Writing IBM SPSS Statistics Extension Commands

Note: Before using this information and the product it supports, read the general information under Notices on p. 35.

This edition applies to IBM® SPSS® Statistics 21 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1

Introduction to Extension Commands

Extension commands provide the ability to run functions accessible through the IBM® SPSS® Statistics Programmability Extension using familiar IBM® SPSS® Statistics command syntax. This allows someone who is proficient with the Programmability Extension to share external functions with users of SPSS Statistics command syntax. Extension commands require SPSS Statistics release 16.0.1 or later, although some features (as noted) require a post 16.0.1 release.

This document provides an overview of the requirements for producing and using extension commands.

To produce an extension command

- Install the Integration Plug-in for the programming language in which you want to implement the command.
- Write the XML specification of the syntax diagram for the command based on the extension schema.
- Write the implementation code for the command. Note that you can write implementation code that simply accepts parsed command syntax and passes it to an existing function. For an example of doing this with Python, see Wrapping Around an Existing Python Function: PARETO, on p. 9. For an example of doing this with R, see Wrapping Around an Existing R Function: polychor, on p. 12.
- Write documentation for the extension command, including a SPSS Statistics-style syntax chart for reference.

Optionally, you can:

- Create a custom dialog that generates the command syntax for your extension command. For an introduction to this feature, see the topic on Creating and Deploying Custom Dialogs for Extension Commands in *Programming and Data Management for SPSS Statistics*, available in PDF from the Articles page at *http://www.ibm.com/developerworks/spssdevcentral*. *Note*: The custom dialog feature requires SPSS Statistics release 17.0 or higher.
- Package the XML specification file, implementation code, and any associated custom dialog in an extension bundle so that your extension command can be easily installed by end users. Extension bundles require SPSS Statistics version 18 or higher. For details, see the topic on extension bundles in the SPSS Statistics Help system.

To use an existing extension command

- Install the Integration Plug-in for the programming language in which the extension command is implemented.
- Deploy the implementation code and the XML specification of the syntax for the command.

- Write and run SPSS Statistics command syntax in the Syntax Editor according to the diagram for the extension command, just as you would for any other SPSS Statistics command.

Integration Plug-ins

IBM® SPSS® Statistics Integration Plug-ins allow you to use programming features from other programming languages within SPSS Statistics. Extension commands can be implemented in the following languages once the associated Plug-ins are installed: Python, R and Java (requires SPSS Statistics version 21 or higher).

Information on how to get the Plug-ins for Python and R is available from Core System > Frequently Asked Questions > How to Get Integration Plug-Ins in the SPSS Statistics Help system. The Plug-in for Java (requires SPSS Statistics version 21 or higher) is installed with SPSS Statistics and SPSS Statistics Server and requires no separate installation or configuration.

XML Specification of the Syntax Diagram

The extension schema provides the ability to create an XML representation of the command syntax diagram for an extension command. The XML representation of the syntax diagram for a given extension command describes the syntactic structure for running that command. The IBM® SPSS® Statistics command parser uses this specification to interpret and validate instances of the command. A copy of the extension schema, *extension-1.0.xsd*, is installed with SPSS Statistics. See Extension Schema Element Reference for detailed schema documentation. The following figure shows a basic outline of the extension schema.

Figure 1-1
Outline of the extension schema

```
<Command Name="string" Language="Python" | "R" Mode="Source" | "Package">
 <Subcommand Name="string" Occurrence="Required" | "Optional"
  IsArbitrary="True" | "False" | "Yes" | "No">
  <Parameter Name="string" ParameterType="DatasetName" | "Integer" | "IntegerList" | "Keyword" |
   "KeywordList" | "LeadingToken" | "Number" | "NumberList" | "QuotedString" | "TokenList" |
   "VariableName" | "VariableNameList" | "InputFile" | "OutputFile">
   <EnumValue Name="string"/>
  </Parameter>
 </Subcommand>
</Command>
```

For example, the syntax diagram for the `PLS` extension command, available from *http://www.ibm.com/developerworks/spssdevcentral*, is:

```
PLS dependent variable [MLEVEL={N}] [REFERENCE={FIRST }]
                               {O}             {LAST**}
                               {S}             {value }
    [dependent variable...]
    [BY factor list] [WITH covariate list]

[/ID VARIABLE = variable]

[/MODEL effect [...effect]]
```

```
[/OUTDATASET [CASES=dataset]
             [LATENTFACTORS=dataset]
             [PREDICTORS=dataset]]

[/CRITERIA  LATENTFACTORS={5**     }]
                          {integer}
```

which is represented by the following XML specification:

```
<Command xmlns="http://xml.spss.com/extension"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Name="PLS" Language="Python">
 <Subcommand Name="" Occurrence="Required" IsArbitrary="True"/>
 <Subcommand Name="ID">
  <Parameter Name="VARIABLE" ParameterType="VariableName"/>
 </Subcommand>
 <Subcommand Name="OUTDATASET">
  <Parameter Name="CASES" ParameterType="DatasetName"/>
  <Parameter Name="LATENTFACTORS" ParameterType="DatasetName"/>
  <Parameter Name="PREDICTORS" ParameterType="DatasetName"/>
 </Subcommand>
 <Subcommand Name="CRITERIA">
  <Parameter Name="LATENTFACTORS" ParameterType="Integer"/>
 </Subcommand>
 <Subcommand Name="MODEL" IsArbitrary="True"> </Subcommand>
</Command>
```

- The top-level element, Command, names the command. Subcommands are children of this element. The Name attribute is required and specifies the command name. For release 16.0.1, the name must be a single word in upper case with a maximum of eight bytes. For release 17.0 and higher, the name can consist of up to three words separated by spaces, as in `MY EXTENSION COMMAND`, and is not case sensitive. Command names are restricted to 7-bit ascii characters. The Language attribute is optional and specifies the implementation language. The default is the Python programming language. The choices for Language are *Python* or *R*.
- In this example, the first subcommand has an empty string for a name. Because it doesn't have a name, it is referred to as the **anonymous** subcommand. It is specified to be a required element, and contains the variable list. Its "arbitrary" structure means that elements in the variable list will simply be passed as a tokenlist to the implementation code, and further parsing of the list, including the `MLEVEL`, `REFERENCE`, `BY` and `WITH` (and possibly `TO`) keywords will be left entirely to the implementation code.
- The next subcommand, named ID, is optional by default and has a single child element. Parameters are children of Subcommands. This Parameter element is named `VARIABLE` and is of the type VariableName.
- The next subcommand, `OUTDATASET`, has three child elements. The `CASES`, `LATENTFACTORS`, and `PREDICTORS` Parameter elements are all of type DatasetName.
- The next subcommand, `CRITERIA`, has a single child element. The `LATENTFACTORS` Parameter element is of type Integer.
- The last subcommand, `MODEL`, has no child elements. Its "arbitrary" structure means that the model effects list will simply be passed as a tokenlist to the implementation code, and further parsing of the model effects will be left entirely to the implementation code.

What do all these ParameterTypes mean?

The ParameterType attribute of a Parameter element determines how SPSS Statistics passes the information to the implementation code. For example, when the ParameterType is Integer, the parameter value is passed as an integer number. When ParameterType is Number, the parameter value is passed as a floating point number. See Extension Schema Element Reference for details.

Naming Conventions and Name Conflicts

- Extension commands take priority over built-in command names. For example, if you create an extension command named `MEANS`, the built-in `MEANS` command will be replaced by your extension. Likewise, if an abbreviation is used for a built-in command and the abbreviation matches the name of an extension command, the extension command will be used (abbreviations are not supported for extension commands).
- To reduce the risk of creating extension command names that conflict with built-in commands or commands created by other users, you should use two- or three-word command names, using the first word to specify your organization.
- There are no naming requirements for the file containing the XML specification of the syntax. For example, the XML specification for the `PLS` extension command could be contained in the file *plscommand.xml*. As with choosing the name of the extension command, take care when choosing a name to avoid conflicting XML file names. A useful convention is to use the same name as the Python module, R source file (or package), or Java class file (or JAR file) that implements the command.

Implementation Code

The extension command mechanism requires that the implementation code (whether written in Python, R or Java) reside in a function named `Run`, which is then contained in a Python module, R source code file (requires IBM® SPSS® Statistics release 18 or later), R package, or Java class file (which may be a standalone class file or reside in a JAR file). For general reference when working with Integration Plug-ins, see the Integration-specific documentation that ships with each plug-in. Issues specific to extension commands are discussed below.

Naming Conventions

The Python module, R source file (or package), or Java class file (or JAR file) containing the `Run` function that implements an extension command must adhere to the following naming conventions:

- **Python.** The `Run` function must reside in a Python module file with the same name as the command—for instance, in the Python module file *MYCOMMAND.py* for an extension command named `MYCOMMAND`. The name of the Python module file must be in upper case, although the command name itself is case insensitive. For multi-word command names, replace the spaces between words with underscores. For example, for an extension command with the name `MY COMMAND`, the associated Python module would be *MY_COMMAND.py*.
- **R.** The `Run` function must reside in an R source file or R package with the same name as the command—for instance, in a source file named *MYRFUNC.R* for an extension command named `MYRFUNC`. The name of the R source file or package must be in upper case, although the command name itself is case insensitive. For multi-word command names, replace the

spaces between words with underscores for R source files and periods for R packages. For example, for an extension command with the name `MY RFUNC`, the associated R source file would be named *MY_RFUNC.R*, whereas an R package that implements the command would be named *MY.RFUNC.R*. The source file or package should include any `library` function calls required to load R functions used by the code. *Note*: Use of multi-word command names for R extension commands requires SPSS Statistics release 17.0.1 or later.

- **Java.** The `Run` function must reside in a Java class file or JAR file with the same name as the command—for instance, in a class file named *MYCOMMAND.class* for an extension command named `MYCOMMAND`. The name of the Java class file or JAR file must be in upper case, although the command name itself is case insensitive. For multi-word command names, spaces between words should be replaced with underscores when constructing the name of the Java class file or JAR file. For example, for an extension command with the name `MY COMMAND`, the associated Java class file would be *MY_COMMAND.class*. For more information on creating extension commands implemented in Java, see Integration Plug-in for Java User Guide > Getting Started with the Integration Plug-in for Java in the SPSS Statistics Help system.

Input from IBM SPSS Statistics

SPSS Statistics parses syntax for the extension command according to the XML representation of the syntax diagram, then passes the parameters to the implementation code within a `BEGIN PROGRAM–END PROGRAM` block of command syntax. Continuing with the example of the `PLS` extension command (implemented in Python), when you run the following `PLS` command syntax

```
PLS lnsales MLEVEL=S BY type WITH price engine_s horsepow wheelbas width
    length curb_wgt fuel_cap mpg
  /CRITERIA LATENTFACTORS=5
  /OUTDATASET CASES=indvCases LATENTFACTORS=latentFactors
    PREDICTORS=indepVars.
```

it produces and runs, in the background, the following program block (formatted here for readability)

```
BEGIN PROGRAM.
import spss
import PLS
PLS.Run({'PLS': {
  '': [{'TOKENLIST': ['lnsales', 'MLEVEL', '=', 'S', 'BY', 'type',
                      'WITH', 'price', 'engine_s', 'horsepow', 'wheelbas',
                      'width', 'length', 'curb_wgt', 'fuel_cap', 'mpg']}],
  'OUTDATASET': [{'CASES': ['INDVCASES']},
                 {'LATENTFACTORS': ['LATENTFACTORS']},
                 {'PREDICTORS': ['INDEPVARS']}],
  'CRITERIA': [{'LATENTFACTORS': [5]}]}})
END PROGRAM.
```

The program block attempts to import the `PLS` module (i.e. a Python module with the same name as the extension command) and call the module's `Run` function, passing the parsed command syntax as the single argument. As this example illustrates, the argument passed to the `Run` function has a nontrivial structure. The `extension` module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-in for Python, greatly simplifies the task of argument

parsing for extension commands implemented in Python (see Wrapping Around an Existing Python Function: PARETO on p. 9 for an example of the approach).

In lieu of using the `extension` module, you will need to explicitly parse the argument. The following is a brief summary of the structure for the present example.

- The argument passed to the `Run` function is a complex dictionary structure whose top-level key is the command name—in this case, `PLS`. The next level of nested items corresponds to the subcommands of the `PLS` command.
- The first nested item is the string `''`, which is keyed to a list containing a dictionary that has the single item `TOKENLIST`, which in turn is keyed to a list containing the variable list. From this, the implementation code needs to be able to come away with the information that the scale variable *lnsales* is the sole dependent, that *type* is a factor, and that *price*, *engine_s*, *horsepow*, *wheelbas*, *width*, *length*, *curb_wgt*, *fuel_cap*, and *mpg* are covariates.
- The next nested item, `OUTDATASET`, is keyed to a list containing three dictionaries, each of which contains a single item (corresponding to a keyword of the `OUTDATASET` subcommand) keyed to a list containing a single string (corresponding to the value assigned to the keyword in the `PLS` syntax above). The implementation code needs to be able to determine that output variables related to individual cases, latent factors, and predictors be saved to new datasets *indvCases*, *latentFactors*, and *indepVars*, respectively. These selections will also produce plots of latent factor scores, latent factor weights, and variable importance to projection (VIP) by latent factor.
- The last nested item under `PLS`, `CRITERIA`, is keyed to a list containing a dictionary that contains a single item (corresponding to the `LATENTFACTORS` keyword of the `CRITERIA` subcommand) keyed to a list containing a single integer (corresponding to the value assigned to the keyword in the PLS syntax above). The implementation code needs to determine that a solution with 5 latent factors should be produced.

For more information, see the topic Examples of Extension Commands in Chapter 2 on p. 9.

Notes

Command syntax errors. Syntax errors—for example, not providing an integer for a parameter specified as Integer—are handled by SPSS Statistics and stop the module from running, so the implementation code does not need to handle deviations from the XML syntax diagram.

Generating output. Generating and sending output to SPSS Statistics is handled by the implementation code. See the Integration-specific documentation that ships with each plug-in.

- For Python, the implementation code is responsible for generating the procedure name (associated with the extension command) that labels the output in the Viewer. In other words, unlike built-in SPSS Statistics procedures such as `FREQUENCIES`, there is no automatic association of the extension command name with the name that labels output from the command. Specifically, the procedure name is the argument to the `StartProcedure` function.
- For R, the default name associated with output from an extension command is *R*. For SPSS Statistics release 18 or later, the name can be customized. For more information, see the topic R Source File in Chapter 2 on p. 14.

Localization. You can localize messages and output produced by the implementation code. Details on how to do this for Python are provided in Appendix A. For information on localizing extension commands implemented in R, see the documentation for the IBM® SPSS® Statistics - Integration Plug-in for R, available from the Help system, once the Plug-in has been installed.

Deploying an Extension Command

Using an extension command requires that IBM® SPSS® Statistics can access both the XML syntax specification file and the implementation code (Python module, R source file, R package, Java class file or JAR file). If the extension command is distributed in an extension bundle (.spe) file, then you can simply install the bundle from Utilities>Extension Bundles>Install Extension Bundle within SPSS Statistics (extension bundles require SPSS Statistics version 18 or higher). Otherwise, you will need to manually install the XML syntax specification file and the implementation code. Both should be placed in the *extensions* directory, located at the root of the SPSS Statistics installation directory. For Mac, the installation directory refers to the *Contents* directory in the SPSS Statistics application bundle.

Note: For version 18 on Mac, the files can also be placed in */Library/Application Support/SPSSInc/PASWStatistics/18/extensions*. For version 19 and higher on Mac, the files can also be placed in */Library/Application Support/IBM/SPSS/Statistics/<version>/extensions*, where <version> is the two digit SPSS Statistics version—for example, 21.

- For Windows and UNIX, for release 21 and higher, if you do not have write permissions to the SPSS Statistics installation directory then you can unzip the contents to the following general user-writable locations:

  Windows 7 and Windows Vista. Unzip the contents to *C:\Users\<user>\AppData\Local\IBM\SPSS\Statistics\<version>\extensions* where <user> is the user name and <version> is the two digit SPSS Statistics version—for example, 21. Note that you may need to create the directories in the specified path.

  Windows XP. Unzip the contents to *C:\Documents and Settings\<user>\Local Settings\Application Data\IBM\SPSS\Statistics\<version>\extensions* where <user> is the user name and <version> is the two digit SPSS Statistics version—for example, 21. Note that you may need to create the directories in the specified path.

  UNIX (including Linux). Unzip the contents to *~/.IBM/SPSS/Statistics/<version>/extensions* where <version> is the two digit SPSS Statistics version—for example, 21. Note that you may need to create the directories in the specified path.

- For Windows, UNIX and Mac, and for release 18 and higher, if you do not have write permissions to the SPSS Statistics installation directory or would like to store the XML file and the implementation code elsewhere, you can specify one or more alternate locations by defining the *SPSS_EXTENSIONS_PATH* environment variable. For multiple locations, separate each with a semicolon on Windows and a colon on UNIX and Mac when specifying the environment variable. When present, the paths specified in *SPSS_EXTENSIONS_PATH* take precedence over the SPSS Statistics installation directory. The *extensions* subdirectory of the installation directory is always searched after any locations specified in the environment variable, followed by the application data directories described above.

- For an extension command implemented in Python, you can always store the associated Python module to a location on the Python search path (such as the Python *site-packages* directory), independent of where you store the XML specification file. The *extensions* subdirectory and any other directories specified in *SPSS_EXTENSIONS_PATH* are automatically added to the Python search path when SPSS Statistics starts.
- For an extension command implemented in R, the R source file or R package containing the implementation code should be installed to the directory containing the XML syntax specification file. R packages can alternatively be installed to the default location for the associated platform—for example, *R_Home/library* on Windows, where *R_Home* is the installation location of R and *library* is a subdirectory under that location. For help with installing R packages, consult the *R Installation and Administration* guide, distributed with R.

At startup, SPSS Statistics reads the *extensions* directory and any directories specified in *SPSS_EXTENSIONS_PATH*, and registers the extension commands found in those locations. If you want to load a new extension command without restarting SPSS Statistics you will need to use the `EXTENSION` command (see the SPSS Statistics Help system or the *Command Syntax Reference* for more information).

Note: If you or your end users will be running an extension command while in distributed mode, be sure that the extension command files (XML specification and implementation code) and the relevant SPSS Statistics Integration Plug-In(s) (Python and/or R) are installed to both the client and server machines.

Enabling Color Coding and Auto-Completion in the Syntax Editor

The XML syntax specification file contains all of the information needed to provide color coding and auto-completion for your extension command in the Syntax Editor. For SPSS Statistics release 18 and later these features are automatically enabled. To enable these features for release 17, place a copy of the XML file in the *syntax_xml* directory—located at the root of the SPSS Statistics installation directory for Windows, and under the *bin* subdirectory of the installation directory for Linux and Mac. The contents of the *syntax_xml* directory are read when SPSS Statistics starts up.

Chapter 2

Examples of Extension Commands

This section contains examples of extension commands implemented in Python and R. More examples of extension commands can be found on SPSS community. Additional information on extension commands can also be found in *Programming and Data Management for SPSS Statistics* available in PDF from the Articles page at *http://www.ibm.com/developerworks/spssdevcentral*. A tutorial on creating an extension command implemented in R is available from Help>Working with R, within IBM® SPSS® Statistics, for version 18 and higher, and once the IBM® SPSS® Statistics - Integration Plug-in for R is installed.

Wrapping Around an Existing Python Function: PARETO

The *paretochart.py* file, found at *http://www.ibm.com/developerworks/spssdevcentral*, contains code for producing a Pareto chart with extra features beyond those in the standard IBM® SPSS® Statistics Pareto chart. However, it requires the use of Python calls within `BEGIN PROGRAM–END PROGRAM` syntax. In order to provide this functionality to users of SPSS Statistics syntax, you can create an extension command that accepts parsed syntax and passes it to *paretochart.py*.

Consider the `chart` function in *paretochart.py*:

```
chart(varname, title=None, closeds=True, alpha=.05, totalcategories=None,
      template=None)
```

To start, let's create the simplest possible extension command syntax — one that simply takes the name of the variable for which the Pareto chart is created. The SPSS Statistics syntax diagram could look like:

```
PARETO VARIABLE=variable-name.
```

The corresponding XML syntax specification, based on the extension schema, would then be:

```
<Command xmlns="http://xml.spss.com/extension"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Name="PARETO">
 <Subcommand Name="" Occurrence="Required">
  <Parameter Name="VARIABLE" ParameterType="VariableName"/>
 </Subcommand>
</Command>
```

- The Command element names the command `PARETO`.
- The first Subcommand element has an empty string for a name. It is specified to be a required element, and contains a single child element. The `VARIABLE` Parameter element is of type VariableName.

You would then save this specification to an XML file—for example, *PARETO.xml*—in the */extensions* directory.

The code for the Python module—which must be named PARETO—that implements the command is:

```
"""Pareto Charts Extension Module"""

from paretochart import chart, ParetoAcceptance
from extension import Syntax, Template, processcmd

def Run(args):
  """Execute the PARETO command"""

  synObj = Syntax([
    Template(kwd="VARIABLE", subc="", var="varname", ktype="existingvarlist")])
  processcmd(synObj, args[args.keys()[0]], chart)
```

- The `from...import` statements load modules that will do most of the work for us. Functionality in `paretochart` creates the Pareto chart. The `extension` module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-in for Python, contains functionality to parse the arguments passed to the `Run` function and create the argument list to be sent to the function being wrapped—in this case, the `chart` function. It also allows you to specify more validation than is possible in the XML specification of the extension command. Particularly important is the ability to specify that a variable referenced in the submitted syntax must already exist in the active dataset.
- The `Template` class from the `extension` module is used to specify a keyword. Each keyword of each subcommand should have an associated instance of the `Template` class. In this example, `VARIABLE` is the only keyword and it belongs to the anonymous subcommand.

  The argument *kwd* to the `Template` class specifies the name of the keyword.

  The argument *subc* to the `Template` class specifies the name of the subcommand that contains the keyword. If the keyword belongs to the anonymous subcommand, the argument *subc* can be omitted or set to the empty string as shown here.

  The argument *var* specifies the name of the Python variable that receives the value specified for the keyword. In this case the Python variable *varname* will contain the value specified for the `VARIABLE` keyword. If *var* is omitted, the lowercase value of *kwd* is used.

  The argument *ktype* specifies the type of keyword, such as whether the keyword specifies a variable name, a string, or a floating point number. In this example, the keyword defines a variable name, representing a variable that must exist in the active dataset, and is specified as the type existingvarlist.
- The `Syntax` class from the `extension` module validates the syntax specified by the `Template` objects. You instantiate the `Syntax` class with a sequence of one or more `Template` objects. In this example, there is only one `Template` object so the argument to the `Syntax` class is a list with a single element.
- The `processcmd` function from the `extension` module parses the values passed to the `Run` function and executes the implementation function.

  The first argument to the `processcmd` function is the Syntax object for the command, created from the `Syntax` class.

The value of the second argument must be the top-level item in the dictionary passed to the `Run` function. This is given by the expression `args[args.keys()[0]]`. It could alternatively be written `args["PARETO"]`.

The third argument to `processcmd` is the name of the implementation function—in this case, the `chart` function, from the `paretochart` module, that produces the Pareto chart. The values of the keywords specified by the `Template` objects are passed to the implementation function as a set of keyword arguments. In the present example, the `chart` function will be called with the following signature: `chart(varname=<value of VARIABLE keyword>)`.

Note: If a Python exception is raised in the implementation function, the Python traceback is suppressed, but the error message is displayed.

You can obtain additional help for the `extension` module by including the statement `help(extension)` in a program block, once you've imported the module, or by reading the detailed comments in the module.

Expanding the Syntax Diagram

With a simple extension and implementation code under your belt, now consider expanding the options in the `PARETO` command. For example, the following chart adds the ability to specify a title, a chart template, and alpha level for confidence intervals:

```
PARETO VARIABLE=variable-name
  /GRAPHSPEC TITLE='title-string' TEMPLATE='filename'
  /CRITERIA ALPHA=value.
```

The corresponding XML syntax specification based on the extension schema, would then be:

```
<Command xmlns="http://xml.spss.com/extension"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Name="PARETO">
 <Subcommand Name="" Occurrence="Required">
  <Parameter Name="VARIABLE" ParameterType="VariableName"/>
 </Subcommand>
 <Subcommand Name="GRAPHSPEC">
  <Parameter Name="TITLE" ParameterType="QuotedString"/>
  <Parameter Name="TEMPLATE" ParameterType="InputFile"/>
 </Subcommand>
 <Subcommand Name="CRITERIA">
  <Parameter Name="ALPHA" ParameterType="Number"/>
 </Subcommand>
</Command>
```

The code for the Python module that implements the command is:

```
"""Pareto Charts Extension Module"""

from paretochart import chart, ParetoAcceptance
from extension import Syntax, Template, processcmd

def Run(args):
```

```
    """Execute the PARETO command"""

    synObj = Syntax([
      Template(kwd="VARIABLE", subc="", var="varname", ktype="existingvarlist"),
      Template("TITLE", subc="GRAPHSPEC", ktype="literal"),
      Template("TEMPLATE", subc="GRAPHSPEC", ktype="literal"),
      Template("ALPHA", subc="CRITERIA", ktype="float", vallist=[0,1])])
    processcmd(synObj, args[args.keys()[0]], chart)
```

There is a `Template` object for each of the new keywords in the syntax diagram.

- The `TITLE` and `TEMPLATE` keywords on the `GRAPHSPEC` subcommand are of `literal` type, since you want them to retain any capitalization; the `str` keyword type converts letters to lower case.
- The `ALPHA` keyword on the `CRITERIA` subcommand is a `float`; moreover, it takes values in the range [0,1].

Wrapping Around an Existing R Function: polychor

Using the extension command mechanism you can wrap any R function. As an example, we'll wrap the `polychor` function from the `polycor` package (available from any CRAN mirror site) in an extension command named `RPOLYCHOR`. In its simplest usage, the function computes the correlation between two ordinal variables. The function has the following signature:

```
polychor(x,y,ML=FALSE,control=list(),std.err=FALSE,maxcor=.9999)
```

To simplify the associated syntax, we'll omit all parameters other than the two variables *x* and *y* and the *maxcor* parameter and consider the case where *x* and *y* are numeric variables. The IBM® SPSS® Statistics syntax diagram could look like:

```
RPOLYCHOR VARIABLES=varlist
   /OPTIONS MAXCOR = {.9999**}
                     {value  }
```

The corresponding XML specification, based on the extension schema, would then be:

```
<Command xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="extension.xsd" Name="RPOLYCHOR" Language="R">
    <Subcommand Name="" IsArbitrary="False" Occurrence="Required">
        <Parameter Name="VARIABLES" ParameterType="VariableNameList"/>
    </Subcommand>
    <Subcommand Name="OPTIONS">
        <Parameter Name="MAXCOR" ParameterType="Number"/>
    </Subcommand>
</Command>
```

- The `Command` element names the command `RPOLYCHOR`. The `Language` attribute specifies R as the implementation language.

- The `VARIABLES` keyword, associated with the anonymous subcommand, is used to specify the input variables. It has a parameter type of `VariableNameList`. Values specified for `VariableNameList` parameters are checked to be sure they represent syntactically valid SPSS Statistics variable names (the existence of the variables is not checked).
- The `OPTIONS` `Subcommand` element contains a `Parameter` element for the value of *maxcor*. The parameter type is specified as `Number`, which means that the value can be a number, possibly in scientific notation using e or E.

You would then save this specification to an XML file—for example, *RPOLYCHOR.xml*—in the */extensions* directory.

Implementation Code

When wrapping an R function in an extension command, different architectures for the implementation code are available and depend on your version of IBM® SPSS® Statistics. For SPSS Statistics version 18 and higher it is recommended to use the R source file approach.

- **R source file.** The implementation code is contained in an R source file. This approach requires that you and your end users have R and the IBM® SPSS® Statistics - Integration Plug-in for R installed on machines that will run the extension command, and is only available for SPSS Statistics version 18 and higher. This is by far the simplest approach and is the recommended method for users who have SPSS Statistics version 18 or higher. An example of this approach is described in R Source File on p. 14.
- **Wrapping in python.** You can wrap the code that implements the R function in Python. This is the recommended approach for users who do not have SPSS Statistics version 18 or higher or who need to customize the output with Python scripts. This approach requires that you and your end users have R, Python, the Integration Plug-in for R, and the IBM® SPSS® Statistics - Integration Plug-in for Python installed on machines that will run the extension command. *Note*: Full support for this approach requires SPSS Statistics version 17.0.1 or higher.
- **R package.** The implementation code is contained in an R package. This approach is the most involved because it requires creating and installing R packages, but it allows you to potentially distribute your package through the Comprehensive R Archive Network (CRAN). This approach requires that you and your end users have R and the Integration Plug-in for R installed on machines that will run the extension command. For users with SPSS Statistics version 18 or higher, the approach for creating the implementation code is the same as for the R source file approach but requires the further step of creating an R package containing the implementation code. If you are interested in this approach but are not familiar with creating R packages, you may consider creating a skeleton package using the R `package.skeleton` function (distributed with R). If you and any end users do not have SPSS Statistics version 18 or higher, then you will have to manually parse the argument passed to the `Run` function. For more information, see the topic Manually Parsing on p. 18.

It is also possible to generate an R program directly from a custom dialog, bypassing the extension method entirely. However, the entire program will then appear in the log file, and extra care must be taken with long lines of code. For an example of this approach, see the Rboxplot example, available from the SPSS community.

R Source File

To wrap an R function, you create an R source file containing a `Run` function that parses and validates the syntax specified by the end user, and another function—called by `Run`—that actually implements the command. Following the example of the `RPOLYCHOR` extension command, the associated Run function is:

```
Run<-function(args){
    args <- args[[2]]
    oobj<-spsspkg.Syntax(templ=list(
             spsspkg.Template(kwd="VARIABLES",subc="",ktype="existingvarlist",
                              var="vars",islist=TRUE),
             spsspkg.Template(kwd="MAXCOR",subc="OPTIONS",ktype="float",var="maxcor")
             ))
    res <- spsspkg.processcmd(oobj,args,"rpolycor")
}
```

- IBM® SPSS® Statistics parses the command syntax entered by the user and passes the specified values to the `Run` function in a single argument—*args* in this example. The argument is a list structure whose first element is the command name and whose second element is a set of nested lists containing the values specified by the user.
- The `Run` function contains calls to the `spsspkg.Syntax`, `spsspkg.Template`, and `spsspkg.processcmd` functions, which are designed to work together.

  `spsspkg.Template` specifies the details needed to process a specified keyword in the syntax for an extension command.

  `spsspkg.Syntax` validates the values passed to the `Run` function according to the templates specified for the keywords.

  `spsspkg.processcmd` parses the values passed to the `Run` function and calls the function that will actually implement the command—in this example, the function *rpolycor* (discussed below) which resides in the same source file as the `Run` function.

  Note: Complete documentation for these functions is available from the SPSS Statistics Help system.
- You call `spsspkg.Template` once for each keyword supported by the extension command. In this example, the extension command contains the two keywords `VARIABLES` and `MAXCOR`, so `spsspkg.Template` is called twice. The function returns an object, that we'll refer to as a template object, for use with the `spsspkg.Syntax` function.
- The argument *kwd* to `spsspkg.Template` specifies the name of the keyword (in uppercase) for which the template is being defined.
- The argument *subc* to `spsspkg.Template` specifies the name of the subcommand (in uppercase) that contains the keyword. If the keyword belongs to the anonymous subcommand, the argument *subc* can be omitted or set to the empty string as shown here.
- The argument *ktype* to `spsspkg.Template` specifies the type of keyword, such as whether the keyword specifies a variable name, a string, or a floating point number.
- The value *existingvarlist* for *ktype* specifies a list of variable names that are validated against the variables in the active dataset. It is used for the `VARIABLES` keyword that specifies the variables for the analysis.
- The value *float* for *ktype* specifies a real number. It is used for the `MAXCOR` keyword.

- The argument *var* to `spsspkg.Template` specifies the name of an R variable that will be set to the value specified for the keyword. This variable will be passed to the implementation function by the `spsspkg.processcmd` function.
- The optional argument *islist* to `spsspkg.Template` is a boolean (*TRUE* or *FALSE*) specifying whether the keyword contains a list of values. The default is *FALSE*. The keyword `VARIABLES` in this example is a list of variable names, so it should be specified with *islist=TRUE*.
- Once you have specified the templates for each of the keywords, you call `spsspkg.Syntax` with a list of the associated template objects. The returned value from `spsspkg.Syntax` is passed to the `spsspkg.processcmd` function, which has the following required arguments:

  The first argument is the value returned from the `spsspkg.Syntax` function.

  The second argument is the list structure containing the values specified by the user in the submitted syntax.

  The third argument is the name of the function that will actually implement the extension command—in this example, the function `rpolycor`.

```
rpolycor<- function(vars,maxcor=0.9999){
library(polycor)
x <- vars[[1]]
y <- vars[[2]]

# Get the data from the active dataset and run the analysis
data <- spssdata.GetDataFromSPSS(variables=c(x,y),missingValueToNA=TRUE)
result <- polychor(data[[x]],data[[y]],maxcor=maxcor)

# Create output to display the result in the Viewer
spsspkg.StartProcedure("Polychoric Correlation")
spsspivottable.Display(result,title="Correlation",
                       rowlabels=c(x),
                       collabels=c(y),
                       format=formatSpec.Correlation)
spsspkg.EndProcedure()
}
```

- The `spsspkg.processcmd` function calls the specified implementation function—in this case, *rpolycor*—with a set of named arguments, one for each template object. The names of the arguments are the values of the *var* arguments specified for the associated template objects and the values of the arguments are the values of the associated keywords from the syntax specified by the end user.
- The implementation function contains the `library` statement for the R `polycor` package.
- The `spssdata.GetDataFromSPSS` function reads the case data for the two specified variables from the active dataset. `missingValueToNA=TRUE` specifies that missing values of numeric variables are converted to the R *NA* value (by default, they are converted to the R *NaN* value). The data are then passed to the R `polychor` function to compute the correlation.
- You group output under a common heading using a `spsspkg.StartProcedure-spsspkg.EndProcedure` block. The argument to `spsspkg.StartProcedure` specifies the name that appears in the outline pane of the Viewer associated with the output.

- The `spsspivottable.Display` function creates a pivot table that is displayed in the SPSS Statistics Viewer. The first argument is the data to be displayed as a pivot table. It can be a data frame, matrix, table, or any R object that can be converted to a data frame.

Notes

- To use the R source file method, you will need to specify `Language="R"` in the `Command` element of the XML specification file.
- Use of the argument `missingValueToNA` to `spssdata.GetDataFromSPSS` requires SPSS Statistics version 18 or higher. For earlier versions, use the statement `is.na(data)<-is.na(data)` following the call to `spssdata.GetDataFromSPSS` to convert missing values of numeric variables to the R *NA* value, as in:

```
data <- spssdata.GetDataFromSPSS()

is.na(data)<-is.na(data)
```

- Use of an `spsspkg.StartProcedure-spsspkg.EndProcedure` block requires SPSS Statistics version 18 or higher.

Wrapping in Python

To wrap the implementation code for an R function in Python, you create a Python function that generates the necessary R code and submits it to IBM® SPSS® Statistics in a `BEGIN PROGRAM R–END PROGRAM` block. Following the example of the `RPOLYCHOR` extension command, the code for the Python module—which must be named `RPOLYCHOR`—that implements the command, including all necessary `import` statements, is:

```
import spss, spssaux
from extension import Template, Syntax, processcmd

def Run(args):

    args = args[args.keys()[0]]

    oobj = Syntax([
      Template("VARIABLES",subc="",ktype="existingvarlist",var="vars",islist=True),
      Template("MAXCOR",subc="OPTIONS",ktype="float",var="maxcor")])

    processcmd(oobj, args, rpolychor, vardict=spssaux.VariableDict())

def rpolychor(vars, maxcor=0.9999):

        varX = vars[0]
        varY = vars[1]
        pgm = r"""BEGIN PROGRAM R.
library(polycor)
data <- spssdata.GetDataFromSPSS(variables=c("%(varX)s","%(varY)s"),missingValueToNA=TRUE)
result <- polychor(data[["%(varX)s"]],data[["%(varY)s"]],maxcor=%(maxcor)s)

# Create output to display the result in the Viewer
spsspkg.StartProcedure("Polychoric Correlation")
spsspivottable.Display(result,title="Correlation",
                       rowlabels=c("%(varX)s"),
                       collabels=c("%(varY)s"),
                       format=formatSpec.Correlation)
spsspkg.EndProcedure()
END PROGRAM.
""" % locals()

        spss.Submit(pgm)
```

- The module consists of the `Run` function that parses the values passed from SPSS Statistics and a Python function named `rpolychor` that generates the `BEGIN PROGRAM R–END PROGRAM` block that calls the R `polychor` function.
- The `Run` function uses the Python `extension` module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-in for Python, to parse the arguments passed from SPSS Statistics and to pass those arguments to the `rpolychor` function. For more information, see the topic Wrapping Around an Existing Python Function: PARETO on p. 9.
- The `rpolychor` function generates a `BEGIN PROGRAM R–END PROGRAM` block of command syntax containing all of the R code needed to get the data from SPSS Statistics, call the R `polychor` function, and display the results in a pivot table in the SPSS Statistics Viewer. The block is submitted to SPSS Statistics with the `spss.Submit` function.

As an alternative to directly submitting the `BEGIN PROGRAM R–END PROGRAM` block, you can write the block to an external file and use the `INSERT` command to run the syntax. Writing the block to an external file has the benefit of saving the generated R code for future use. The following code—which replaces `spss.Submit(pgm)` in the previous example—shows how to do this in the case that the block is written to a file in the directory currently designated as the temporary directory. To use this code, you will also have to import the `tempfile`, `os`, and `codecs` modules.

```
cmdfile = (tempfile.gettempdir() + os.sep + "pgm.R").replace("\\", "/")
f = codecs.open(cmdfile, "wb", encoding="utf_8_sig")
f.write(pgm)
f.close()
spss.Submit("INSERT FILE='%s'" % cmdfile)
```

- Setting `encoding="utf_8_sig"` means that the file is written in UTF-8 with a byte order mark (BOM). This ensures that SPSS Statistics will properly handle any extended ascii characters in the file when the file is read with the `INSERT` command.

As a useful convention, you may want to consider adding a `SAVE` subcommand with a `PROGRAMFILE` keyword to your extension command to let the user decide whether to save the generated R code. For an example of this approach, see the `SPSSINC HETCOR` command, installed with IBM® SPSS® Statistics - Essentials for R.

Notes

- In the specification of the `BEGIN PROGRAM R` block, use of the argument `missingValueToNA` to `spssdata.GetDataFromSPSS` requires SPSS Statistics version 18 or higher. For earlier versions, use the statement `is.na(data)<-is.na(data)` following the call to `spssdata.GetDataFromSPSS` to convert missing values of numeric variables to the R *NA* value, as in:

  ```
  data <- spssdata.GetDataFromSPSS()

  is.na(data)<-is.na(data)
  ```

- In the specification of the `BEGIN PROGRAM R` block, use of an `spsspkg.StartProcedure-spsspkg.EndProcedure` block requires SPSS Statistics version 18 or higher.

Manually Parsing

In the case that you are using the R package approach and you do not have IBM® SPSS® Statistics version 18 or higher, you will need to manually parse the argument passed to the `Run` function. Following the example of the `RPOLYCHOR` extension command, consider the following command syntax entered by a user:

```
RPOLYCHOR VARIABLES=var1 var2 /OPTIONS MAXCOR=.999.
```

The argument passed to the R `Run` function is a set of nested lists. When rendered with the R `print` command from within the `Run` function, the structure is as follows:

```
[[1]]
[1] "RPOLYCHOR"

[[2]]
[[2]]$` `
[[2]]$` `[[1]]
[[2]]$` `[[1]]$VARIABLES
[[2]]$` `[[1]]$VARIABLES[[1]]
[1] "var1"

[[2]]$` `[[1]]$VARIABLES[[2]]
[1] "var2"

[[2]]$OPTIONS
[[2]]$OPTIONS[[1]]
[[2]]$OPTIONS[[1]]$MAXCOR
[[2]]$OPTIONS[[1]]$MAXCOR[[1]]
[1] 0.999
```

In deciphering this structure it may be useful to notice that it can be generated from the following R command:

```
list("RPOLYCHOR",
     list(' '=list(list(VARIABLES=list("var1","var2"))),
          OPTIONS=list(list(MAXCOR=list(0.999)))
         )
    )
```

- The argument passed to the `Run` function is a list whose first element is the command name and whose second element is a list containing all of the specifications provided by the user in the submitted command syntax.
- Each subcommand is represented as an element of the inner list, as shown for the anonymous subcommand (represented by the name `' '`) and the `OPTIONS` subcommand. The parameter specifications for each subcommand are contained in a list structure.

The R code for a `Run` function that implements the `RPOLYCHOR` command, including the `library` statement for the `polycor` package, is:

```
.packageName <- "RPOLYCHOR"
Run <- function(args){
library(polycor)

# Parse the arguments
maxcor_parm <- args[[2]]$OPTIONS[[1]]$MAXCOR[[1]]
x <- args[[2]]$' '[[1]]$VARIABLES[[1]]
y <- args[[2]]$' '[[1]]$VARIABLES[[2]]

# Set default value of maxcor
maxcor <- 0.9999

if (!is.null(maxcor_parm)) maxcor <- as.double(maxcor_parm)

# Get the data from the active dataset and run the analysis
data <- spssdata.GetDataFromSPSS(variables=c(x,y))
result <- polychor(data[[x]],data[[y]],maxcor=maxcor)

# Create output to display the result in the Viewer
spsspivottable.Display(result,title="Polychoric Correlation",
                       rowlabels=c(x),
                       collabels=c(y),
                       format=formatSpec.Correlation)
}
```

- The statements that parse the arguments pick out the needed information (the names of the variables and the optional value of *maxcor*) using the known structure of the set of nested lists.
- The `spssdata.GetDataFromSPSS` function reads the case data for the two specified variables from the active dataset.
- The `spsspivottable.Display` function creates a pivot table that is displayed in the SPSS Statistics Viewer.

Chapter
3

Extension Schema Element Reference

This section provides a reference for all elements in the extension schema. Each topic lists the valid attributes for an element and its parent and child elements.

Command Element

The top-level element, also known as the document or root element. This element contains a complete command syntax specification of an extension command.

Table 3-1
Attributes for Command

Attribute	Use	Description	Valid Values
Language	optional	The programming language in which the command is implemented. Defaults to Python.	**Python** **R** **Java**
Mode	optional	Specifies whether the implementation code is contained in an R source file or an R package. Only applies for Language="R". Defaults to Source.	**Source** **Package**
Name	**required**	The name of the command. For release 16.0.1, the name must be a single word in upper case with a maximum of 8 bytes. For release 17.0 and higher, the name can consist of up to three words (case insensitive) separated by spaces. Command names are restricted to 7-bit ascii characters. For multi-word command names, ensure that the first word as well as the first two words do not match the name of another command. For example, do not use the name CMD NEW if there is a command named CMD. Likewise, do not use the name MY CMD NEW if there is a command named MY CMD.	*any*

XML Representation

```
<xs:element name="Command" type="command-content">
    <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="Subcommand"></xs:element>
    </xs:sequence>
    <xs:attribute name="Name" use="required"></xs:attribute>
    <xs:attribute name="Language">
        <xs:simpleType>
            <xs:restriction base="xs:token">
                <xs:enumeration value="Python"></xs:enumeration>
                <xs:enumeration value="R"></xs:enumeration>
                <xs:enumeration value="Java"></xs:enumeration>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="Mode">
        <xs:simpleType>
            <xs:restriction base="xs:token">
                <xs:enumeration value="Source"></xs:enumeration>
                <xs:enumeration value="Package"></xs:enumeration>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:element>
```

Child Elements

Subcommand Element

Parameter Element

A parameter controls a specific piece of a command's functionality. There are many types of parameters so that the Parameter element is flexible enough to cover the different types of functionality provided by parameters. Subcommands contain zero or more parameters.

Table 3-2
Attributes for Parameter

Attribute	Use	Description	Valid Values
Name	**required**	The name of the parameter. Parameter names are restricted to 7-bit ascii characters and must start with a letter or one of the characters @, #, or $. Subsequent characters can be any combination of letters, numbers, nonpunctuation characters, and a period (.).	*any*
ParameterType	**required**		**DatasetName**. Specifies a dataset name. The value will be checked for syntax correctness (same

Attribute	Use	Description	Valid Values
			rules as for variable names) but not existence. The case is preserved when passed to the Run function. **Integer**. A number with no fractional part after conversion. Optionally, you can specify a set of allowed keyword values for an Integer parameter, using EnumValue elements. **IntegerList**. A blank or comma separated list of Integer types. **Keyword**. Specifies a value that adheres to the same rules as the Name attribute of a Parameter element. The value is passed in upper case to the Run function. You can specify the set of allowed values using EnumValue elements. Keyword type parameters must be assigned values. To specify a keyword without an associated value, use the LeadingToken type. **KeywordList**. Specifies a comma or blank separated list of values that adhere to the same rules as the Name attribute of a Parameter element. To specify a list of values not bound by these rules, use the TokenList type. You can specify the set of allowed values using EnumValue elements. **LeadingToken**. Specifies a parameter that has a name (given by the Name attribute of the Parameter element) but no associated value. The name is passed in upper case to the Run function.

Attribute	Use	Description	Valid Values
			Number. A number, possibly in scientific notation using e or E. Optionally, you can specify a set of allowed keyword values for a Number parameter, using EnumValue elements. **NumberList**. A blank or comma separated list of Number types. **QuotedString**. A string enclosed in single or double quotes. The case is preserved when passed to the Run function. Optionally, you can specify a set of allowed keyword values (unquoted) for a QuotedString parameter, using EnumValue elements. **TokenList**. Specifies a comma or blank separated list of values. Case is preserved when values are passed to the Run function. The TokenList type is similar to the KeywordList type but TokenList values are not bound by the rules required of KeywordList values. **VariableName**. Specifies a variable name. The value will be checked for syntax correctness (see the rules for variable names in the Command Syntax Reference) but not existence. The case is preserved when passed to the Run function. **VariableNameList**. Specifies a list of variable names. Each name in the list will be checked for syntax correctness but not existence. Case is preserved when values are passed to the Run function. The TO and

Attribute	Use	Description	Valid Values
			ALL keywords are not supported. **InputFile**. A file specification for an input file. The specified file must exist. The case is preserved when passed to the Run function. **OutputFile**. A file specification for an output file. The specified directory path must exist and either the specified file does not exist or, if it exists, it must be writable. The case is preserved when passed to the Run function.

XML Representation

```
<xs:element name="Parameter" type="parameter-content">
   <xs:sequence maxOccurs="unbounded" minOccurs="0">
      <xs:element name="EnumValue"></xs:element>
   </xs:sequence>
   <xs:attribute name="Name" use="required"></xs:attribute>
   <xs:attribute name="ParameterType" use="required">
      <xs:simpleType>
         <xs:restriction base="xs:token">
            <xs:enumeration value="DatasetName"></xs:enumeration>
            <xs:enumeration value="Integer"></xs:enumeration>
            <xs:enumeration value="IntegerList"></xs:enumeration>
            <xs:enumeration value="Keyword"></xs:enumeration>
            <xs:enumeration value="KeywordList"></xs:enumeration>
            <xs:enumeration value="LeadingToken"></xs:enumeration>
            <xs:enumeration value="Number"></xs:enumeration>
            <xs:enumeration value="NumberList"></xs:enumeration>
            <xs:enumeration value="QuotedString"></xs:enumeration>
            <xs:enumeration value="TokenList"></xs:enumeration>
            <xs:enumeration value="VariableName"></xs:enumeration>
            <xs:enumeration value="VariableNameList"></xs:enumeration>
            <xs:enumeration value="InputFile"></xs:enumeration>
            <xs:enumeration value="OutputFile"></xs:enumeration>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
</xs:element>
```

Parent Elements

Subcommand Element

Child Elements

EnumValue Element

EnumValue Element

EnumValue is used to enumerate a set of allowed values. EnumValue elements are ignored except for Keyword, KeywordList, Number, Integer, and QuotedString parameters. When used with Keyword or KeywordList parameters, the specified EnumValue elements represent the complete set of allowed values. When used with Number, Integer, or QuotedString parameters, the specified EnumValue elements represent a set of valid keywords in addition to the specified type. For example, the value AUTO might be specified as an allowed keyword value for an Integer parameter. The parameter can then be specified as an integer or the unquoted string AUTO.

Table 3-3
Attributes for EnumValue

Attribute	Use	Description	Valid Values
Name	**required**	The enumerated value. Case is ignored.	*any*

XML Representation

```
<xs:element name="EnumValue">
   <xs:attribute name="Name" use="required"></xs:attribute>
</xs:element>
```

Parent Elements

Parameter Element

Subcommand Element

Subcommands divide a command's functionality into distinct groups. Typical subcommands include SAVE for specifying variables to be saved to the active dataset, PRINT for specifying tabular output, and PLOT for specifying chart output. A subcommand can only be specified once per command. The name of a subcommand must be preceded by a forward slash when specified in command syntax.

Table 3-4
Attributes for Subcommand

Attribute	Use	Description	Valid Values
IsArbitrary	optional	Allows arbitrary tokens on the subcommand. This is useful, for example, for specifying variable lists and model effect lists.	**True**. **False**. **Yes**. Equivalent to True. **No**. Equivalent to False.

Attribute	Use	Description	Valid Values
Name	**required**	The name of the subcommand. Subcommand names are restricted to 7-bit ascii characters and start with a letter or one of the characters @, #, or $. Subsequent characters can be any combination of letters, numbers, nonpunctuation characters, and a period (.). To specify the anonymous subcommand, use Name="".	*any*
Occurrence	optional	Specifies whether the subcommand must be included in a syntax job for the command to run.	**Required** **Optional**

XML Representation

```
<xs:element name="Subcommand" type="subcommand-content">
   <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="Parameter"></xs:element>
   </xs:sequence>
   <xs:attribute name="Name" use="required"></xs:attribute>
   <xs:attribute name="Occurrence">
      <xs:simpleType>
         <xs:restriction base="xs:token">
            <xs:enumeration value="Required"></xs:enumeration>
            <xs:enumeration value="Optional"></xs:enumeration>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
   <xs:attribute name="IsArbitrary">
      <xs:simpleType>
         <xs:restriction base="xs:token">
            <xs:enumeration value="True"></xs:enumeration>
            <xs:enumeration value="False"></xs:enumeration>
            <xs:enumeration value="Yes"></xs:enumeration>
            <xs:enumeration value="No"></xs:enumeration>
         </xs:restriction>
      </xs:simpleType>
   </xs:attribute>
</xs:element>
```

Parent Elements

Command Element

Child Elements

Parameter Element

Working with Arbitrary Tokens

TokenList parameter types and Subcommand elements with IsArbitrary=True consist of arbitrary tokens, such as character strings, literals (strings delimited by single or double quotes), and operators like '>'. Typical scenarios where arbitrary tokens are required include variable lists that may contain BY or WITH specifications and model effects lists. For example, a subcommand that allows arbitrary tokens for specifying model interaction effects might be given as:

```
/MODEL var1*var2(var3)
```

When parsed, this results in the following list of six tokens passed to the Run function of the implementation code.

```
['var1','*','var2','(','var3',')']
```

Likewise, an anonymous subcommand that allows arbitrary tokens to specify variables with optional BY or WITH keywords might be specified as:

```
DepVar BY A B WITH X Y
```

When parsed, this results in the following list of seven tokens passed to the Run function of the implementation code.

```
['DepVar','BY','A','B','WITH','X','Y']
```

When working with arbitrary tokens you'll want to test the various forms of input that your implementation function will need to handle, since the tokens that result from the specified input may not always be what you expect. For example, a TokenList parameter named TOKENS might be specified as:

```
TOKENS = 1a 2b
```

When parsed, this results in the following list of four tokens passed to the Run function of the implementation code.

```
['1','a','2','b']
```

The result reflects the standard way that IBM® SPSS® Statistics tokenizes command syntax. In particular, when a set of digits precedes a set of characters, the digits are treated as a separate token. You can force a set of characters to be passed as a single token by enclosing them in quotes. For example, specifying `TOKENS = '1a' '2b'` results in the token list `['1a','2b']`. The same applies if you need to specify multi-word phrases and have each phrase passed as a single token. For example, `TOKENS = 'two words'` results in the single token `'two words'`.

Examples

Using the Keyword Type

The Keyword type is used to specify a parameter that takes a single value. As an example, consider an OPTIONS subcommand with a parameter for controlling missing values, and represented in the syntax diagram as:

```
/OPTIONS MISSING={PAIRWISE}
                 {LISTWISE}
```

The specification of MISSING is best handled with a Keyword type parameter. The XML syntax specification for the OPTIONS subcommand is:

```
<Subcommand Name="OPTIONS">
  <Parameter Name="MISSING" ParameterType="Keyword">
    <EnumValue Name="PAIRWISE"/>
    <EnumValue Name="LISTWISE"/>
  </Parameter>
</Subcommand>
```

An example of command syntax containing the OPTIONS subcommand is:

```
/OPTIONS MISSING=LISTWISE
```

The Keyword type always requires the parameter name, followed by an equals sign, followed by a single value.

Using the KeywordList Type

The KeywordList type is used to specify a parameter that can take on multiple values. As an example, consider an OPTIONS subcommand with a parameter for specifying one or more file types from a fixed set, and represented in the syntax diagram as:

```
/OPTIONS FILETYPES=[SAV SAS STATA]
```

The specification of FILETYPES is best handled with a KeywordList type parameter. The XML syntax specification for the OPTIONS subcommand is:

```
<Subcommand Name="OPTIONS">
  <Parameter Name="FILETYPES" ParameterType="KeywordList">
    <EnumValue Name="SAV"/>
    <EnumValue Name="SAS"/>
    <EnumValue Name="STATA"/>
  </Parameter>
</Subcommand>
```

An example of command syntax containing the OPTIONS subcommand is:

```
/OPTIONS FILETYPES=SAV SAS
```

The KeywordList type always requires the parameter name, followed by an equals sign, followed by one or more values.

Using the LeadingToken Type

The LeadingToken type is used to specify a parameter that has a name but no associated value. As an example, consider a PLOT subcommand for specifying types of plots to include in output, and represented in the syntax diagram as:

```
/PLOT OBSERVED FORECAST FIT
```

The specification of PLOT is best handled with a set of LeadingToken type parameters. The XML syntax specification for the PLOT subcommand is:

```
<Subcommand Name="PLOT">
  <Parameter Name="OBSERVED" ParameterType="LeadingToken"/>
  <Parameter Name="FORECAST" ParameterType="LeadingToken"/>
  <Parameter Name="FIT" ParameterType="LeadingToken"/>
</Subcommand>
```

An example of command syntax containing the PLOT subcommand is:

```
/PLOT OBSERVED FIT
```

Using the TokenList Type

The TokenList type is used to specify a parameter that can take on multiple values. It is similar to the KeywordList type but TokenList values are not bound by the rules required of KeywordList values. As an example, consider a MODEL subcommand with a parameter for specifying model interaction effects, and represented in the syntax diagram as:

```
/MODEL EFFECTS=effect-list
```

The specification of the effects list is handled with a TokenList type parameter. The XML syntax specification for the MODEL subcommand is:

```
<Subcommand Name="MODEL">
  <Parameter Name="EFFECTS" ParameterType="TokenList"/>
</Subcommand>
```

An example of command syntax containing the MODEL subcommand is:

```
/MODEL EFFECTS=A*B C(D)
```

Appendix

A

Localizing Extension Commands Implemented in Python

This section assumes that the reader is familiar with the IBM® SPSS® Statistics extension mechanism and the classes and functions in the Python `extension` module.

Extension commands implemented in Python can be enabled for translation using standard Python tools and the classes and functions in version 1.4.0 or later of the `extension` module, a supplementary module installed with the IBM® SPSS® Statistics - Integration Plug-in for Python. Version 1.4.0 or higher of the `extension` module is included with version 19 or higher of the Integration Plug-in for Python. It can also be downloaded from SPSS community and used with versions 16-18. Extension command code modified for translation as described here will not fail with older versions of the `extension` module, but the output will not be translated.

The localization process consists of the following steps:

- Modifying the Python implementation code to identify translatable strings
- Extracting translatable text from the implementation code using standard Python tools
- Preparing a translated file of strings for each target language
- Installing the translation files along with the extension command

Notes

- When running an extension command from within SPSS Statistics, the language for extension command output will be automatically synchronized with the SPSS Statistics output language (`OLANG`). When running an extension command from an external Python process, such as a Python IDE, you can set the output language by submitting a `SET OLANG` command when SPSS Statistics is started. If no translation for an item is available for the output language, the untranslated string will be used.
- Messages produced by the `extension` module, such as error messages for violation of the specifications in the Syntax definition, are automatically produced in the current output language. Exceptions raised in the extension command implementation code are automatically converted to a Warnings pivot table.
- Translation of dialog boxes built with the Custom Dialog Builder is a separate process, but translators should ensure that the dialog and extension command translations are consistent.

Modifying the Python code

First, ensure that the text to be translated is in a reasonable form for translation.

- Do not build up text by combining fragments of text in code. This makes it impossible to rearrange the text according to the grammar of the target languages and makes it difficult for translators to understand the context of the strings.
- Avoid using multiple parameters in a string. Translators may need to change the parameter order.
- Avoid the use of abbreviations and colloquialisms that are difficult to translate.

Enclose each translatable string in a call to the underscore function `"_"`. For example:

```
_("File not found:  %s") % filespec
```

The _ function will fetch the translation, if available, when the statement containing the string is executed. The following limitations apply:

- Never pass an empty string as the argument to `_`, i.e., `_("")`. This will damage the translation mechanism.
- Do not use the underscore function in static text such as class variables. The _ function is defined dynamically.
- The _ function, as defined in the `extension` module, always returns Unicode text even if IBM® SPSS® Statistics is running in code page mode. If there are text parameters in the string as in the example above, the parameter should be in Unicode. The automatic conversion used in the parameter substitution logic will fail if the parameter text contains any extended characters. One way to resolve this is as follows, assuming that the `locale` module has been imported.

```
if not isinstance(filespec, unicode):
   filespec = unicode(filespec, locale.getlocale()[1])
   _("File not found: %s") % filespec
```

Note: There is a conflict between the definition of the _ function as used by the Python modules (`pygettext` and `gettext`) that handle translations, and the automatic assignment of interactively generated expression values to the variable _. In order to resolve this, the translation initialization code in the `extension` module disables this assignment.

For users with IBM SPSS Statistics version 19 or higher

Calls to the `spss.StartProcedure` function (or the `spss.Procedure` class) should use the form `spss.StartProcedure(procedureName,omsIdentifier)` where *procedureName* is the translatable name associated with output from the procedure and *omsIdentifier* is the language invariant `OMS` command identifier associated with the procedure. For example:

```
spss.StartProcedure(_("Demo"),"demoId")
```

For users with IBM SPSS Statistics versions prior to 19

If the extension command is to be used with SPSS Statistics versions prior to 19, a few extra steps are necessary.

▶ Insert the following code after the call to the `Syntax` class, which is typically in the `Run` function that implements the command.

```
#enable localization
global _
try:
   _("---")
except:
   def _(msg):
     return msg
```

This ensures that the _ function is defined if an older version of the `extension` module is being used. However, the text will not actually be translated unless a current version (1.4.0 or higher) of the `extension` module is used.

For versions 16 and 17, the language for extension command output will NOT be automatically synchronized with the SPSS Statistics output language, even with a current version of the `extension` module. However, the language for extension command output can be set statically. Before launching SPSS Statistics, set the `LANGUAGE` environment variable to the desired output language, using the appropriate operating system command. Use the language identifiers that would be used in a `SET OLANG` command or the POSIX language names. On Windows XP this can be done using Control Panel>System>Advanced>Environment Variables.

- If there are print statements in the code that could display translated text or other non-ascii text such as file names, they must be removed or converted to pivot tables. The `NonProcPivotTable` class in the `extension` module (version 1.4.0 or higher) is a convenient alternative to print. Text in exception messages is converted to a Warnings pivot table by the `extension` module, but will simply be printed by older versions which will fail if it contains extended characters.

- If there is any non-ascii pivot table cell text, it must be removed. This limitation does not apply to other areas of pivot tables such as labels.

- Do not translate the procedure name passed to the `spss.StartProcedure` function or the `spss.Procedure` class. Doing so would change the OMS command identifier, which is expected to be language invariant.

Extracting translatable text

The Python implementation code is never modified by the translators. Translation is accomplished by extracting the translatable text from the code files and then creating separate files containing the translated text, one file for each language. The _ function uses compiled versions of these files.

The standard Python distribution includes `pygettext.py`, which is a command line script that extracts strings marked as translatable (i.e., strings wrapped in the _ function) and saves them to a *.pot* file. Run `pygettext.py` on the implementation code, and specify the name of the implementing Python module (the module containing the `Run` function) as the name of the output file, but with the extension *.pot*. If the implementation uses multiple Python files, the *.pot* files for each should be combined into one under the name of the main implementing module (the module containing the `Run` function).

- Change the *charset* value, in the `msgstr` field corresponding to `msgid ""`, to utf-8.

- A *pot* file includes one `msgid` field with the value `""`, with an associated `msgstr` field containing metadata. There must be only one of these.
- Optionally, update the generated title and organization comments.

Documentation for `pygettext.py` is available from the topic on the `gettext` module in the Python help system.

Translating the pot file

Translators enter the translation of each `msgid` into the corresponding `msgstr` field and save the result as a file with the same name as the *pot* file but with the extension *.po*. There will be one *po* file for each target language.

- *po* files should be saved in Unicode utf-8 encoding.
- *po* files should not have a BOM (Byte Order Mark) at the start of the file.
- If a `msgstr` contains an embedded double quote character (x22), precede it with a backslash (\). as in:

  ```
  msgstr "He said, \"Wow\", when he saw the R-squared"
  ```

- `msgid` and `msgstr` entries can have multiple lines. Enclose each line in double quotes.

Each translated *po* file is compiled into a binary format by running `msgfmt.py` from the standard Python distribution, giving the output the same name as the *po* file but with an extension of *.mo*.

Installing the mo files

When installed, the `mo` files should reside in the following directory structure:

lang/<language-identifier>/LC_MESSAGES/<command name>.mo

- <*command name*> is the name of the extension command in upper case with any spaces replaced with underscores, and is the same as the name of the Python implementation module. Note that the `mo` files have the same name for all languages.
- <*language-identifier*> is the identifier for a particular language. Identifiers for the languages supported by IBM® SPSS® Statistics are shown in the table.

For example, if the extension command is named *MYORG MYSTAT* then an *mo* file for French should be stored in *lang/fr/LC_MESSAGES/MYORG_MYSTAT.mo*.

Manually installing translation files

If you are manually installing an extension command and associated translation files, then the *lang* directory containing the translation files should be installed in the <*command name*> directory under the directory where the Python implementation module is installed.

For example, if the extension command is named *MYORG MYSTAT* and the associated Python implementation module (*MYORG_MYSTAT.py*) is located in the *extensions* directory (under the location where SPSS Statistics is installed), then the *lang* directory should reside under *extensions/MYORG_MYSTAT*.

Using the example of a French translation discussed above, an *mo* file for French would be stored in *extensions/MYORG_MYSTAT/lang/fr/LC_MESSAGES/MYORG_MYSTAT.mo*.

Deploying translation files to other users

If you are localizing output for a custom dialog or extension command that you intend to distribute to other users, then you should create an extension bundle (requires SPSS Statistics version 18 or higher) to package your translation files with your custom components. Specifically, you add the *lang* directory containing your compiled translation files (*mo* files) to the extension bundle during the creation of the bundle (from the Translation Catalogues Folder field on the Optional tab of the Create Extension Bundle dialog). When an end user installs the extension bundle, the directory containing the translation files is installed in the *extensions/<extension bundle name>* directory under the SPSS Statistics installation location, and where *<extension bundle name>* is the name of the extension bundle with spaces replaced by underscores. *Note*: An extension bundle that includes translation files for an extension command should have the same name as the extension command.

- If the *SPSS_EXTENSIONS_PATH* environment variable has been set, then the *extensions* directory (in *extensions/<extension bundle name>*) is replaced by the first writable directory in the environment variable.
- Information on creating extension bundles is available from the Help system, under Core System>Utilities>Working with Extension Bundles.

Language Identifiers

de. German

en. English

es. Spanish

fr. French

it. Italian

ja. Japanese

ko. Korean

pl. Polish

pt_BR. Brazilian Portuguese

ru. Russian

zh_CN. Simplified Chinese

zh_TW. Traditional Chinese

Appendix

B

Notices

This information was developed for products and services offered worldwide.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing, Legal and Intellectual Property Law, IBM Japan Ltd., 1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242-8502 Japan.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Software Group, Attention: Licensing, 233 S. Wacker Dr., Chicago, IL 60606, USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, ibm.com, and SPSS are trademarks of IBM Corporation, registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at *http://www.ibm.com/legal/copytrade.shtml*.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

This product uses WinWrap Basic, Copyright 1993-2007, Polar Engineering and Consulting, *http://www.winwrap.com*.

Other product and service names might be trademarks of IBM or other companies.

Adobe product screenshot(s) reprinted with permission from Adobe Systems Incorporated.

Microsoft product screenshot(s) reprinted with permission from Microsoft Corporation.

Index