 17 4 0
S CHI 9 3 6
W ATL 12 3 0
W SDG 13 4 0
S ATL 10 3 7
S SDG 11 3 8
END DATA.
BEGIN PROGRAM.
import spss
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['emp_est','dtop_est','ltop_est'],[0,0,0])
cur.SetVarFormat('emp_est',5,2,0)
cur.SetVarFormat('dtop_est',5,2,0)
cur.SetVarFormat('ltop_est',5,2,0)
cur.CommitDictionary()
for i in range(cur.GetCaseCount()):
   row=cur.fetchone()
   if (row[0].lower()=='s'):
      cur.SetValueNumeric('emp_est',1.2*row[2])
      cur.SetValueNumeric('dtop_est',1.2*row[3])
      cur.SetValueNumeric('ltop_est',1.2*row[4])
      cur.CommitCase()
cur.close()
END PROGRAM.
```

Example

In this example, we read the data, calculate a summary statistic, and use a second data pass to add a summary variable to the active dataset.

```
*python_cursor_multipass.sps.
DATA LIST FREE /var (F).
BEGIN DATA
57000
40200
21450
21900
END DATA.
BEGIN PROGRAM.
import spss
cur=spss.Cursor(accessType='w')
cur.AllocNewVarsBuffer(8)
total=0
for i in range(spss.GetCaseCount()):
   total+=cur.fetchone()[0]
meanVal=total/spss.GetCaseCount()
cur.reset()
cur.SetOneVarNameAndType('mean',0)
cur.CommitDictionary()
for i in range(spss.GetCaseCount()):
   row=cur.fetchone()
   cur.SetValueNumeric('mean',meanVal)
   cur.CommitCase()
cur.close()
END PROGRAM.
```

- Because we will be adding a new variable on the second data pass, the `AllocNewVarsBuffer` method is called to allocate the required space. In the current example, we are creating a single numeric variable, which requires eight bytes.
- The first `for` loop is used to read the data and total the case values.
- After the data pass, the `reset` method must be called prior to defining new variables.
- The second data pass (second `for` loop) is used to add the mean value of the data as a new variable.

Append Mode (accessType='a')

This mode is used to append new cases to the active dataset. It cannot be used to add new variables or read case data from the active dataset. A dataset must contain at least one variable in order to append cases to it, but it need not contain any cases.

- The `CommitCase` method must be called for each case that is added.
- The `EndChanges` method must be called before the cursor is closed.

- Changes to the active dataset do not take effect until the cursor is closed.
- A numeric variable whose value is not specified (for a new case) is set to the system-missing value.
- A string variable whose value is not specified (for a new case) will have a blank value. The value will be valid unless you explicitly define the blank value to be missing for that variable.
- The `Cursor` methods SetValueChar and SetValueNumeric are used to set variable values for new cases.

Example

```
*python_cursor_append_cases.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
cur=spss.Cursor(accessType='a')
ncases=cur.GetCaseCount()
newcases=2
for i in range(newcases):
   cur.SetValueNumeric('var1',1+10*(ncases+i+1))
   cur.SetValueNumeric('var3',3+10*(ncases+i+1))
   cur.CommitCase()
cur.EndChanges()
cur.close()
END PROGRAM.
```

- An instance of the `Cursor` class in append mode is created and assigned to the variable *cur*.
- The `SetValueNumeric` method is used to set the case values of *var1* and *var3* for two new cases. No value is specified for *var2*. The `CommitCase` method is called to commit the values for each case.
- The `EndChanges` method is called to commit the new cases to the cursor.

spss.Cursor Methods

Each usage mode of the `Cursor` class supports its own set of methods, as shown in the table below. Descriptions of each method follow.

Method	**Read mode**	**Write mode**	**Append mode**
AllocNewVarsBuffer		X	
close	X	X	X
CommitCase		X	X
CommitDictionary		X	
EndChanges			X
fetchall	X	X**	
fetchmany	X	X**	
fetchone	X	X	
GetCaseCount	X	X	X
GetVarAttributeNames	X	X	X
GetVarAttributes	X	X	X
GetVariableCount	X	X	X
GetVariableFormat	X	X	X
GetVariableLabel	X	X	X
GetVariableMeasurementLevel	X	X	X
GetVariableName	X	X	X
GetVariableType	X	X	X
GetVarMissingValues	X	X	X
IsEndSplit	X	X	
reset	X	X	X
SetFetchVarList	X	X	
SetOneVarNameAndType		X	
SetUserMissingInclude	X	X	
SetValueChar		X	X
SetValueNumeric		X	X
SetVarAlignment		X	
SetVarAttributes		X	

Method	Read mode	Write mode	Append mode
SetVarCMissingValues		X	
SetVarCValueLabel		X	
SetVarFormat		X	
SetVarLabel		X	
SetVarMeasureLevel		X	
SetVarNameAndType		X	
SetVarNMissingValues		X	
SetVarNValueLabel		X	

** This method is primarily for use in read mode.

Note

The `Cursor` class `Get` methods (for instance, `GetCaseCount`, `GetVariableCount`, and so on) listed above have the same specifications as the functions in the `spss` module of the same name. For example, the specifications for the `Cursor` method `GetCaseCount` are the same as those for the `spss.GetCaseCount` function. While a cursor is open, both sets of functions return information about the current cursor and give identical results. In the absence of a cursor, the `spss` module functions retrieve information about the active dataset. Refer to the entries for the corresponding `spss` module functions for specifications of these `Cursor` methods.

AllocNewVarsBuffer Method

.AllocNewVarsBuffer(bufSize). *Specifies the buffer size, in bytes, to use when adding new variables in the context of multiple data passes.* The argument *bufSize* is a positive integer large enough to accommodate all new variables to be created by a given write cursor. Each numeric variable requires eight bytes. For each string variable, you should allocate 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.

- This method is only available in write mode.

- `AllocNewVarsBuffer` is required in the case of multiple data passes when you need to add variables on a data pass other than the first. When used, it must be called before reading any data with `fetchone`, `fetchmany`, or `fetchall`.
- `AllocNewVarsBuffer` can only be called once for a given write cursor instance.
- Specifying a larger buffer size than is required has no effect on the result.

Example

In this example, two data passes are executed. The first data pass is used to read the data and compute a summary statistic. The second data pass is used to add a summary variable to the active dataset.

```
*python_cursor_multipass.sps.
DATA LIST FREE /var (F).
BEGIN DATA
57000
40200
21450
21900
END DATA.
BEGIN PROGRAM.
import spss
cur=spss.Cursor(accessType='w')
cur.AllocNewVarsBuffer(8)
total=0
for i in range(spss.GetCaseCount()):
   total+=cur.fetchone()[0]
meanVal=total/spss.GetCaseCount()
cur.reset()
cur.SetOneVarNameAndType('mean',0)
cur.CommitDictionary()
for i in range(spss.GetCaseCount()):
   row=cur.fetchone()
   cur.SetValueNumeric('mean',meanVal)
   cur.CommitCase()
cur.close()
END PROGRAM.
```

close Method

.close(). *Closes the cursor.* You cannot use the `spss.Submit` function while a data cursor is open. You must close or delete the cursor first.

- This method is available in read, write, or append mode.

- When appending cases, you must call the EndChanges method before the `close` method.
- Cursors are implicitly closed at the end of a `BEGIN PROGRAM-END PROGRAM` block.

Example

```
cur=spss.Cursor()
data=cur.fetchall()
cur.close()
```

CommitCase Method

.CommitCase(). *Commits changes to the current case in the current cursor.* This method must be called for each case that is modified, including existing cases modified in write mode and new cases created in append mode.

- This method is available in write or append mode.
- When working in write mode, you advance the record pointer by calling the `fetchone` method. To modify the first case, you must first call `fetchone`.
- When working in append mode, the cursor is ready to accept values for a new record (using `SetValueNumeric` and `SetValueChar`) once `CommitCase` has been called for the previous record.
- Changes to the active dataset take effect when the cursor is closed.

For an example of using `CommitCase` in write mode, see the topic on write mode on p. 462. For an example of using `CommitCase` in append mode, see the topic on append mode on p. 465.

CommitDictionary Method

.CommitDictionary(). *Commits new variables to the current cursor.*

- This method is only available in write mode.
- When adding new variables, you must call this method before setting case values for the new variables.
- Changes to the active dataset take effect when the cursor is closed.

Example

```
*python_cursor_CommitDictionary.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
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarLabel('numvar','New numeric variable')
cur.SetVarFormat('numvar',5,2,0)
cur.CommitDictionary()
for i in range(cur.GetCaseCount()):
   cur.fetchone()
   cur.SetValueNumeric('numvar',4+10*(i+1))
   cur.CommitCase()
cur.close()
END PROGRAM.
```

EndChanges Method

.EndChanges(). *Specifies the end of appending new cases.* This method must be called before the cursor is closed.

- This method can only be called once for a given `Cursor` instance and is only available in append mode.
- Changes to the active dataset take effect when the cursor is closed.

For an example of using `EndChanges`, see the topic on append mode on p. 465.

fetchall Method

.fetchall(). *Fetches all (remaining) cases from the active dataset, or if there are splits, the remaining cases in the current split.* If there are no remaining rows, the result is an empty tuple.

- This method is available in read or write mode.
- When used in write mode, calling `fetchall` will position the record pointer at the last case of the active dataset, or if there are splits, the last case of the current split.

- Cases from the active dataset are returned as a list of tuples. Each tuple represents the data for one case, and the tuples are arranged in the same order as the cases in the active dataset. Each element in a tuple contains the data value for a specific variable. The order of variable values within a tuple is the order specified by the variable index values in the optional argument *n* to the `Cursor` class, or file order if *n* is omitted. For example, if *n*=[5,2,7] the first tuple element is the value of the variable with index value 5, the second is the variable with index value 2, and the third is the variable with index value 7.
- String values are right-padded to the defined width of the string variable.
- System-missing values are always converted to the Python data type *None*.
- By default, user-missing values are converted to the Python data type *None*. You can use the SetUserMissingInclude method to specify that user-missing values be treated as valid.

Examples

```
*python_cursor_fetchall.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()
dataFile=dataCursor.fetchall()
for i in enumerate(dataFile):
  print i
print dataCursor.fetchall()
dataCursor.close()
END PROGRAM.
```

Result

```
(0, (11.0, 'ab', 13.0))
(1, (21.0, 'cd', 23.0))
(2, (31.0, 'ef', 33.0))
()
```

fetchall with Variable Index

```
*python_cursor_fetchall_index.sps.
DATA LIST FREE /var1 var2 var3.
```

```
BEGIN DATA
1 2 3
1 4 5
2 5 7
END DATA.
BEGIN PROGRAM.
import spss
i=[0]
dataCursor=spss.Cursor(i)
oneVar=dataCursor.fetchall()
uniqueCount=len(set(oneVar))
print oneVar
print spss.GetVariableName(0), " has ", uniqueCount, " unique values."
dataCursor.close()
END PROGRAM.
```

Result

```
((1.0,), (1.0,), (2.0,))
var1  has  2  unique values.
```

fetchmany Method

.fetchmany(n). *Fetches the next n cases from the active dataset, where n is a positive integer.* If the value of *n* is greater than the number of remaining cases (and the dataset does not contain splits), it returns the value of all the remaining cases. In the case that the active dataset has splits, if *n* is greater than the number of remaining cases in the current split, it returns the value of the remaining cases in the split. If there are no remaining cases, the result is an empty tuple.

- This method is available in read or write mode.
- When used in write mode, calling `fetchmany(n)` will position the record pointer at case `n` of the active dataset. In the case that the dataset has splits and `n` is greater than the number of remaining cases in the current split, `fetchmany(n)` will position the record pointer at the end of the current split.
- Cases from the active dataset are returned as a list of tuples. Each tuple represents the data for one case, and the tuples are arranged in the same order as the cases in the active dataset. Each element in a tuple contains the data value for a specific variable. The order of variable values within a tuple is the order specified by the variable index values in the optional argument *n* to the `Cursor` class, or file order if *n* is omitted. For example, if *n*=[5,2,7] the first tuple element is the value of the variable with index value 5, the second is the variable with index value 2, and the third is the variable with index value 7.

- String values are right-padded to the defined width of the string variable.
- System-missing values are always converted to the Python data type *None*.
- By default, user-missing values are converted to the Python data type *None*. You can use the SetUserMissingInclude method to specify that user-missing values be treated as valid.

Example

```
*python_cursor_fetchmany.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()
n=2
print dataCursor.fetchmany(n)
print dataCursor.fetchmany(n)
print dataCursor.fetchmany(n)
dataCursor.close()
END PROGRAM.
```

Result

```
((11.0, 'ab', 13.0), (21.0, 'cd', 23.0))
((31.0, 'ef', 33.0),)
()
```

fetchone Method

.fetchone(). *Fetches the next row (case) from the active dataset.* The result is a single tuple or the Python data type *None* after the last row has been read. A value of *None* is also returned at a split boundary. In this case, a subsequent call to `fetchone` will retrieve the first case of the next split group.

- This method is available in read or write mode.
- Each element in the returned tuple contains the data value for a specific variable. The order of variable values in the tuple is the order specified by the variable index values in the optional argument *n* to the `Cursor` class, or file order if *n* is omitted. For example, if *n*=[5,2,7] the first tuple element is the value of the variable with

index value 5, the second is the variable with index value 2, and the third is the variable with index value 7.

- String values are right-padded to the defined width of the string variable.
- System-missing values are always converted to the Python data type *None*.
- By default, user-missing values are converted to the Python data type *None*. You can use the SetUserMissingInclude method to specify that user-missing values be treated as valid.

Example

```
*python_cursor_fetchone.sps.
DATA LIST FREE /var1 var2 var3.
BEGIN DATA
1 2 3
4 5 6
END DATA.
BEGIN PROGRAM.
import spss
dataCursor=spss.Cursor()
firstRow=dataCursor.fetchone()
secondRow=dataCursor.fetchone()
thirdRow=dataCursor.fetchone()
print "First row: ",firstRow
print "Second row ",secondRow
print "Third row...there is NO third row: ",thirdRow
dataCursor.close()
END PROGRAM.
```

Result

```
First row:  (1.0, 2.0, 3.0)
Second row  (4.0, 5.0, 6.0)
Third row...there is NO third row:  None
```

IsEndSplit Method

.IsEndSplit(). *Indicates if the cursor position has crossed a split boundary.* The result is Boolean—*true* if a split boundary has been crossed, otherwise *false*. This method is used in conjunction with the SplitChange function when creating custom pivot tables from data with splits.

- This method is available in read or write mode.

- The value returned from the `fetchone` method is *None* at a split boundary. Once a split has been detected, you will need to call `fetchone` again to retrieve the first case of the next split group.
- `IsEndSplit` returns *true* when the end of the dataset has been reached. 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`, as shown in the following example.

Example

```
*python_cursor_IsEndSplit.sps.
GET FILE='c:/program files/spss/employee data.sav'.
SORT CASES BY GENDER.
SPLIT FILE LAYERED BY GENDER.

BEGIN PROGRAM.
import spss
i=0
cur=spss.Cursor()
while True:
   cur.fetchone()
   i+=1
   if cur.IsEndSplit():
      # Try to fetch the first case of the next split group
      if not None==cur.fetchone():
         print "Found split end. New split begins at case: ", i
      else:
         #There are no more cases, so quit
         break
cur.close()
END PROGRAM.
```

reset Method

.reset(). *Resets the cursor.*

- This method is available in read, write, or append mode.
- In read and write modes, `reset` moves the record pointer to the first case, allowing multiple data passes. In append mode, it deletes the current cursor instance and creates a new one.
- When executing multiple data passes, the `reset` method must be called prior to defining new variables on subsequent passes. For an example, see the topic on write mode on p. 462.

Example

```
import spss
cur=spss.Cursor()
data=cur.fetchall()
cur.reset()
data10=cur.fetchmany(10)
cur.close()
```

SetFetchVarList

.SetFetchVarList(var). *Resets the list of variables to return when reading case data from the active dataset.* The argument *var* is a list or tuple of variable index values representing position in the active dataset, starting with 0 for the first variable in file order.

- This method is available in read or write mode.

Example

```
*python_cursor_reset_varlist.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
cur=spss.Cursor()
oneRow=cur.fetchone()
cur.SetFetchVarList([0])
cur.reset()
oneVar=cur.fetchall()
cur.close()
print "One row (case): ", oneRow
print "One column (variable): ", oneVar
END PROGRAM.
```

SetOneVarNameAndType Method

.SetOneVarNameAndType(varName,varType). *Creates one new variable in the active dataset.* The argument *varName* is a string that specifies the name of the new variable. The argument *varType* is an integer specifying the variable type of the new variable. You can create multiple variables with a single call using the SetVarNameAndType method.

- This method is only available in write mode.
- 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 (maximum of 32767).
- Use of the `SetOneVarNameAndType` method requires the `AllocNewVarsBuffer` method to allocate space for the variable.

Example

```
*python_cursor_create_onevar.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
cur=spss.Cursor(accessType='w')
cur.AllocNewVarsBuffer(8)
cur.SetOneVarNameAndType('var4',0)
cur.SetVarFormat('var4',5,2,0)
cur.CommitDictionary()
for i in range(cur.GetCaseCount()):
   cur.fetchone()
   cur.SetValueNumeric('var4',4+10*(i+1))
   cur.CommitCase()
cur.close()
END PROGRAM.
```

SetUserMissingInclude Method

.SetUserMissingInclude(incMissing). *Specifies the treatment of user-missing values read from the active dataset.* The argument is a Boolean with *true* specifying that user-missing values be treated as valid. A value of *false* specifies that user-missing values should be converted to the Python data type *None*.

- By default, user-missing values are converted to the Python data type *None*.
- System-missing values are always converted to *None*.
- This method is available in read or write mode.

Example

In this example, we will use the following data to demonstrate both the default behavior and the behavior when user missing values are treated as valid.

```
DATA LIST LIST (',') /numVar (f) stringVar (a4).
BEGIN DATA
1,a
,b
3,,
0,d
END DATA.
MISSING VALUES stringVar (' ') numVar(0).
```

This first `BEGIN PROGRAM` block demonstrates the default behavior.

```
BEGIN PROGRAM.
import spss
cur=spss.Cursor()
print cur.fetchall()
cur.close()
END PROGRAM.
```

Result

```
((1.0, 'a   '), (None, 'b   '), (3.0, None), (None, 'd   '))
```

This second `BEGIN PROGRAM` block demonstrates the behavior when user-missing values are treated as valid.

```
BEGIN PROGRAM.
import spss
cur=spss.Cursor()
cur.SetUserMissingInclude(True)
print cur.fetchall()
cur.close()
END PROGRAM.
```

Result

```
((1.0, 'a   '), (None, 'b   '), (3.0, '    '), (0.0, 'd   '))
```

SetValueChar Method

.SetValueChar(varName,varValue). *Sets the value for the current case for a string variable.* The argument *varName* is a string specifying the name of a string variable. The argument *varValue* is a string specifying the value of this variable for the current case.

- This method is available in write or append mode.
- The `CommitCase` method must called for each case that is modified. This includes new cases created in append mode.

Example

```
*python_cursor_SetValueChar.sps.
DATA LIST FREE /var1 (F) var2(F).
BEGIN DATA
11 12
21 22
31 32
END DATA.
BEGIN PROGRAM.
import spss
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['strvar'],[8])
cur.CommitDictionary()
for i in range(cur.GetCaseCount()):
   cur.fetchone()
   cur.SetValueChar('strvar','row' + str(i+1))
   cur.CommitCase()
cur.close()
END PROGRAM.
```

SetValueNumeric Method

.SetValueNumeric(varName,varValue). *Sets the value for the current case for a numeric variable.* The argument *varName* is a string specifying the name of a numeric variable. The argument *varValue* specifies the numeric value of this variable for the current case.

- This method is available in write or append mode.
- The `CommitCase` method must be called for each case that is modified. This includes new cases created in append mode.

- The Python data type *None* specifies a missing value for a numeric variable.
- Values of numeric variables with a date or datetime format should be specified as the number of seconds from October 14, 1582. You can convert from a four-digit year, month, and day to the associated number of seconds using the `yrmodasec` function from the `spssdata` module, a supplementary Python module available from *http://www.spss.com/devcentral*.

Example

```
*python_cursor_SetValueNumeric.sps.
DATA LIST FREE /var1 (F) var2 (F).
BEGIN DATA
11 12
21 22
31 32
END DATA.
BEGIN PROGRAM.
import spss
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['var3'],[0])
cur.SetVarFormat('var3',5,2,0)
cur.CommitDictionary()
for i in range(cur.GetCaseCount()):
   cur.fetchone()
   cur.SetValueNumeric('var3',3+10*(i+1))
   cur.CommitCase()
cur.close()
END PROGRAM.
```

SetVarAlignment Method

.SetVarAlignment(varName,alignment). *Sets the alignment of data values in the Data Editor for a new variable.* It has no effect on the format of the variables or the display of the variables or values in other windows or printed results. The argument *varName* is a string specifying the name of a new variable. The argument *alignment* is an integer and can take on one of the following values: 0 (left), 1 (right), 2 (center).

- This method is only available in write mode.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarAlignment('numvar',0)
cur.CommitDictionary()
```

```
cur.close()
```

SetVarAttributes Method

.SetVarAttributes(varName,attrName,attrValue,index). *Sets a value in an attribute array for a new variable.* The argument *varName* is a string specifying the name of a new variable. The argument *attrName* is a string specifying the name of the attribute array. The argument *attrValue* is a string specifying the attribute value, and *index* is the index position in the array, starting with the index 0 for the first element in the array.

- This method is only available in write mode.
- An attribute array with one element is equivalent to an attribute that is not specified as an array.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarAttributes('numvar','myattribute','first element',0)
cur.SetVarAttributes('numvar','myattribute','second element',1)
cur.CommitDictionary()
cur.close()
```

SetVarCMissingValues Method

.SetVarCMissingValues(varName,missingVal1,missingVal2,missingVal3). *Sets user-missing values for a new string variable.* The argument *varName* is a string specifying the name of a new string variable. The optional arguments *missingVal1*, *missingVal2*, and *missingVal3* are strings, each of which can specify one user-missing value. Use the SetVarNMissingValues method to set missing values for new numeric variables.

- This method is only available in write mode.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['strvar'],[8])
cur.SetVarCMissingValues('strvar',' ','NA')
cur.CommitDictionary()
cur.close()
```

SetVarCValueLabel Method

.SetVarCValueLabel(varName,value,label). *Sets the value label of a single value for a new string variable.* The argument *varName* is a string specifying the name of a new string variable. The arguments *value* and *label* are strings specifying the value and the associated label. Use the SetVarNValueLabel method to set value labels for new numeric variables.

- This method is only available in write mode.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['strvar'],[8])
cur.SetVarCValueLabel('strvar','f','female')
cur.CommitDictionary()
cur.close()
```

SetVarFormat Method

.SetVarFormat(varName,type,width,decimals). *Sets the display format for a new variable.* The argument *varName* is a string specifying the name of a new variable. The argument *type* is an integer that specifies one of the format types listed in the table below. The argument *width* is an integer specifying the defined width, which must include enough positions to accommodate any punctuation characters such as decimal points, commas, dollar signs, or date and time delimiters. The optional argument *decimals* is an integer specifying the number of decimal digits for numeric formats.

Allowable settings for decimal and width depend on the specified type. For a list of the minimum and maximum widths and maximum decimal places for commonly used format types, see the `FORMATS` command in the *SPSS Command Syntax Reference*, available in PDF form from the Help menu and also integrated into the overall Help system.

- This method is only available in write mode.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarFormat('numvar',5,2,0)
cur.CommitDictionary()
cur.close()
```

Format Types

Type	Description
1	**A.** Standard characters.
2	**AHEX.** Hexadecimal characters.
3	**COMMA.** Numbers with commas as grouping symbol and period as decimal indicator. For example: 1,234,567.89.
4	**DOLLAR.** Numbers with a leading dollar sign ($), commas as grouping symbol, and period as decimal indicator. For example: $1,234,567.89.
5	**F.** Standard numeric.
6	**IB.** Integer binary.
7	**PIBHEX.** Hexadecimal of PIB (positive integer binary).
8	**P.** Packed decimal.
9	**PIB.** Positive integer binary.
10	**PK.** Unsigned packed decimal.
11	**RB.** Real binary.
12	**RBHEX.** Hexadecimal of RB (real binary).
15	**Z.** Zoned decimal.
16	**N.** Restricted numeric.
17	**E.** Scientific notation.
20	**DATE.** International date of the general form dd-mmm-yyyy.
21	**TIME.** Time of the general form hh:mm:ss.ss.
22	**DATETIME.** Date and time of the general form dd-mmm-yyyy hh:mm:ss.ss.
23	**ADATE.** American date of the general form mm/dd/yyyy.
24	**JDATE.** Julian date of the general form yyyyddd.
25	**DTIME.** Days and time of the general form dd hh:mm:ss.ss.
26	**MONTH.** Month.
27	**MOYR.** Month and year.
28	**QYR.** Quarter and year of the general form qQyyyy.
29	**WKYR.** Week and year.
30	**PCT.** Percentage sign after numbers.
31	**DOT.** Numbers with period as grouping symbol and comma as decimal indicator. For example: 1.234.567,89.
32	**CCA.** Custom currency format 1.

Type	Description
33	**CCB.** Custom currency format 2.
34	**CCC.** Custom currency format 3.
35	**CCD.** Custom currency format 4.
36	**CCE.** Custom currency format 5.
37	**EDATE.** European date of the general form dd/mm/yyyy.
38	**SDATE.** Sortable date of the general form yyyy/mm/dd.

SetVarLabel Method

.SetVarLabel(varName,varLabel). *Sets the variable label for a new variable.* The argument *varName* is a string specifying the name of a new variable. The argument *varLabel* is a string specifying the label.

- This method is only available in write mode.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarLabel('numvar','New numeric variable')
cur.CommitDictionary()
cur.close()
```

SetVarMeasureLevel Method

.SetVarMeasureLevel(varName,measureLevel). *Sets the measurement level for a new variable.* The argument *varName* is a string specifying the name of a new variable. The argument *measureLevel* is an integer specifying the measurement level: 2 (nominal), 3 (ordinal), 4 (scale).

- This method is only available in write mode.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarMeasureLevel('numvar',3)
cur.CommitDictionary()
cur.close()
```

SetVarNameAndType Method

.SetVarNameAndType(varName,varType). *Creates one or more new variables in the active dataset.* The argument *varName* is a list or tuple of strings that specifies the name of each new variable. The argument *varType* is a list or tuple of integers specifying the variable type of each variable named in *varName*. *varName* and *varType* must be the same length. For creating a single variable you can also use the SetOneVarNameAndType method.

- This method is only available in write mode.
- 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 (maximum of 32767).

Example

```
*python_cursor_create_var.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
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['var4','strvar'],[0,8])
cur.SetVarFormat('var4',5,2,0)
cur.CommitDictionary()
for i in range(cur.GetCaseCount()):
   cur.fetchone()
   cur.SetValueNumeric('var4',4+10*(i+1))
   cur.SetValueChar('strvar','row' + str(i+1))
   cur.CommitCase()
cur.close()
END PROGRAM.
```

SetVarNMissingValues Method

.SetVarNMissingValues(varName,missingFormat,missingVal1,missingVal2,missingVal3). *Sets user-missing values for a new numeric variable.* The argument *varName* is a string specifying the name of a new numeric variable. The argument *missingFormat* has the value 1 for a discrete list of missing values (for example, 0, 9, 99), the value 2 for a range of missing values (for example, 9–99), and the value 3 for a combination of a discrete value and a range (for example, 0 and 9–99). Use the SetVarCMissingValues method to set missing values for new string variables.

- This method is only available in write mode.
- To specify *LO* and *HI* in missing value ranges, use the values returned by the spss.GetSPSSLowHigh function.

The meaning of the arguments *missingVal1*, *missingVal2*, and *missingVal3* depends on the value of *missingFormat* as shown in the following table.

missingFormat	missingVal1	missingVal2	missingVal3
0	Discrete value (optional)	Discrete value (optional)	Discrete value (optional)
1	Start point of range	End point of range	Not applicable
2	Start point of range	End point of range	Discrete value

Examples

Specify the three discrete missing values 0, 9, and 99 for a new variable.

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarNMissingValues('numvar',0,0,9,99)
cur.CommitDictionary()
cur.close()
```

Specify the range of missing values 9–99 for a new variable.

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarNMissingValues('numvar',1,9,99)
cur.CommitDictionary()
cur.close()
```

Specify the range of missing values 9–99 and the discrete missing value 0 for a new variable.

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarNMissingValues('numvar',2,9,99,0)
cur.CommitDictionary()
cur.close()
```

SetVarNValueLabel Method

.SetVarNValueLabel(varName,value,label). *Sets the value label of a single value for a new variable.* The argument *varName* is a string specifying the name of a new numeric variable. The argument *value* is a numeric value and *label* is the string specifying the label for this value. Use the SetVarCValueLabel method to set value labels for new string variables.

- This method is only available in write mode.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarNValueLabel('numvar',1,'female')
cur.CommitDictionary()
cur.close()
```

spss.DeleteXPathHandle Function

spss.DeleteXPathHandle(handle). *Deletes the XPath dictionary DOM or output DOM with the specified handle name.* The argument is a handle name that was defined with a previous `spss.CreateXPathDictionary` function or an SPSS `OMS` command.

Example

```
handle = 'demo'
spss.DeleteXPathHandle(handle)
```

spss.EndProcedure Function

spss.EndProcedure(). *Signals the end of pivot table or text block output.*

- `spss.EndProcedure` must be called to end output initiated with spss.StartProcedure.

spss.EvaluateXPath Function

spss.EvaluateXPath(handle,context,xpath). *Evaluates an XPath expression against a specified XPath DOM and returns the result as a list.* The argument *handle* specifies the particular XPath DOM and must be a valid handle name defined by a previous `spss.CreateXPathDictionary` function or SPSS `OMS` command. The argument *context* defines the XPath context for the expression and should be set to `"/dictionary"` for a dictionary DOM or `"/outputTree"` for an output XML DOM created by the `OMS` command. The argument *xpath* specifies the remainder of the XPath expression and must be quoted.

Example

```
#retrieve a list of all variable names for the active dataset.
handle='demo'
spss.CreateXPathDictionary(handle)
context = "/dictionary"
xpath = "variable/@name"
varnames = spss.EvaluateXPath(handle,context,xpath)
```

Example

```
*python_EvaluateXPath.sps.
*Use OMS and a Python program to determine the number of uniques values
  for a specific variable.
OMS SELECT TABLES
  /IF COMMANDs=['Frequencies'] SUBTYPES=['Frequencies']
 /DESTINATION FORMAT=OXML XMLWORKSPACE='freq_table'.
FREQUENCIES VARIABLES=var1.
OMSEND.

BEGIN PROGRAM.
import spss
handle='freq_table'
context="/outputTree"
#get rows that are totals by looking for varName attribute
#use the group element to skip split file category text attributes
xpath="//group/category[@varName]/@text"
values=spss.EvaluateXPath(handle,context,xpath)
#the "set" of values is the list of unique values
#and the length of that set is the number of unique values
uniqueValuesCount=len(set(values))
END PROGRAM.
```

Note: In the SPSS documentation, XPath examples for the `OMS` command use a namespace prefix in front of each element name (the prefix `oms:` is used in the OMS examples). Namespace prefixes are not valid for `EvaluateXPath`.

Documentation for the output schema and the dictionary schema is available from the Help system.

spss.GetCaseCount Function

spss.GetCaseCount(). *Returns the number of cases (rows) in the active dataset.*

Example

```
#python_GetCaseCount.sps
#build SAMPLE syntax of the general form:
#SAMPLE [NCases] FROM [TotalCases]
#Where Ncases = 10% truncated to integer
TotalCases=spss.GetCaseCount()
NCases=int(TotalCases/10)
command1="SAMPLE " + str(NCases) + " FROM " + str(TotalCases) +  "."
command2="Execute."
spss.Submit([command1, command2])
```

spss.GetDefaultPlugInVersion Function

spss.GetDefaultPlugInVersion(). *Returns the default version of the SPSS-Python Integration Plug-in used when driving SPSS from Python.* This function returns the string identifier for the default version of the SPSS-Python Integration Plug-in—for example, `"spss150"` for SPSS 15.0—to be used when driving the SPSS backend from a separate Python process, such as the Python interpreter or a Python IDE. You can change the default using the spss.SetDefaultPlugInVersion function.

Example

```
import spss
version = spss.GetDefaultPlugInVersion()
```

spss.GetHandleList Function

spss.GetHandleList(). *Returns a list of currently defined dictionary and output XML DOMs available for use with spss.EvaluateXpath.*

spss.GetLastErrorLevel and spss.GetLastErrorMessage Functions

spss.GetLastErrorLevel(). *Returns a number corresponding to an error in the preceding SPSS package function.*

- For the `spss.Submit` function, it returns the maximum SPSS error level for the submitted SPSS commands. SPSS error levels range from 1 to 5. An SPSS error level of 3 or higher causes an exception in Python.
- For other functions, it returns an error code with a value greater than 5.
- Error codes from 6 to 22 are from the SPSS XD API.
- Error codes from 1000 to 1013 are from the SPSS-Python integration package.

SPSS error levels (return codes), their meanings, and any associated behaviors are shown in the following table.

Table A-2
SPSS error levels

Value	Definition	Behavior
0	None	Command runs
1	Comment	Command runs
2	Warning	Command runs
3	Serious error	Command does not run, subsequent commands are processed
4	Fatal error	Command does not run, subsequent commands are not processed, and the current job terminates
5	Catastrophic error	Command does not run, subsequent commands are not processed, and the SPSS processor terminates

spss.GetLastErrorMessage(). *Returns a text message corresponding to an error in the preceding SPSS package function.*

- For the `spss.Submit` function, it returns text that indicates the severity level of the error for the last submitted SPSS command.

- For other functions in the SPSS package, it returns the error message text from the SPSS XD API or from Python.

Example

```
*python_GetLastErrorLevel.sps.
DATA LIST FREE/var1 var2.
BEGIN DATA
1 2 3 4
END DATA.
BEGIN PROGRAM.
try:
  spss.Submit("""
COMPUTE newvar=var1*10.
COMPUTE badvar=nonvar/4.
FREQUENCIES VARIABLES=ALL.
""")
except:
  errorLevel=str(spss.GetLastErrorLevel())
  errorMsg=spss.GetLastErrorMessage()
  print("Error level " + errorLevel + ": " + errorMsg)
  print("At least one command did not run.")
END PROGRAM.
```

- The first `COMPUTE` command and the `FREQUENCIES` command will run without errors, generating error values of 0.
- The second `COMPUTE` command will generate a level 3 error, triggering the exception handling in the `except` clause.

spss.GetSPSSLowHigh Function

spss.GetSPSSLowHigh(). *Returns the values SPSS uses for LO and HI as a tuple of two values*. The first element in the tuple is the value for *LO* and the second is the value for *HI*. These values can used to specify missing value ranges for new numeric variables with the SetVarNMissingValues method.

Example

```
import spss
spsslow, spsshigh = spss.GetSPSSLowHigh()
```

spss.GetVarAttributeNames Function

spss.GetVarAttributeNames(index). *Returns the names of any variable attributes, as a tuple, for the variable in the active dataset indicated by the index value.* The argument is the index value. Index values represent position in the active dataset, starting with 0 for the first variable in file order.

Example

```
#Create a list of variables that have a specified attribute
import spss
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
```

spss.GetVarAttributes Function

spss.GetVarAttributes(index,attrName). *Returns the attribute values, as a tuple, for the specified attribute of the variable in the active dataset indicated by the index value.* The argument *index* is the index value. Index values represent position in the active dataset, starting with 0 for the first variable in file order. The argument *attrName* is a string that specifies the name of the attribute—for instance, a name returned by `GetVarAttributeNames`.

Example

```
#Create a list of variables whose attribute array contains
#a specified value
import spss
varList=[]
attrName='demographicvartypes'
attrVal='2'
for i in range(spss.GetVariableCount()):
   try:
      if(attrVal in spss.GetVarAttributes(i,attrName)):
         varList.append(spss.GetVariableName(i))
   except:
      pass
if varList:
   print "Variables with attribute value " + attrVal + \
         " for attribute " + attrName + ":"
   print '\n'.join(varList)
else:
   print "No variables have the attribute value " + attrVal + \
         " for attribute " + attrName
```

spss.GetVariableCount Function

spss.GetVariableCount(). *Returns the number of variables in the active dataset.*

Example

```
#python_GetVariableCount.sps
#build a list of all variables by using the value of
#spssGetVariableCount to set the number of for loop interations
varcount=spss.GetVariableCount()
varlist=[]
for i in xrange(varcount):
    varlist.append(spss.GetVariableName(i))
```

spss.GetVariableFormat Function

GetVariableFormat(index). *Returns a string containing the display format for the variable in the active dataset indicated by the index value.* The argument is the index value. Index values represent position in the active dataset, starting with 0 for the first variable in file order.

- The character portion of the format string is always returned in all upper case.

- Each format string contains a numeric component after the format name that indicates the defined width, and optionally, the number of decimal positions for numeric formats. For example, A4 is a string format with a maximum width of four bytes, and F8.2 is a standard numeric format with a display format of eight digits, including two decimal positions and a decimal indicator.

Format Values

- **A.** Standard characters.
- **ADATE.** American date of the general form mm/dd/yyyy.
- **JDATE.** Julian date of the general form yyyyddd.
- **AHEX.** Hexadecimal characters.
- **CCA.** Custom currency format 1.
- **CCB.** Custom currency format 2.
- **CCC.** Custom currency format 3.
- **CCD.** Custom currency format 4.
- **CCE.** Custom currency format 5.
- **COMMA.** Numbers with commas as grouping symbol and period as decimal indicator. For example: 1,234,567.89.
- **DATE.** International date of the general form dd-mmm-yyyy.
- **DATETIME.** Date and time of the general form dd-mmm-yyyy hh:mm:ss.ss.
- **DOLLAR.** Numbers with a leading dollar sign ($), commas as grouping symbol, and period as decimal indicator. For example: $1,234,567.89.
- **F.** Standard numeric.
- **IB.** Integer binary.
- **PIBHEX.** Hexadecimal of PIB (positive integer binary).
- **DOT.** Numbers with period as grouping symbol and comma as decimal indicator. For example: 1.234.567,89.
- **DTIME.** Days and time of the general form dd hh:mm:ss.ss.
- **E.** Scientific notation.
- **EDATE.** European date of the general form dd/mm/yyyy.
- **MONTH.** Month.

- **MOYR.** Month and year.
- **N.** Restricted numeric.
- **P.** Packed decimal.
- **PIB.** Positive integer binary.
- **PK.** Unsigned packed decimal.
- **QYR.** Quarter and year of the general form qQyyyy.
- **WKYR.** Week and year.
- **PCT.** Percentage sign after numbers.
- **RB.** Real binary.
- **RBHEX.** Hexadecimal of RB (real binary).
- **SDATE.** Sortable date of the general form yyyy/mm/dd.
- **TIME.** Time of the general form hh:mm:ss.ss.
- **WKDAY.** Day of the week.
- **Z.** Zoned decimal.

Example

```
*python_GetVariableFormat.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 spss
#create a list of all formats and a list of time format variables
varcount=spss.GetVariableCount()
formatList=[]
timeVarList=[]
for i in xrange(varcount):
  formatList.append(spss.GetVariableFormat(i))
  #check to see if it's a time format
  if spss.GetVariableFormat(i).find("TIME")==0:
     timeVarList.append(spss.GetVariableName(i))
print formatList
print timeVarList
END PROGRAM.
```

spss.GetVariableLabel Function

spss.GetVariableLabel(index). *Returns a character string containing the variable label for the variable in the active dataset indicated by the index value.* The argument is the index value. Index values represent position in the active dataset, starting with 0 for the first variable in file order. If the variable does not have a defined value label, a null string is returned.

Example

```
#create a list of all variable labels
varcount=spss.GetVariableCount()
labellist=[]
for i in xrange(varcount):
  labellist.append(spss.GetVariableLabel(i))
```

spss.GetVariableMeasurementLevel Function

spss.GetVariableMeasurementLevel(index). *Returns a string value that indicates the measurement level for the variable in the active dataset indicated by the index value.* The argument is the index value. Index values represent position in the active dataset, starting with 0 for the first variable in file order. The value returned can be: `"nominal"`, `"ordinal"`, `"scale"`, or `"unknown"`.

"Unknown" occurs only for numeric variables prior to the first data pass when the measurement level has not been explicitly set, such as data read from an external source or newly created variables. The measurement level for string variables is always known.

Example

```
#build a string containing  scale variable names
varcount=spss.GetVariableCount()
ScaleVarList=''
for i in xrange(varcount):
  if spss.GetVariableMeasurementLevel(i)=="scale":
    ScaleVarList=ScaleVarList + " " + spss.GetVariableName(i)
```

spss.GetVariableName Function

GetVariableName(index). *Returns a character string containing the variable name for the variable in the active dataset indicated by the index value.* The argument is the index value. Index values represent position in the active dataset, starting with 0 for the first variable in file order.

Example

```
#python_GetVariableName.sps
#get names of first and last variables in the file
#last variable is index value N-1 because index values start at 0
firstVar=spss.GetVariableName(0)
lastVar=spss.GetVariableName(spss.GetVariableCount()-1)
print firstVar, lastVar
#sort the data file in alphabetic order of variable names
varlist=[]
varcount=spss.GetVariableCount()
for i in xrange(varcount):
  varlist.append(spss.GetVariableName(i))
sortedlist=' '.join(sorted(varlist))
spss.Submit(
  ["ADD FILES FILE=* /KEEP ",sortedlist, ".", "EXECUTE."])
```

spss.GetVariableType Function

GetVariableType(index). *Returns 0 for numeric variables or the defined length for string variables for the variable in the active dataset indicated by the index value.* The argument is the index value. Index values represent position in the active dataset, starting with 0 for the first variable in file order.

Example

```
#python_GetVariableType.sps
#create separate strings of numeric and string variables
numericvars=''
stringvars=''
varcount=spss.GetVariableCount()
for i in xrange(varcount):
  if spss.GetVariableType(i) > 0:
    stringvars=stringvars + " " + spss.GetVariableName(i)
  else:
    numericvars=numericvars + " " + spss.GetVariableName(i)
```

spss.GetVarMissingValues Function

spss.GetVarMissingValues(index). *Returns the user-missing values for the variable in the active dataset indicated by the index value.* The argument is the index value. Index values represent position in the active dataset, starting with 0 for the first variable in file order.

- The result is 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 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.
- If there are no missing values, the result is `(0,None,None,None)`.

Table A-3
Structure of the result

tuple[0]	tuple[1]	tuple[2]	tuple[3]
0	Discrete value or *None*	Discrete value or *None*	Discrete value or *None*
1	Start point of range	End point of range	*None*
2	Start point of range	End point of range	Discrete value

Example

```
#List all variables without user-missing values
nomissList=[]
for i in range(spss.GetVariableCount()):
   missing=spss.GetVarMissingValues(i)
   if (missing[0]==0 and missing[1]==None):
      nomissList.append(spss.GetVariableName(i))
if nomissList:
   print "Variables without user-missing values:"
   print '\n'.join(nomissList)
else:
   print "All variables have user-missing values"
```

spss.GetWeightVar Function

spss.GetWeightVar(). *Returns the name of the weight variable, or None if unweighted.*

Example

```
import spss
weightVar = spss.GetWeightVar()
```

spss.GetXmlUtf16 Function

spss.GetXmlUtf16(handle, filespec). . *Writes the XML for the specified handle (dictionary or output XML) to a file or returns the XML if no filename is specified.* When writing and debugging XPath expressions, it is often useful to have a sample file that shows the XML structure. This function is particularly useful for dictionary DOMs, since there are not any alternative methods for writing and viewing the XML structure. (For output XML, the `OMS` command can also write XML to a file.) You can also use this function to retrieve the XML for a specified handle, enabling you to process it with third-party utilities like XML parsers.

Example

```
handle = "activedataset"
spss.CreateXPathDictionary(handle)
spss.GetXmlUtf16(handle,'c:/temp/temp.xml')
```

spss.HasCursor Function

spss.HasCursor(). *Returns an integer indicating whether there is an open cursor.* A value of 0 indicates there is no open cursor, and a value of 1 indicates there is an open cursor. Cursors allow you to read data from the active dataset, create new variables in the active dataset, and append cases to the active dataset. For information on working with cursors, see the topic on the Cursor class on p. 460.

spss.IsOutputOn Function

spss.IsOutputOn(). *Returns the status of SPSS output display in Python.* The result is Boolean—*true* if output display is on in Python, *false* if it is off. For more information, see spss.SetOutput Function on p. 502.

Example

```
import spss
```

```
spss.SetOutput("on")
if spss.IsOutputOn():
    print "The current spssOutput setting is 'on'."
else:
    print "The current spssOutput setting is  'off'."
```

spss.PyInvokeSpss.IsXDriven Function

spss.PyInvokeSpss.IsXDriven(). *Checks to see how the SPSS backend is being run.* The result is 1 if Python is controlling the SPSS backend or 0 if SPSS is controlling the SPSS backend.

Example

```
import spss
spss.Submit("""
GET FILE
 'c:/program files/spss/employee data.sav'.
""")
isxd = spss.PyInvokeSpss.IsXDriven()
if isxd==1:
  print "Python is running SPSS."
else:
  print "SPSS is running Python."
```

spss.SetDefaultPlugInVersion Function

spss.SetDefaultPlugInVersion(value). *Controls the default version of the SPSS-Python Integration Plug-in used when driving SPSS from Python.* This function is intended for use when driving the SPSS backend from a separate Python process, such as the Python interpreter or a Python IDE. It determines which installed version of the SPSS-Python Integration Plug-in (and the compatible version of the SPSS backend) to use when driving SPSS from Python. It does not determine the version of the plug-in to use when running Python code from within SPSS—that is determined by SPSS. The value of the argument is a quoted string specifying a plug-in version—for example, `"spss140"` or `"spss150"` for SPSS 14.0 or SPSS 15.0. The strings representing the installed versions of the plug-in are available from the function spss.ShowInstalledPlugInVersions.

Example

```
import spss
```

```
spss.SetDefaultPlugInVersion("spss140")
```

spss.SetMacroValue Function

spss.SetMacroValue(name, value). *Defines an SPSS macro variable that can be used outside a program block in SPSS command syntax.* The first argument is the macro name, and the second argument is the macro value. Both arguments must resolve to strings.

Example

```
*python_SetMacroValue.sps.
DATA LIST FREE /var1 var2 var3 var4.
begin data
1 2 3 4
end data.
VARIABLE LEVEL var1 var3 (scale) var2 var4 (nominal).

BEGIN PROGRAM.
import spss
macroValue=[]
macroName="!NominalVars"
varcount=spss.GetVariableCount()
for i in xrange(varcount):
  if spss.GetVariableMeasurementLevel(i)=="nominal":
    macroValue.append(spss.GetVariableName(i))
spss.SetMacroValue(macroName, macroValue)
END PROGRAM.
FREQUENCIES VARIABLES=!NominalVars.
```

spss.SetOutput Function

spss.SetOutput("value"). *Controls the display of SPSS output in Python when running SPSS from Python.* Output is displayed as standard output, and charts and classification trees are not included. When running Python from SPSS within program blocks (`BEGIN PROGRAM-END PROGRAM`), this function has no effect. The value of the argument is a quoted string:

- **"on".** Display SPSS output in Python.
- **"off".** Do not display SPSS output in Python.

Example

```
import spss
spss.SetOutput("on")
```

spss.ShowInstalledPlugInVersions Function

spss.ShowInstalledPlugInVersions(). *Returns the installed versions of the SPSS-Python Integration Plug-in.* This function is intended for use when driving the SPSS backend from a separate Python process, such as the Python interpreter or a Python IDE. It returns a list of string identifiers for the installed versions—for example, `["spss140","spss150"]` for SPSS 14.0 and SPSS 15.0. Use an identifier from this list as the argument to the spss.SetDefaultPlugInVersion function (only intended for use when driving the SPSS backend from Python).

Example

```
import spss
versionList = spss.ShowInstalledPlugInVersions()
```

spss.SplitChange Function

spss.SplitChange(outputName). *Used to process splits when creating pivot tables from data that have splits.* The argument *outputName* is the name associated with the output, as specified on the associated call to the `StartProcedure` function. For more information, see spss.StartProcedure Function on p. 506.

- This function should be called after detecting a split and reading the first case of the new split. It should also be called after reading the first case in the active dataset.
- The creation of pivot table output does not support operations involving data in different split groups. When working with splits, each split should be treated as a separate set of data.
- Use the `SPLIT FILE` command to control whether split-file groups will be displayed in the same table or in separate tables. The `SPLIT FILE` command should be called before the `StartProcedure` function.
- The IsEndSplit method from the `Cursor` class is used to detect a split change.

Example

In this example, a split is created and separate averages are calculated for the split groups. Results for different split groups are shown in a single pivot table. In order to understand the example, you will need to be familiar with creating pivot tables using the BasePivotTable class and creating output with the spss.StartProcedure function.

```
import spss
from spss import CellText
from spss import FormatSpec

spss.Submit(r"""
GET FILE="c:/program files/spss/employee data.sav".
SORT CASES BY GENDER.
SPLIT FILE LAYERED BY GENDER.
""")

spss.StartProcedure("spss.com.demo")

table = spss.BasePivotTable("Table Title","OMS table subtype")
table.Append(spss.Dimension.Place.row,"Minority Classification")
table.Append(spss.Dimension.Place.column,"coldim",hideName=True)

cur=spss.Cursor()
salary = 0; salarym = 0; n = 0; m = 0
minorityIndex = 9
salaryIndex = 5

row = cur.fetchone()
spss.SplitChange("spss.com.demo")
while True:
   if cur.IsEndSplit():
      if n>0:
         salary=salary/n
      if m>0:
         salarym=salarym/m
      # Populate the pivot table with values for the previous split group
      table[(CellText.String("No"),CellText.String("Average Salary"))] = \
             CellText.Number(salary,FormatSpec.Count)
      table[(CellText.String("Yes"),CellText.String("Average Salary"))] = \
             CellText.Number(salarym,FormatSpec.Count)
      salary=0; salarym=0; n = 0; m = 0
      # Try to fetch the first case of the next split group
      row=cur.fetchone()
      if not None==row:
         spss.SplitChange("spss.com.demo")
      else:
         #There are no more cases, so quit
         break
   if row[minorityIndex]==1:
      salarym += row[salaryIndex]
      m += 1
   elif row[minorityIndex]==0:
      salary += row[salaryIndex]
      n += 1
   row=cur.fetchone()

cur.close()
spss.EndProcedure()
```

- The `spss.Submit` function is used to submit command syntax to create a split on a gender variable. The `LAYERED` subcommand on the `SPLIT FILE` command indicates that results for different split groups are to be displayed in the same table. Notice that the command syntax is executed before calling `spss.StartProcedure`.

- The `spss.SplitChange` function is called after fetching the first case from the active dataset. This is required so that the pivot table output for the first split group is handled correctly.
- Split changes are detected using the IsEndSplit method from the `Cursor` class. Once a split change is detected, the pivot table is populated with the results from the previous split.
- The value returned from the `fetchone` method is *None* at a split boundary. Once a split has been detected, you will need to call `fetchone` again to retrieve the first case of the new split group, followed by `spss.SplitChange`. *Note*: `IsEndSplit` returns *true* when the end of the dataset has been reached. 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`, as shown in this example.

spss.StartProcedure Function

spss.StartProcedure(outputName). *Signals the beginning of pivot table or text block output.* Pivot table and text block output is typically associated with procedures. Procedures are user-defined Python functions or custom Python classes that can read the data, perform computations, add new variables and/or new cases to the active dataset, and produce pivot table output and text blocks in the SPSS Viewer. Procedures have almost the same capabilities as built-in SPSS procedures, such as `DESCRIPTIVES` and `REGRESSION`, but they are written in Python by users. You read the data and create new variables and/or new cases using the cursor class. Pivot tables are created using the BasePivotTable class. Text blocks are created using the TextBlock class.

- The argument *outputName* 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. And finally it is the name associated with this output for use with autoscripts.
- In order that names associated with output not conflict with names of existing SPSS commands (when working with OMS or autoscripts), SPSS recommends that they have the form *yourcompanyname.com.procedurename*. When working with autoscripts, note that periods (.) contained in an output name are replaced with

zeros (0), dashes are replaced with underscores, and spaces are removed in the associated autoscript's name. Avoid any other punctuation characters that might create illegal names in a programming language. For instance, output associated with the name `Smith & Jones, Ltd` generates an associated autoscript named `Smith&Jones,Ltd`, which would be illegal as part of a subroutine name in Sax Basic.

- Within a `StartProcedure-EndProcedure` block you cannot use the `spss.Submit` function. You cannot nest `StartProcedure-EndProcedure` blocks.
- Within a `StartProcedure-EndProcedure` block, you can create a single cursor instance.
- Output from `StartProcedure-EndProcedure` blocks does not support operations involving data in different split groups. When working with splits, each split should be treated as a separate set of data. To cause results from different split groups to display properly in custom pivot tables, use the SplitChange function. Use the IsEndSplit method from the `Cursor` class to determine a split change.
- `spss.StartProcedure` must be followed by spss.EndProcedure.

Example

As an example, we will create a procedure that calculates group means for a selected variable using a specified categorical variable to define the groups. The output of the procedure is a pivot table displaying the group means. For an alternative approach to creating the same procedure, but with a custom class, see the example for the spss.BaseProcedure class.

```
def groupMeans(groupVar,sumVar):

    #Determine variable indexes from variable names
    varCount = spss.GetVariableCount()
    groupIndex = 0
    sumIndex = 0
    for i in range(varCount):
        varName = spss.GetVariableName(i)
        if varName == groupVar:
            groupIndex = i
            continue
        elif varName == sumVar:
            sumIndex = i
            continue

    varIndex = [groupIndex,sumIndex]
    cur = spss.Cursor(varIndex)
    Counts={};Statistic={}

    #Calculate group sums
    for i in range(cur.GetCaseCount()):
        row = cur.fetchone()
        cat=int(row[0])
        Counts[cat]=Counts.get(cat,0) + 1
        Statistic[cat]=Statistic.get(cat,0) + row[1]

    cur.close()

    #Call StartProcedure
    spss.StartProcedure("mycompany.com.groupMeans")

    #Create a pivot table
    table = spss.BasePivotTable("Group Means","OMS table subtype")
    table.Append(spss.Dimension.Place.row,
                 spss.GetVariableLabel(groupIndex))
    table.Append(spss.Dimension.Place.column,
                 spss.GetVariableLabel(sumIndex))

    category2 = spss.CellText.String("Mean")
    for cat in sorted(Counts):
        category1 = spss.CellText.Number(cat)
        table[(category1,category2)] = \
        spss.CellText.Number(Statistic[cat]/Counts[cat])

    #Call EndProcedure
    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*).

- The name associated with output from this procedure is *mycompany.com.groupMeans*. The output consists of a pivot table populated with the group means.
- `spss.EndProcedure` marks the end of output creation.

Saving and Running Procedures

To use a procedure you have written, you save it in a Python module on the Python search path so that you can call it. 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. To be sure that Python can find your new module, you may want to save it to your Python "site-packages" directory, typically *C:\Python24\Lib\site-packages*.

For the example procedure described above, you might choose to save the definition of the `groupMeans` function to a Python module named *myprocs.py*. And be sure to include an `import spss` statement in the module. Sample command syntax to run the function is:

```
import spss, myprocs
spss.Submit("get file='c:/program files/spss15/Employee data.sav'.")
myprocs.groupMeans("educ","salary")
```

- The import statement containing `myprocs` makes the contents of the Python module *myprocs.py* available to the current session (assuming that the module is on the Python search path).
- `myprocs.groupMeans("educ","salary")` runs the `groupMeans` function for the variables *educ* and *salary* in *c:/program files/spss15/Employee data.sav*.

Result

Figure A-15
Output from the groupMeans procedure

Educational Level (years)	Current Salary Mean
8	24399
12	25887
14	31625
15	31685
16	48226
17	59527
18	65128
19	72520
20	64313
21	65000

spss.StartSPSS Function

spss.StartSPSS(). *Starts an SPSS session.*

- When running SPSS from Python, this function starts an SPSS session. The function has no effect if an SPSS session is already running. *Note*: The `spss.Submit` function automatically starts an SPSS session.
- This function has no effect when running Python from SPSS (within program blocks defined by `BEGIN PROGRAM-END PROGRAM`).

spss.StopSPSS Function

spss.StopSPSS(). *Stops SPSS, ending the SPSS session.*

- This function is ignored when running Python from SPSS (within program blocks defined by `BEGIN PROGRAM-END PROGRAM`).
- When running SPSS from Python, this function ends the SPSS session, and any subsequent `spss.Submit` functions that restart SPSS will not have access to the active dataset or to any other session-specific settings (for example, `OMS` output routing commands) from the previous session.

Example: Running SPSS from Python

```
#run_spss_from_python.py
```

```
import spss
#start SPSS and run some commands
#including one that defines an active dataset
spss.Submit("""
GET FILE 'c:/program files/spss/employee data.sav'.
FREQUENCIES VARIABLES=gender jobcat.
""")
#shutdown SPSS
spss.StopSPSS()
#insert bunch of Python statements
#start SPSS again and run some commands without defining
#an active dataset results in an error
spss.Submit("""
FREQUENCIES VARIABLES=gender jobcat.
""")
```

Example: Running Python from SPSS

```
*run_python_from_spss.sps.
BEGIN PROGRAM.
import spss
#start SPSS and run some commands
#including one that defines an active dataset
spss.Submit("""
GET FILE 'c:/program files/spss/employee data.sav'.
FREQUENCIES VARIABLES=gender jobcat.
""")
#following function is ignored when running Python from SPSS
spss.StopSPSS()
#active dataset still exists and subsequent spss.Submit functions
#will work with that active dataset.
spss.Submit("""
FREQUENCIES VARIABLES=gender jobcat.
""")
END PROGRAM.
```

spss.Submit Function

spss.Submit(command text). *Submits the command text to SPSS for processing.* The argument can be a quoted string, a list, or a tuple.

- The argument should resolve to one or more complete SPSS commands.
- For lists and tuples, each element must resolve to a string.

- You can also use the Python triple-quoted string convention to specify blocks of SPSS commands on multiple lines that more closely resemble the way you might normally write command syntax.
- If SPSS is not currently running (when running SPSS from Python), `spss.Submit` will start the SPSS backend processor.

Example

```
*python_Submit.sps.
BEGIN PROGRAM.
import spss
#run a single command
spss.Submit("DISPLAY NAMES.")
#run two commands
spss.Submit(["DISPLAY NAMES.", "SHOW $VARS."])

#build and run two commands
command1="FREQUENCIES VARIABLES=var1."
command2="DESCRIPTIVES VARIABLES=var3."
spss.Submit([command1, command2])
END PROGRAM.
```

Example: Triple-Quoted Strings

```
*python_Submit_triple_quote.sps.
BEGIN PROGRAM.
import spss
file="c:/program files/spss/tutorial/sample_files/demo.sav"
varlist="marital gender inccat"
spss.Submit("""
GET FILE='%s'.
FREQUENCIES VARIABLES=%s
  /STATISTICS NONE
  /BARCHART.
""" %(file,varlist))
END PROGRAM.
```

Within the triple-quoted string, `%s` is used for string substitution; thus, you can insert Python variables that resolve to strings in the quoted block of commands.

spss.TextBlock Class

spss.TextBlock(name,content,outline). *Creates and populates a text block item in the Viewer.* The argument *name* is a string that specifies the name of this item in the outline pane of the Viewer. The argument *content* is a string that specifies a single line of text.

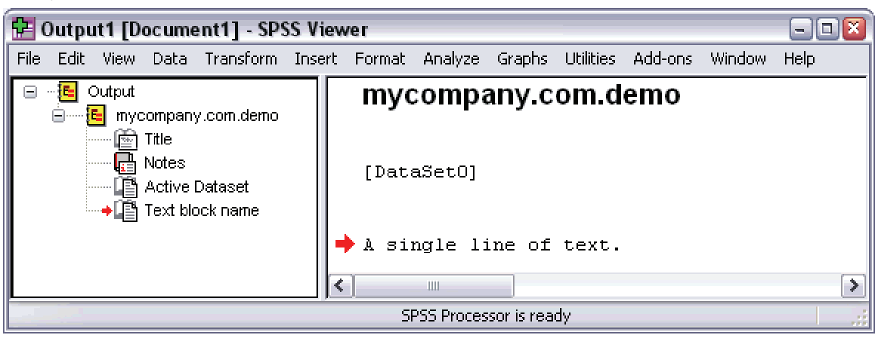
To add additional lines, use the append method. The optional argument *outline* is a string that specifies a title for this item that appears in the outline pane of the Viewer. The item for the text block itself will be placed one level deeper than the item for the *outline* title. If *outline* is omitted, the Viewer item for the text block will be placed one level deeper than the root item for the output containing the text block.

An instance of the `TextBlock` class can only be used within a `StartProcedure-EndProcedure` block or within a custom procedure class based on the `spss.BaseProcedure` class.

Example

```
import spss
spss.StartProcedure("mycompany.com.demo")
textBlock = spss.TextBlock("Text block name",
                           "A single line of text.")
spss.EndProcedure()
```

Figure A-16
Sample text block



- This example shows how to generate a text block within a `spss.StartProcedure-spss.EndProcedure` block. The output will be contained under an item named *mycompany.com.demo* in the outline pane of the Viewer.
- The variable *textBlock* stores a reference to the instance of the text block object. You will need this object reference if you intend to append additional lines to the text block with the `append` method.

append Method

.append(line,skip). *Appends lines to an existing text block.* The argument *line* is a string that specifies a single line of text. The optional argument *skip* specifies the number of new lines to create when appending the specified line. The default is 1 and results in appending the single specified line. Integers greater than 1 will result in blank lines preceding the appended line. For example, specifying `skip=3` will result in two blank lines before the appended line.

Example

```
import spss
spss.StartProcedure("mycompany.com.demo")
textBlock = spss.TextBlock("Text block name",
                           "A single line of text.")
textBlock.append("A second line of text.")
textBlock.append("A third line of text preceded by a blank line.",skip=2)
spss.EndProcedure()
```

Index

active dataset
appending cases, 275, 284, 460, 465
creating a new dataset, 307, 311
creating new variables, 275, 282, 288, 460, 462, 477, 486
reading into Python, 275–276, 460
ADD DOCUMENT (command), 99
ADD FILES (command), 70
ADD VALUE LABELS (command), 95
AGGREGATE (command), 76
aggregating data, 76
AllocNewVarsBuffer method, 288, 468
Append method, 429
Append method (BasePivotTable class), 437
append method (TextBlock class), 514
APPLY DICTIONARY (command), 98
attributes
custom variable attributes, 493
Attributes method, 271
automation objects in Python, 366
average
mean, 106

BasePivotTable class, 425
Append method, 429, 437
Caption method, 438
CellText class, 450
Footnotes method, 438
GetDefaultFormatSpec method, 439
HideTitle method, 439
Insert method, 429, 439
SetCategories method, 431, 441
SetCell method, 442
SetCellsByColumn method, 432, 444
SetCellsByRow method, 432, 443
SetDefaultFormatSpec method, 446
SimplePivotTable method, 427, 447
BEGIN PROGRAM (command), 213, 223
binning scale variables, 102
bootstrapping
with OMS, 160

Caption method, 438
case
changing case of string values, 109
case count, 490
case number
system variable $casenum, 15
$casenum
with SELECT IF command, 15
cases
case number, 15
weighting cases to replicate crosstabulation, 79
CASESTOVARS (command), 83
categorical variables, 95
CellText class, 450
Number class, 450
String class, 453
toNumber method, 454
toString method, 455
VarName class, 453
VarValue class, 454
cleaning data, 123, 130
close method, 469
CloseDesignatedOutput method, 364
combining data files, 66
command syntax
invoking command file with INSERT command, 19
syntax rules for INSERT files, 19
commands
displaying in the log, 7
COMMENT (command), 16
macro names, 16
comments, 16
CommitCase method, 470
CommitDictionary method, 470
COMPUTE (command), 105
CONCAT (function), 110

concatenating string values, 109
conditional loops, 149
conditional transformations, 134
connect string
  reading databases, 25
CreateDatasetOutput, 318
CreateXMLOutput, 318
CreateXPathDictionary, 267, 459
CSV data, 40
CTIME.DAYS (function), 119
CTIME.HOURS (function), 119
CTIME.MINUTES (function), 119
Cursor class, 460
  AllocNewVarsBuffer method, 288, 468
  append mode, 465
  close method, 469
  CommitCase method, 470
  CommitDictionary method, 470
  EndChanges method, 471
  fetchall method, 471
  fetchmany method, 473
  fetchone method, 474
  IsEndSplit method, 280, 475
  read mode, 460
  reset method, 476
  SetFetchVarList method, 477
  SetOneVarNameAndType method, 477
  SetUserMissingInclude method, 478
  SetValueChar method, 480
  SetValueNumeric method, 480
  SetVarAlignment method, 481
  SetVarAttributes method, 482
  SetVarCMissingValues method, 482
  SetVarCValueLabel method, 483
  SetVarFormat method, 483
  SetVarLabel method, 485
  SetVarMeasureLevel method, 485
  SetVarNameAndType method, 486
  SetVarNMissingValues method, 486
  SetVarNValueLabel method, 488
  write mode, 462

data
  appending cases, 275, 284, 460, 465
  creating a new dataset, 307, 311
  creating new variables, 275, 282, 288, 460, 462
  fetching data in Python, 275–276, 460
  reading active dataset into Python, 275–276, 460
data files
  activating an open dataset, 62
  aggregating, 76
  making cases from variables, 86
  making variables from cases, 83
  merging, 66, 70
  multiple open datasets, 62
  read-only, 9
  saving output as SPSS-format data files, 156
  transposing, 82
  updating, 74
DATA LIST (command)
  delimited data, 37
  fixed-width data, 41
  freefield data, 37
data types, 260, 498
databases
  connect string, 25
  Database Wizard, 24
  GET DATA (command), 24
  installing drivers, 22
  outer joins, 27
  reading data, 22
  reading multiple tables, 26
  selecting tables, 25
  SQL statements, 25
  writing data to a database, 184
DATAFILE ATTRIBUTE (command), 99
datafile attributes
  retrieving, 271
DATASET ACTIVATE (command), 62
DATASET COPY (command), 62
DATASET NAME (command), 62
DATE.MDY (function), 118
DATE.MOYR (function), 118
dates, 114
  combining multiple date components, 118
  computing intervals, 119
  extracting date components, 121
  functions, 117
  input and display formats, 115
  reading datetime values into Python, 298
  setting date format variables from Python, 305

days
  calculating number of, 120
DeleteXPathHandle, 314, 488
DETECTANOMALY (command), 130
dictionary
  CreateXPathDictionary, 267, 459
  reading SPSS dictionary information in Python, 262, 267, 489
  writing to an XML file, 500
DO IF (command), 134
  conditions that evaluate to missing, 136
DO REPEAT (command), 138
duplicate cases
  filtering, 127
  finding, 127

EndChanges method, 471
EndProcedure, 488
error handling in Python, 225, 247
error messages, 226, 491
EvaluateXPath, 314, 489
Excel
  exporting results, 188
  reading Excel files, 29
  saving data in Excel format, 184
EXECUTE (command), 12
executing SPSS commands in Python, 215, 511
ExportDesignatedOutput method, 364
exporting
  data and results, 155
  data in Excel format, 184
  data in SAS format, 182
  data in Stata format, 183
  data to a database, 184
  HTML, 155
  Output Management System, 155
  text, 155
  XML, 155

fetchall method, 471
fetching data in Python, 275–276, 460
fetchmany method, 473
fetchone method, 474
FILE HANDLE (command)
  defining wide records with LRCL, 45
FILE LABEL (command), 99
file properties, 99
FILTER (command), 128, 137
filtering duplicates, 127
FLIP (command), 82
Footnotes method, 438
format of variables, 257, 494
FORMATS (command), 116
functions
  arithmetic, 106
  date and time, 117
  random distribution, 107
  statistical, 106

GET DATA (command)
  TYPE=ODBC subcommand, 24
  TYPE=TXT subcommand, 40
  TYPE=XLS subcommand, 29
GetCaseCount, 490
GetDefaultFormatSpec method, 439
GetDefaultPlugInVersion, 490
GetDesignatedOutput method, 364, 366
GetHandleList, 491
GetLastErrorlevel, 491
GetLastErrorMessage, 491
GetSPSSInstallDir, 245
GetSPSSLowHigh, 492
GetValuesFromXMLWorkspace, 219, 318
GetVarAttributeNames, 493
GetVarAttributes, 493
GetVariableCount, 254, 494
GetVariableFormat, 257, 494
GetVariableLabel, 259, 497
GetVariableMeasurementLevel, 256, 497
GetVariableName, 254, 498
GetVariableType, 260, 498
GetVarMissingValues, 261, 499
GetWeightVar, 499
GetXmlUtf16, 267, 317, 500
grouped text data, 49

HasCursor, 500
HideTitle method, 439
hierarchical text data, 52

IDE
  using a Python IDE to drive SPSS, 226
IF (command), 134
if/then/else logic, 134
importing data, 22
  Excel, 29
  SAS format, 59
  text, 35
INDEX (function), 113
INSERT (command), 19
INSERT files
  command syntax rules, 19
Insert method, 429, 439
invalid values
  excluding, 126
  finding, 123
IsEndSplit method, 280, 475
IsOutputOn, 500

labels
  value, 94, 267
  variable, 94, 259, 497
LAG (function), 13
LAST (subcommand)
  MATCH FILES command, 128
leading zeros
  preserving with N format, 110
level of measurement, 95
locales, 390
log
  displaying commands, 7
logical variables, 134
long records
  defining with FILE HANDLE command, 45
lookup file, 69
loops
  conditional, 149
  default maximum number of loops, 152
  indexing clause, 146
  LOOP (command), 143
  nested, 146
  using XSAVE to build a data file, 150
LOWER (function), 109

macro variables in Python, 224, 502
macros
  macro names in comments, 16
MATCH FILES (command), 69
  LAST subcommand, 128
MEAN (function), 106
measurement level, 95, 256, 485, 497
merging data files, 66
  same cases, different variables, 66
  same variables, different cases, 70
  table lookup file, 69
missing values
  identifying cases with missing values, 295
  in DO IF structures, 136
  retrieving user missing value definitions, 261, 499
  setting missing values from Python, 302, 482, 486
  skipping cases with missing values, 294
  user-missing, 95
  when reading data into Python, 279, 461
MISSING VALUES (command), 16, 95
mixed format text data, 48
MOD (function), 106
modulus, 106
multiple data sources, 62

N format, 110
names of variables, 254, 498
nested loops, 146
nested text data, 52
nominal variables, 95
normal distribution, 108
NUMBER (function), 110, 117
Number class, 343, 450
number of cases (rows), 490
number of variables, 254, 494
numeric variables, 260, 498
NVALID (function), 107

ODBC, 22
  installing drivers, 22
OLE DB, 23
OMS
  bootstrapping, 160
  using XSLT with OXML, 165
OMS (command)
  exporting results, 155

ordinal variables, 95
outer joins
  reading databases, 27
output
  reading SPSS output results in Python, 219, 314, 318, 489
  using as input with OMS, 156
output documents, 189
Output Management System (OMS), 156
OXML, 165
  reading output XML in Python, 219, 314, 318, 489

parsing string values, 110
PDF
  exporting results, 188
PERMISSIONS (subcommand)
  SAVE command, 9
pivot tables, 335, 425
  formatting numeric cells, 342
  modifying in Python, 366
  using SPSS variable names or values for categories or cells, 340
Poisson distribution, 108
PowerPoint
  exporting results, 188
procedures, 327
protecting data, 9
Python
  automation objects, 366
  creating Python modules, 243
  creating user-defined functions, 243
  debugging, 250
  displaying submitted SPSS syntax in SPSS output log, 242
  error handling, 225, 247
  file specifications, 215, 241
  handling wide output, 242
  passing information from Python, 224
  passing information to Python, 271
  print statement, 213
  raw strings, 220, 237, 241
  regular expressions, 273, 350, 386
  string substitution, 238
  syntax rules, 220
  triple-quoted strings, 220, 237
  using a Python IDE to drive SPSS, 226
Python functions and classes, 424
  BaseProcedure class, 456
  CreateDatasetOutput, 318
  CreateXMLOutput, 318
  CreateXPathDictionary, 459
  Cursor class, 275, 460, 467
  DeleteXPathHandle, 314, 488
  EndProcedure, 488
  EvaluateXPath, 314, 489
  GetCaseCount, 490
  GetDefaultPlugInVersion, 490
  GetHandleList, 491
  GetLastErrorlevel, 491
  GetLastErrorMessage, 491
  GetSPSSInstallDir, 245
  GetSPSSLowHigh, 492
  GetValuesFromXMLWorkspace, 219, 318
  GetVarAttributeNames, 493
  GetVarAttributes, 493
  GetVariableCount, 254, 494
  GetVariableFormat, 257, 494
  GetVariableLabel, 259, 497
  GetVariableMeasurementLevel, 256, 497
  GetVariableName, 254, 498
  GetVariableType, 260, 498
  GetVarMissingValues, 499
  GetWeightVar, 499
  GetXmlUtf16, 317, 500
  HasCursor, 500
  IsOutputOn, 500
  SetDefaultPlugInVersion, 501
  SetMacroValue, 224, 502
  SetOutput, 502
  ShowInstalledPlugInVersions, 503
  SplitChange, 503
  spssapp class, 364, 366
  Spssdata class, 291
  StartProcedure, 506
  StartSPSS, 510
  StopSPSS, 510
  Submit, 215, 511
  VariableDict class, 263

random distribution functions, 107
random samples
  reproducing with SET SEED, 17

raw strings, 220, 237, 241
reading data, 22
  database tables, 22
  Excel, 29
  SAS format, 59
  Stata format, 61
  text, 35
RECODE (command), 101
  INTO keyword, 102
recoding
  categorical variables, 101
  scale variables, 102
records
  defining wide records with FILE HANDLE, 45
  system variable $casenum, 15
regular expressions, 273, 350, 386
remainder, 106
repeating text data, 58
REPLACE (function), 110
reset method, 476
RINDEX (function), 113
row count, 490
running SPSS commands in Python, 215, 511
RV.NORMAL (function), 108
RV.POISSON (function), 108

SAS
  reading SAS format data, 59
  saving data in SAS format, 182
SAS vs. SPSS
  aggregating data, 400
  arithmetic functions, 410
  banding scale data, 408
  calculating date/time differences, 414
  CALL EXECUTE equivalent, 419
  cleaning and validating data, 404
  dates and times, 414
  extracting date/time parts, 416
  finding duplicate records, 406
  finding invalid values, 404
  %MACRO equivalent, 418
  merging data files, 397
  random number functions, 411
  reading database tables, 392
  reading Excel files, 395
  reading text data files, 397
  recoding categorical data, 407
  statistical functions, 410
  string concatenation, 412
  string parsing, 413
  SYMPUT equivalent, 421
  SYSPARM equivalent, 422
  value labels, 402
  variable labels, 402
SAVE (command)
  PERMISSIONS subcommand, 9
SAVE TRANSLATE (command), 182
SaveDesignatedOutput method, 364
saving
  data in SAS format, 182
  data in Stata format, 183
scale variables, 95
  recoding (binning), 102
scoring, 191
  batch jobs, 208
  command syntax, 204
  exporting data transformations, 197
  mapping variables, 195
  merging data transformations and models, 203
  missing values, 195
scratch variables, 11
SELECT IF (command), 137
  with $casenum, 15
selecting subsets of cases, 137
SET (command)
  SEED subcommand, 17
SetCategories method, 431, 441
SetCell method, 442
SetCellsByColumn method, 432, 444
SetCellsByRow method, 432, 443
SetDefaultFormatSpec method, 342, 446
SetDefaultPlugInVersion, 501
SetFetchVarList method, 477
SetMacroValue, 224, 502
SetOneVarNameAndType method, 477
SetOutput, 502
SetUserMissingInclude method, 478
SetValueChar method, 480
SetValueNumeric method, 480
SetVarAlignment method, 481
SetVarAttributes method, 482
SetVarCMissingValues method, 482

SetVarCValueLabel method, 483
SetVarFormat method, 483
SetVarLabel method, 485
SetVarMeasureLevel method, 485
SetVarNameAndType method, 486
SetVarNMissingValues method, 486
SetVarNValueLabel method, 488
ShowInstalledPlugInVersions, 503
SimplePivotTable method, 335, 427, 447
split-file processing
  creating pivot tables from data with splits, 503
  reading datasets with splits in Python, 280, 297, 475
SplitChange, 503
spo files, 189
spss module, 214
spssapp class, 364, 366
spssaux module
  reading SPSS dictionary information, 262
  reading SPSS output results, 318
Spssdata class, 291
spssdata module, 291
SQL
  reading databases, 25
SQRT (function), 106
square root, 106
StartProcedure, 327, 506
StartSPSS, 510
Stata
  reading Stata data files, 61
  saving data in Stata format, 183
StopSPSS, 510
String class, 453
string substitution, 238
string values
  changing case, 109
  combining, 109
  concatenating, 109
  converting numeric strings to numbers, 110
  converting string dates to date format numeric values, 117
  parsing, 110
  substrings, 110
string variables, 260, 498
Submit, 215, 511
SUBSTR (function), 110
substrings, 110

table lookup file, 69
TEMPORARY (command), 10, 137
temporary transformations, 10
temporary variables, 11
text data
  comma-separated values, 40
  complex text data files, 48
  CSV format, 40
  delimited, 35
  fixed width, 36, 41
  GET DATA vs. DATA LIST, 36
  grouped, 49
  hierarchical, 52
  mixed format, 48
  nested, 52
  reading text data files, 35
  repeating, 58
  wide records, 45
TextBlock class, 327, 512
  append method, 514
TIME.DAYS (function), 120
TIME.HMS (function), 118
times, 114
  computing intervals, 119
  functions, 117
  input and display formats, 115
toNumber method, 454
toString method, 455
transaction files, 74
transformations
  date and time, 114
  numeric, 105
  statistical functions, 106
  string, 108
  using Python functions, 344
transposing cases and variables, 82
triple-quoted strings in Python, 220, 237
TRUNC (function), 107
truncating values, 107

UNIFORM (function), 108
uniform distribution, 108
unknown measurement level, 497

UPCASE (function), 109
UPDATE (command), 74
updating data files, 74
user-missing values, 95
using case weights to replicate crosstabulations, 80

valid cases
  NVALID function, 107
VALIDATEDATA (command), 130
validating data, 123, 130
value labels, 94, 267, 304, 483, 488
  adding, 95
VALUE LABELS (command), 94
ValueLabels method, 267
variable alignment, 481
VARIABLE ATTRIBUTE (command), 96
variable attributes, 304, 482, 493
  retrieving, 271
variable count, 254, 494
variable format, 257, 483, 494
variable label, 259, 485, 497
variable labels, 94
VARIABLE LABELS (command), 94
VARIABLE LEVEL (command), 95
variable names, 254, 498
VariableDict class, 263
variables
  creating with VECTOR command, 143
  making variables from cases, 83
  measurement level, 95
VarName class, 340, 453
VARSTOCASES (command), 86
VarValue class, 340, 454
VECTOR (command), 141
  creating variables, 143
  short form, 144
vectors, 141
  errors caused by disappearing vectors, 143
versions
  managing multiple versions, 229, 490, 501, 503
viewer module, 363
  exporting viewer contents, 364
  modifying pivot tables, 366
  using from a Python IDE, 369
visual binning, 102

WEIGHT (command), 79
weight variable, 499
weighting data, 79–80
wide records
  defining with FILE HANDLE command, 45
Word
  exporting results, 188
WRITE (command), 16

XDATE.DATE (function), 122
XML
  OXML output from OMS, 165
XML workspace, 314
  writing contents to an XML file, 317
XPath expressions, 314, 489
XSAVE (command), 16
  building a data file with LOOP and XSAVE, 150
XSLT
  using with OXML, 165

years
  calculating number of years between dates, 119

zeros
  preserving leading zeros, 110