Python Integration Package for IBM SPSS Statistics

Note: Before using this information and the product it supports, read the general information under Notices on p. 117.

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 Python Programs

The Python® Integration Package for IBM® SPSS® Statistics allows you to create **Python programs** that control the flow of command syntax jobs, read and write data, and create custom procedures that generate their own pivot table output. This feature requires the IBM® SPSS® Statistics - Integration Plug-in for Python, installed with IBM® SPSS® Statistics - Essentials for Python.

A companion interface is available for creating **Python scripts** that operate on the SPSS Statistics user interface and manipulate output objects. For information, see the topic for the Scripting Guide for IBM SPSS Statistics, under Integration Plug-in for Python in the Help system.

Python programming features described here are available inside `BEGIN PROGRAM-END PROGRAM` program blocks in command syntax. A program block provides access to all the functionality of the Python programming language, including the functions specific to SPSS Statistics and provided in the Python Integration Package for SPSS Statistics. You can use program blocks to combine the programmability features of Python with all the capabilities of SPSS Statistics by building strings of command syntax that are then executed by SPSS Statistics.

You can also run SPSS Statistics from an external Python process, such as a Python IDE or the Python interpreter. For more information, see the topic Running IBM SPSS Statistics from an External Python Process on p. 9.

Within a program block, Python is in control, and it knows nothing about SPSS Statistics commands. When the Python Integration Package for SPSS Statistics is loaded, Python knows about the functions provided in the package, but standard SPSS Statistics commands are basically invalid within a program block. For example:

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

will generate an error, because `FREQUENCIES` is not recognized by Python. But since the goal of a program block is typically to generate some command syntax that SPSS Statistics can understand, there must be a way to specify command syntax within a program block. This is done by expressing syntax commands, or parts of commands, as character strings, as in:

```
spss.Submit("FREQUENCIES VARIABLES=var1, var2, var3.")
```

The real power of program blocks comes from the ability to dynamically build strings of command syntax, as in:

```
BEGIN PROGRAM PYTHON.
import spss
string1="DESCRIPTIVES VARIABLES="
N=spss.GetVariableCount()
scaleVarList=[]
for i in xrange(N):
```

```
  if spss.GetVariableMeasurementLevel(i)=='scale':
    scaleVarList.append(spss.GetVariableName(i))
string2="."
spss.Submit([string1, ' '.join(scaleVarList), string2])
END PROGRAM.
```

- `spss.GetVariableCount` returns the number of variables in the active dataset.
- `if spss.GetVariableMeasurementLevel(i)=="scale"` is true only for variables with a scale measurement level.
- `scaleVarList.append(spss.GetVariableName(i))` builds a list of variable names that includes only those variables with a scale measurement level.
- `spss.Submit` submits a `DESCRIPTIVES` command to SPSS Statistics that looks something like this:

```
DESCRIPTIVES VARIABLES=
scalevar1 scalevar2 scalevar3...etc.
.
```

Working with Python Program Blocks

Use `SET PRINTBACK ON MPRINT ON` to display the syntax generated by program blocks.

Example

```
SET PRINTBACK ON MPRINT ON.
GET FILE='/examples/data/Employee data.sav'.
BEGIN PROGRAM PYTHON.
import spss
scaleVarList=[]
catVarList=[]
varcount=spss.GetVariableCount()
for i in xrange(varcount):
  if spss.GetVariableMeasurementLevel(i)=='scale':
    scaleVarList.append(spss.GetVariableName(i))
  else:
    catVarList.append(spss.GetVariableName(i))
spss.Submit("""
FREQUENCIES
 /VARIABLES=%s.
DESCRIPTIVES
   /VARIABLES=%s.
""" %(' '.join(catVarList), ' '.join(scaleVarList)))
END PROGRAM.
```

The generated command syntax is displayed in the log in the IBM® SPSS® Statistics Viewer:

```
225 M>  FREQUENCIES
226 M>    /VARIABLES=gender educ jobcat minority.
227 M>  DESCRIPTIVES
228 M>    /VARIABLES=id bdate salary salbegin jobtime prevexp.
```

Basic Specification for a Python Program Block

The basic specification for a Python program block is `BEGIN PROGRAM PYTHON` (the keyword `PYTHON` can be omitted) followed by one or more Python statements, followed by `END PROGRAM`.

Note: The Python function `sys.exit()` is not supported for use within a program block.

- The first program block in a session should start with the Python function `import spss`, which imports the `spss` module, providing access to the functions in the Python Integration Package for IBM® SPSS® Statistics. For more information, see the topic Python Functions and Classes in Chapter 2 on p. 15.
- Subsequent program blocks in the same session do not require `import spss`, and it is silently ignored if the module has already been imported.

Example

```
DATA LIST FREE /var1.
BEGIN DATA
1
END DATA.
DATASET NAME File1.
BEGIN PROGRAM PYTHON.
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 PYTHON.
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.
- 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.
- Since the value of *File1N* persists from the first program block, the two variable counts can be compared in the second program block.

Syntax Rules

- Within a program block, only statements recognized by the specified programming language are allowed.
- Command syntax generated within a program block must follow **interactive** syntax rules.

- Within a program block, each line should not exceed 251 bytes (although syntax generated by those lines can be longer).
- With the SPSS Statistics Batch Facility (available only with SPSS Statistics Server), use the -i switch when submitting command files that contain program blocks. All command syntax (not just the program blocks) in the file must adhere to interactive syntax rules.

Within a program block, the programming language is in control, and the syntax rules for that programming language apply. Command syntax generated from within program blocks must always follow interactive syntax rules. For most practical purposes this means command strings you build in a programming block must contain a period (.) at the end of each command.

Scope and Limitations

- Programmatic variables created in a program block cannot be used outside of program blocks.
- Program blocks cannot be contained within `DEFINE-!ENDDEFINE` macro definitions.
- Program blocks can be contained in command syntax files run via the `INSERT` command, with the default `SYNTAX=INTERACTIVE` setting.
- Program blocks cannot be contained within command syntax files run via the `INCLUDE` command.
- Python variables specified in a given program block persist to subsequent program blocks.
- Python programs (*.py*, *.pyc*) utilizing the `spss` module cannot be run as autoscripts, nor are they intended to be run from Utilities>Run Script.

More information about Python programs and Python scripts is available from the SPSS Statistics Help system, and accessed from Core System>Scripting Facility.

Nested Program Blocks

From within Python, you can submit command syntax containing a `BEGIN PROGRAM` block, thus allowing you to nest program blocks. This can be done by including the nested program block in a separate command syntax file and submitting an `INSERT` command to read in the block. It can also be done by submitting the nested program block from within a user-defined Python function.

Example: Nesting Program Blocks Using the INSERT Command

```
import spss
spss.Submit("INSERT FILE='/myprograms/nested_block.sps'.")
```

The file */myprograms/nested_block.sps* would contain a `BEGIN PROGRAM` block, as in:

```
BEGIN PROGRAM PYTHON.
import spss
<Python code>
END PROGRAM.
```

Note: You cannot import a Python module containing code that nests a program block, such as the above code that uses the `INSERT` command to insert a file containing a program block. If you wish to encapsulate nested program blocks in a Python module that can be imported, then embed the nesting code in a user-defined function as shown in the following example.

Example: Nesting Program Blocks With a User-Defined Python Function

```
import spss, myfuncs
myfuncs.demo()
```

- `myfuncs` is a user-defined Python module containing the function (`demo`) that will submit the nested program block.

  A Python module is simply a text file containing Python definitions and statements. You can create a module with a Python IDE, or with any text editor, by saving a file with an extension of *.py*. The name of the file, without the *.py* extension, is then the name of the module.
- The `import` statement includes `myfuncs` so that it is loaded along with the `spss` module. To be sure that Python can find your module, you may want to save it to your Python "site-packages" directory, typically */Python27/Lib/site-packages*.
- The code `myfuncs.demo()` calls the function `demo` in the `myfuncs` module.

Following is a sample of the contents of `myfuncs`.

```
import spss
def demo():
   spss.Submit("""
BEGIN PROGRAM PYTHON.
<Python code>
END PROGRAM.""")
```

- The sample `myfuncs` module includes an `import spss` statement. This is necessary since a function in the module makes use of a function from the `spss` module—specifically, the `Submit` function.
- The nested program block is contained within a Python triple-quoted string. Triple-quoted strings allow you to specify a block of commands on multiple lines, resembling the way you might normally write command syntax.
- Notice that `spss.Submit` is indented but the `BEGIN PROGRAM` block is not. Python statements that form the body of a user-defined Python function must be indented. The level of indentation is arbitrary but must be the same for all statements in the function body. The `BEGIN PROGRAM` block is passed as a string argument to the `Submit` function and is processed by IBM® SPSS® Statistics as a block of Python statements. Python statements are not indented unless they are part of a group of statements, as in a function or class definition, a conditional expression, or a looping structure.

Notes

- You can nest program blocks within nested program blocks, up to five levels of nesting.
- Python variables specified in a nested program block are local to that block unless they are specified as global variables. In addition, Python variables specified in a program block that invokes a nested block can be read, but not modified, in the nested block.

- Nested program blocks are not restricted to being Python program blocks, but you can only submit a nested block from Python. For example, you can nest an R program block in a Python program block, but you cannot nest a Python program block in an R program block.
- If a `Submit` function containing a triple quoted string nests a Python program block containing another triple quoted string, use a different type of triple quotes in the nested block. For example, if the outer block uses triple double quotes, then use triple single quotes in the nested block.

Unicode Mode

When IBM® SPSS® Statistics is in Unicode mode (controlled by the `UNICODE` subcommand of the `SET` command) the following conversions are automatically done when passing and receiving strings through the functions available with the `spss` module:

- Strings received by Python from SPSS Statistics are converted from UTF-8 to Python Unicode, which is UTF-16.
- Strings passed from Python to SPSS Statistics are converted from UTF-16 to UTF-8.

Note: Changing the locale and/or the unicode setting during an `OMS` request may result in incorrectly transcoded text.

Command Syntax Files

Special care must be taken when working in Unicode mode with command syntax files. Specifically, Python string literals used in command syntax files need to be explicitly expressed as UTF-16 strings. This is best done by using the `u()` function from the `spssaux` module (installed with IBM® SPSS® Statistics - Essentials for Python). The function has the following behavior:

- If SPSS Statistics is in Unicode mode, the input string is converted to UTF-16.
- If SPSS Statistics is not in Unicode mode, the input string is returned unchanged.

Note: If the string literals in a command syntax file only consist of plain roman characters (7-bit ascii), the `u()` function is not needed.

The following example demonstrates some of this behavior and the usage of the `u()` function.

```
set unicode on locale=english.
BEGIN PROGRAM.
import spss, spssaux
from spssaux import u
literal = "âbc"
try:
   print "literal without conversion:", literal
except:
   print "can't print literal"
try:
   print "literal converted to utf-16:", u(literal)
except:
   print "can't print literal"
END PROGRAM.
```

Following are the results:

```
literal without conversion: can't print literal
literal converted to utf-16: âbc
```

Truncating Unicode Strings

When working in Unicode mode, use the `truncatestring` function from the `spssaux` module (installed with Essentials for Python) to correctly truncate a string to a specified maximum length in bytes. This is especially useful for truncating strings to be used as SPSS Statistics variable names, which have a maximum allowed length of 64 bytes.

The `truncatestring` function takes two arguments—the string to truncate, and the maximum number of bytes, which is optional and defaults to 64. For example:

```
import spss, spssaux
newstring = spssaux.truncatestring(string,8)
```

Python Syntax Rules

Within a Python program block, only statements and functions recognized by Python are allowed. Python syntax rules differ from IBM® SPSS® Statistics 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 variable name of *myvariable* is not the same as *MyVariable*, and the function `spss.GetVariableCount` cannot be written as `SPSS.getvariablecount`.

Python uses UNIX-style path specifications, with forward slashes. This applies even for SPSS Statistics command syntax generated within a Python program block. For example:

```
spss.Submit("GET FILE '/data/somedata.sav'.")
```

Alternatively, you can escape each backslash with another backslash, as in:

```
spss.Submit("GET FILE '\\data\\somedata.sav'.")
```

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. The expression continues implicitly until the closing character for the expression.
- **Explicit.** All other expression 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, as is the body of a user-defined Python function. 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 xrange(varcount):
  if spss.GetVariableMeasurementLevel(i)=="scale":
```

```
      ScaleVarList=ScaleVarList + " " + spss.GetVariableName(i)
  else:
     CatVarList=CatVarList + " " + spss.GetVariableName(i)
print CatVarList
```

Note: You should avoid the use of tab characters in Python code within `BEGIN PROGRAM-END PROGRAM` blocks. For line indentation, use spaces.

Working with Multiple Versions of IBM SPSS Statistics

Multiple versions of the IBM® SPSS® Statistics - Integration Plug-in for Python can be used on the same machine, each associated with a major version of IBM® SPSS® Statistics, such as 20 or 21.

Running Python Programs from Within IBM SPSS Statistics

- For versions 14.0 and 15.0, Python programs run from within SPSS Statistics will automatically use the appropriate version of the plug-in.
- For versions 16.0 and higher, and by default, Python programs run from within the last installed version of SPSS Statistics will automatically use the appropriate version of the plug-in. To run Python programs from within a different version of SPSS Statistics, use the spss.SetDefaultPlugInVersion function to set the default to a different version (the setting persists across sessions). You can then run Python programs from within the other version. If you are attempting to change the default version from 16.0 to 17.0, additional configuration is required; please see the Notes below.

Running Python Programs from an External Python Process

When driving the SPSS Statistics backend from a separate Python process, such as the Python interpreter or a Python IDE, the plug-in will drive the version of the SPSS Statistics backend that matches the default plug-in version specified for that version of Python. Unless you change it, the default plug-in version for a given version of Python (such as Python 2.7) is the last one installed. You can view the default version using the spss.GetDefaultPlugInVersion function and you can change the default version using the spss.SetDefaultPlugInVersion function. The setting persists across sessions. If you are attempting to change the default version from 16.0 to 17.0 please see the Notes below.

Notes

- If you are using the `spss.SetDefaultPlugInVersion` function to change the default from version 16.0 to version 17.0, you should also manually modify the file *SpssClient.pth* located in the Python 2.5 *site-packages* directory. Change the order of entries in the file so that the first line is `SpssClient170`.

  Windows. The *site-packages* directory is located in the *Lib* directory under the Python 2.5 installation directory—for example, *C:\Python25\Lib\site-packages*.

  Mac OS X 10.4 (Tiger). The *site-packages* directory is located at */Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/site-packages*.

Mac OS X 10.5 (Leopard). The *site-packages* directory is typically located at */Library/Python/2.5/site-packages*.

Linux and UNIX Server. The *site-packages* directory is located in the */lib/python2.5/* directory under the Python 2.5 installation directory—for example, */usr/local/python25/lib/python2.5/site-packages*.

- Beginning with version 15.0, a restructuring of the Integration Plug-in for Python installation directory and changes to some class structures may affect Python code written for an earlier version and used with a 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.cursors.Cursor` when run with version 14.0, `spss.spss150.cursors.ReadCursor` when run with version 15.0, and `spss.cursors.ReadCursor` when run with a version higher than 15.0.

Python and IBM SPSS Statistics Working Directories

When running Python code that is within a `BEGIN PROGRAM-END PROGRAM` block and that contains relative paths in file specifications, you will need to understand the notions of working directories, both for Python and IBM® SPSS® Statistics. You may want to avoid the subtleties involved with working directories by avoiding the use of relative paths and using full paths for file specifications.

- Relative paths used for file specifications in command syntax submitted from Python (with `spss.Submit`) are relative to the SPSS Statistics backend working directory. The SPSS Statistics backend working directory determines the full path used for file specifications in command syntax in the case where only a relative path is provided. It can be changed with the `CD` command, but is not affected by actions involving the file open dialogs, and it is private to the SPSS Statistics backend.
- Relative paths used when reading and writing files with built-in Python functions—such as `open`—are relative to the Python current working directory. You can get the Python current working directory from the `getcwd` function in the `os` module.

Running IBM SPSS Statistics from an External Python Process

You can run Python programs utilizing the `spss` module from any external Python process, such as a Python IDE or the Python interpreter. In this mode, the Python program starts up a new instance of the IBM® SPSS® Statistics processor without an associated instance of the SPSS Statistics client. You can use this mode to debug your Python programs using the Python IDE of your choice.

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

Linux Users

In order to drive the SPSS Statistics processor from an external Python process on Linux, the following locations need to be added to the *LD_LIBRARY_PATH* environment variable:

```
LD_LIBRARY_PATH=<PYTHON_HOME>/lib:<SPSS_HOME>/lib:$LD_LIBRARY_PATH
```

where <PYTHON_HOME> is the location where Python is installed—typically, */usr/local*—and where <SPSS_HOME> is the installation location of SPSS Statistics—for example, */opt/IBM/SPSS/Statistics/21*.

Mac Users

To drive the SPSS Statistics processor from an external Python process on Mac, launch the *Programmability External Python Process* application, installed with IBM® SPSS® Statistics - Essentials for Python and located in the directory where SPSS Statistics is installed. The application launches IDLE (the default IDE provided with Python) and sets environment variables necessary for driving SPSS Statistics. If you choose not to use the *Programmability External Python Process* application you will need to add a number of locations to the *DYLD_LIBRARY_PATH* environment variable as follows:

```
export
DYLD_LIBRARY_PATH=<SPSS_HOME>/lib:<SPSS_HOME>/Library/Frameworks/Sentinel.framework/Versions/A:
<SPSS_HOME>/Library/Frameworks/SuperPro.framework/Versions/A
```

where <SPSS_HOME> is the location of the *Contents* folder in the SPSS Statistics application bundle—for example, */Applications/IBM/SPSS/Statistics/21/SPSSStatistics.app/Contents*.

Localizing Output from Python Programs

You can localize output, such as messages and pivot table strings, from extension commands implemented in Python. 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

The process described here assumes use of the Python `extension` module, which is installed with IBM® SPSS® Statistics - Essentials for Python.

Notes

- When running an extension command from within IBM® 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.

Additional Resources

Examples of extension commands implemented in Python with localized output are included with Essentials for Python. The Python modules for these examples are located in the *extensions* directory under the SPSS Statistics installation directory. If you have specified alternate locations for extension commands with the *SPSS_EXTENSIONS_PATH* environment variable then the Python modules will be located in the first writable location in that variable instead of in the *extensions* directory.

Information on creating extension commands is also available from the following sources:

- The article *"Writing SPSS Statistics Extension Commands"*, available from the SPSS community at *http://www.ibm.com/developerworks/spssdevcentral*.
- The chapter on Extension Commands in *Programming and Data Management for SPSS Statistics*, available in PDF from the Articles page at *http://www.ibm.com/developerworks/spssdevcentral*.

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.

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")
```

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 Language Identifiers 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

Chapter
2

Python Functions and Classes

The Python Integration Package for IBM® SPSS® Statistics contains functions and classes that facilitate the process of using Python programming features with SPSS Statistics, including those that:

Build and run command syntax

- spss.Submit

Get information about data files in the current IBM SPSS Statistics session

- spss.GetCaseCount
- spss.GetDataFileAttributes
- spss.GetFileHandles
- spss.GetMultiResponseSet
- spss.GetSplitVariableNames
- spss.GetVarAttributes
- spss.GetVariableCount
- spss.GetVariableFormat
- spss.GetVariableLabel
- spss.GetVariableMeasurementLevel
- spss.GetVariableName
- spss.GetVariableType
- spss.GetVarMissingValues
- spss.GetWeightVar

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

- spss.Cursor

Access and manage multiple datasets

- spss.ActiveDataset
- spss.Dataset
- spss.GetDatasets
- spss.GetFileHandles
- spss.IsActive
- spss.SetActive

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 IBM SPSS Statistics - Integration Plug-in for Python

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

Locale and Output Language Settings

- spss.GetSPSSLocale
- spss.SetOutputLanguage

Brief descriptions of each function are available using the Python `help` function, as in:

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

spss.ActiveDataset Function

spss.ActiveDataset(). *Returns the name of the active dataset.*

- If the active dataset is unnamed, '*' is returned.

Example

```
import spss
name = spss.ActiveDataset()
```

spss.AddProcedureFootnotes Function

spss.AddProcedureFootnotes(footnote). *Adds a footnote to all tables generated by a procedure.* The argument *footnote* is a string specifying the footnote.

- The `AddProcedureFootnotes` function 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.demoProc")
spss.AddProcedureFootnotes("A footnote")
table = spss.BasePivotTable("Table Title",
                            "OMS table subtype")
table.SimplePivotTable(cells = [1,2,3,4])
spss.EndProcedure()
```

spss.BasePivotTable Class

spss.BasePivotTable(title,templateName,outline,isSplit,caption). *Provides the ability to create custom pivot tables that can be displayed in the IBM® SPSS® Statistics Viewer or written to an external file using the SPSS Statistics 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*. Multiple tables within a given procedure can, however, have the same value of the *title* argument as long as they have different values of the *outline* argument.
- 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, you will not need to keep track of this value, although you do have to provide a value that meets the stated requirements.

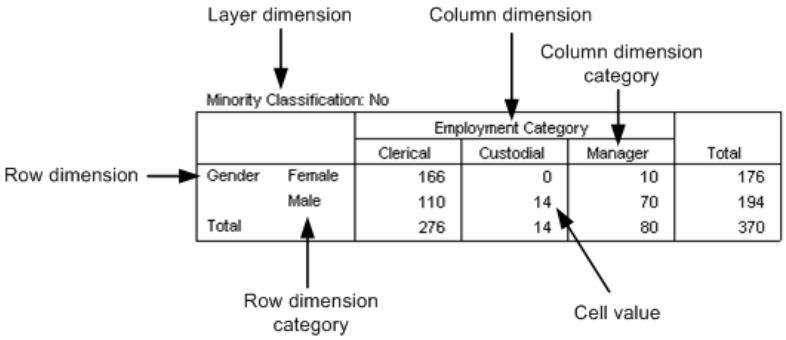
  Note: Specifying "Warnings" for *templateName* will generate an SPSS Statistics Warnings table. Unless you want to generate an SPSS Statistics Warnings table, you should avoid specifying "Warnings" for *templateName*. For more information, see the topic Creating a Warnings Table on p. 42.
- The optional 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. 18. For an example of creating a pivot table using a class based on the `spss.BaseProcedure` class, see spss.BaseProcedure Class on p. 43.

Figure 2-1 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 2-1
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 the topic General Approach to Creating Pivot Tables on p. 20.

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 2-2
Simple pivot table

Table Title

	column dimension	
row 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.

  Note: In order that names associated with output do not conflict with names of existing IBM® SPSS® Statistics commands (when working with OMS), it is recommended that they have the form *yourcompanyname.com.procedurename*. For more information, see the topic spss.StartProcedure Function on p. 109.
- 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 Statistics 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 `GeneralStat`. 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 the topic Using Cell Values in Expressions on p. 24.

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 2-3
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 2-4
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. 22.

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 unique category values, 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 2-5
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. Within a given dimension, the specified category values must be unique. 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 `GeneralStat`. 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.Correlation)`. For more information, see the topic Number Class on p. 38.

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 `GeneralStat`. 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.Correlation)`. For more information, see the topic Number Class on p. 38.

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

The `BasePivotTable` class has methods that allow you to build complex pivot tables. If you only need to create a pivot table with a single row and a single column dimension then consider using the `SimplePivotTable` method.

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. Each dimension must have a unique name. 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 2-6
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 the topic General Approach to Creating Pivot Tables on p. 20.

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")
```

CategoryFootnotes Method

.CategoryFootnotes(dimPlace,dimName,category,footnote). *Used to add a footnote to a specified category.*

- The argument *dimPlace* specifies the dimension type associated with the category, 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 argument *dimName* is the string that specifies the dimension name associated with the category. This is the name specified when the dimension was created by the `Append` or `Insert` method.
- The argument *category* specifies the category and 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

```
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.CategoryFootnotes(spss.Dimension.Place.row,"row dimension",
                        row_cat1,"A category footnote")
```

DimensionFootnotes Method

.DimensionFootnotes(dimPlace,dimName,footnote). *Used to add a footnote to a dimension.*

- The argument *dimPlace* 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 argument *dimName* is the string that specifies the name given to this dimension when it was created by the `Append` or `Insert` method.
- The argument *footnote* is a string specifying the footnote.

Example

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

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

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 Method

.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 2-1 on p. 39. 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 `GeneralStat`.

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. Each dimension must have a unique name. 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`.

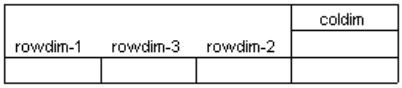
  Note: The order in which categories should be specified is not determined by dimension positions as specified by the argument *i*.

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 2-7
Resulting table structure



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 the topic General Approach to Creating Pivot Tables on p. 20.

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 unique 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 the topic General Approach to Creating Pivot Tables on p. 20.

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 `GeneralStat`. 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.Correlation)`. For more information, see the topic Number Class on p. 38.

Figure 2-8
Resulting table

	coldim	
rowdim	column1	column2
row1	11	12
row2		22

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 `GeneralStat`. 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.Correlation)`. For more information, see the topic Number Class on p. 38.

Figure 2-9
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 the topic General Approach to Creating Pivot Tables on p. 20.

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 `GeneralStat`. 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.Correlation)`. For more information, see the topic Number Class on p. 38.

Figure 2-10
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 the topic General Approach to Creating Pivot Tables on p. 20.

SetDefaultFormatSpec Method

.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 2-1 on p. 39—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 `GeneralStat`.

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. If specified, it must be distinct from the value, if any, of the *coldim* argument.
- **rowlabels.** An optional list of items to label the rows. Each item must be unique and 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. If specified, it must be distinct from the value, if any, of the *rowdim* argument.
- **collabels.** An optional list of items to label the columns. Each item must be unique and 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, a CellText object (one of `CellText.Number`, `CellText.String`, `CellText.VarName`, or `CellText.VarValue`), a Python `time.struct_time` object, or a Python `datetime.datetime` object. 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 `GeneralStat`.
- String values specified in *cells*, *rowlabels*, or *collabels* will be converted to CellText.String objects.
- When specifying cell values with Python `time.struct_time` or `datetime.datetime` objects, the value will be displayed in seconds—specifically, the number of seconds from October 14, 1582. You can change the format of a cell to a datetime format using the `SetNumericFormatAt` Python Scripting method, described in *Scripting Guide for IBM® SPSS® Statistics.pdf*. This requires embedding Python Scripting code within your Python program. For more information, see Scripting Facility in the Help system.

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 2-11
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 2-12
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 2-13
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

TitleFootnotes Method

.TitleFootnotes(footnote). *Used to add a footnote to the table title.* The argument *footnote* is a string specifying the footnote.

Example

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

table.TitleFootnotes("A title footnote")
```

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. You can also pass a string representation of a numeric value, a Python `time.struct_time` object, or a Python `datetime.datetime` object to this argument. 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 `GeneralStat`.
- 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.
- When specifying cell values with Python `time.struct_time` or `datetime.datetime` objects, the value will be displayed in seconds—specifically, the number of seconds from October 14, 1582. You can change the format of a cell to a datetime format using the `SetNumericFormatAt` Python Scripting method, described in *Scripting Guide for IBM® SPSS® Statistics.pdf*. This requires embedding Python Scripting code within your Python program. For more information, see Scripting Facility in the Help system.

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 2-1
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
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 also pass a numeric value, and it will be converted to a string.

- 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")`.

Creating a Warnings Table

You can create an IBM® SPSS® Statistics Warnings table using the `BasePivotTable` class by specifying `"Warnings"` for the *templateName* argument. Note that an SPSS Statistics Warnings table has a very specific structure, so unless you actually want a Warnings table you should avoid using `"Warnings"` for *templateName*.

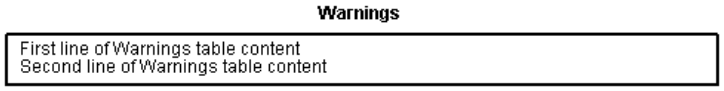
Example

```
import spss
spss.StartProcedure("demo")
table = spss.BasePivotTable("Warnings ","Warnings")
table.Append(spss.Dimension.Place.row,"rowdim",hideLabels=True)
rowLabel = spss.CellText.String("1")
table[(rowLabel,)] = spss.CellText.String("""First line of Warnings table content
Second line of Warnings table content""")
```

- The *title* argument is set to the string “Warnings “, It can be set to an arbitrary value but it cannot be identical to the *templateName* value, hence the space at the end of the string.
- The *templateName* argument must be set to the string “Warnings”, independent of the SPSS Statistics output language.
- A Warnings table has a single row dimension with all labels hidden and can consist of one or more rows. In this example, the table has a single multi-line row, formatted with a Python triple-quoted string.

Result

Figure 2-14
Warnings table



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 IBM® SPSS® Statistics Viewer. Procedures have almost the same capabilities as built-in SPSS Statistics 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 and the optional OMS identifier. 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 the topic spss.StartProcedure Function on p. 109.

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.omsIdentifier = ""
        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={};Totals={}

        #Calculate group sums
        for i in range(cur.GetCaseCount()):
            row = cur.fetchone()
            cat=int(row[0])
            Counts[cat]=Counts.get(cat,0) + 1
            Totals[cat]=Totals.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(Totals[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 `spss.BaseProcedure` class allows for an optional *omsIdentifier* property that specifies the command name associated with output from this procedure when routing the output with OMS (Output Management System), as used in the `COMMANDS` keyword of the `OMS` command. If *omsIdentifier* is an empty string then the value of *procName* is used as the OMS

identifier. Although specifying a non-blank value of the *omsIdentifier* property is optional, the property itself must be included.

Note:

- 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 */Python27/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='/examples/data/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 */examples/data/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). In order that names associated with procedure output not conflict with names of existing SPSS Statistics commands (when working with OMS), it is recommended that they have the form *yourcompanyname.com.procedurename*. For more information, see the topic spss.StartProcedure Function on p. 109.

Result

Figure 2-15
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.

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(var, accessType, cvtDates). *Provides the ability to read cases, append cases, and add new variables to the active dataset.*

- The optional argument *var* 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.
- 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 *cvtDates* specifies a set of IBM® SPSS® Statistics variables with date or datetime formats to convert to Python `datetime.datetime` objects when reading data from SPSS Statistics. The argument is a sequence of variable index values representing position in the active dataset, starting with 0 for the first variable in file order. If the optional argument *var* is specified, then *cvtDates* must be a subset of the index values specified for *var*. You can specify to convert all date or datetime format variables with `cvtDates='ALL'`, or by

setting *cvtDates* to a list or tuple with the single element `'ALL'`, as in `cvtDates=['ALL']`. When `'ALL'` is specified in conjunction with *var*, it refers to all variables specified in *var*. If *cvtDates* is omitted, then no conversions are performed. Variables included in *cvtDates* that do not have a date or datetime format are ignored in terms of the conversion. *cvtDates* applies to read and write mode and cannot be used in append mode.

Note: Values of variables with date or datetime formats that are not converted with *cvtDates* are returned as integers representing the number of seconds from October 14, 1582.

- 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. If you need to concurrently work with the data from multiple datasets, consider using the `Dataset` class.
- 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

This is the default for the `Cursor` class and provides the ability to read case data from the active dataset. Read mode is specified with `spss.Cursor(accessType='r')` or simply `spss.Cursor()`.

Note: For users of a 14.0.x version of the plug-in who are upgrading to a newer 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 a newer 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 *var* to the `Cursor` class, or file order if *var* is omittted.

Example: Missing Values

```
*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

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. Write mode is specified with `spss.Cursor(accessType='w')`.

- 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: Setting Values for Specific Cases

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: Multiple Data Passes

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

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. Append mode is specified with `spss.Cursor(accessType='a')`.

- 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
GetDataFileAttributeNames	X	X	X
GetDataFileAttributes	X	X	X
GetMultiResponseSetNames	X	X	X
GetMultiResponseSet	X	X	X
GetVarAttributeNames	X	X	X
GetVarAttributes	X	X	X
GetVariableCount	X	X	X
GetVariableFormat	X	X	X
GetVariableLabel	X	X	X

Method	Read mode	Write mode	Append mode
GetVariableMeasurementLevel	X	X	X
GetVariableName	X	X	X
GetVariableRole	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	
SetVarCMissingValues		X	
SetVarCValueLabel		X	
SetVarFormat		X	
SetVarLabel		X	
SetVarMeasureLevel		X	
SetVarNameAndType		X	
SetVarNMissingValues		X	
SetVarNValueLabel		X	
SetVarRole		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. 48. For an example of using `CommitCase` in append mode, see the topic on append mode on p. 51.

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. 51.

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.
- Values of variables with time formats are returned as integers representing the number of seconds from midnight.
- By default, values of variables with date or datetime formats are returned as integers representing the number of seconds from October 14, 1582. You can specify to convert values of those variables to Python `datetime.datetime` objects with the *cvtDates* argument to the `spss.Cursor` function. For more information, see the topic spss.Cursor Class on p. 46.
- If a weight variable has been defined for the active dataset, then cases with zero, negative, or missing values for the weighting variable are skipped when fetching data with `fetchone`, `fetchall`, or `fetchmany`. If you need to retrieve all cases when weighting is in effect, then you can use the Dataset class.

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.
- Values of variables with time formats are returned as integers representing the number of seconds from midnight.
- By default, values of variables with date or datetime formats are returned as integers representing the number of seconds from October 14, 1582. You can specify to convert values of those variables to Python `datetime.datetime` objects with the *cvtDates* argument to the `spss.Cursor` function. For more information, see the topic spss.Cursor Class on p. 46.
- If a weight variable has been defined for the active dataset, then cases with zero, negative, or missing values for the weighting variable are skipped when fetching data with `fetchone`, `fetchall`, or `fetchmany`. If you need to retrieve all cases when weighting is in effect, then you can use the Dataset class.

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.
- Values of variables with time formats are returned as integers representing the number of seconds from midnight.
- By default, values of variables with date or datetime formats are returned as integers representing the number of seconds from October 14, 1582. You can specify to convert values of those variables to Python `datetime.datetime` objects with the *cvtDates* argument to the `spss.Cursor` function. For more information, see the topic spss.Cursor Class on p. 46.
- If a weight variable has been defined for the active dataset, then cases with zero, negative, or missing values for the weighting variable are skipped when fetching data with `fetchone`, `fetchall`, or `fetchmany`. If you need to retrieve all cases when weighting is in effect, then you can use the Dataset class.

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='/examples/data/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. 48.

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 Python `time.struct_time` or `datetime.datetime` objects, which are then converted to the appropriate IBM® SPSS® Statistics value. Values of variables with `TIME` and `DTIME` formats should be specified as the number of seconds in the time interval.

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 available format types (see Appendix A on p. 116). 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 Variable Types and Formats in the Universals section of the *Command Syntax Reference*, available in PDF from the Help menu and also integrated into the overall Help system.

- This method is only available in write mode.
- Setting the argument *width* for a string variable will not change the defined length of the string. If the specified value does not match the defined length, it is forced to be the defined length.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['numvar'],[0])
cur.SetVarFormat('numvar',5,2,0)
cur.CommitDictionary()
cur.close()
```

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 0 for a discrete list of missing values (for example, 0, 9, 99), the value 1 for a range of missing values (for example, 9–99), and the value 2 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()
```

SetVarRole Method

.SetVarRole(varName,varRole). *Sets the role for a new variable.* The argument *varName* is a string specifying the name of a new variable. The argument *varRole* is a string specifying the role: “Input”, “Target”, “Both”, “None”, “Partition” or “Split”.

- This method is only available in write mode.

Example

```
cur=spss.Cursor(accessType='w')
cur.SetVarNameAndType(['targetvar'],[0])
cur.SetVarRole('targetvar','Target')
cur.CommitDictionary()
cur.close()
```

spss.Dataset Class

spss.Dataset(name,hidden,cvtDates). *Provides the ability to create new datasets, read from existing datasets, and modify existing datasets.* A `Dataset` object provides access to the case data and variable information contained in a dataset, and allows you to read from the dataset, add new cases, modify existing cases, add new variables, and modify properties of existing variables.

An instance of the `Dataset` class can only be created within a data step or `StartProcedure-EndProcedure` block, and cannot be used outside of the data step or procedure block in which it was created. Data steps are initiated with the `spss.StartDataStep` function. You can also use the `spss.DataStep` class to implicitly start and end a data step without the need to check for pending transformations. For more information, see the topic spss.DataStep Class on p. 90.

- The argument *name* is optional and specifies the name of an open dataset for which a `Dataset` object will be created. Note that this is the name as assigned by IBM® SPSS® Statistics or as specified with `DATASET NAME`. Specifying `name="*"` or omitting the argument will create a `Dataset` object for the active dataset. If the active dataset is unnamed, a name will be automatically generated for it. This is true regardless of whether the `Dataset` object is for the active dataset.
- If the Python data type *None* or the empty string `''` is specified for *name*, then a new empty dataset is created. The name of the dataset is automatically generated and can be retrieved from the `name` property of the resulting `Dataset` object. The name cannot be changed from within the data step. To change the name, use the `DATASET NAME` command following `spss.EndDataStep`.

  A new dataset created with the `Dataset` class is not set to be the active dataset. To make the dataset the active one, use the `spss.SetActive` function.

- The optional argument *hidden* specifies whether the Data Editor window associated with the dataset is hidden—by default, it is displayed. Use `hidden=True` to hide the associated Data Editor window.
- The optional argument *cvtDates* specifies whether SPSS Statistics variables with date or datetime formats are converted to Python `datetime.datetime` objects when reading data from SPSS Statistics. The argument is a boolean—*True* to convert all variables with date or datetime formats, *False* otherwise. If *cvtDates* is omitted, then no conversions are performed.

  Note: Values of variables with date or datetime formats that are not converted with *cvtDates* are returned as integers representing the number of seconds from October 14, 1582.
- Instances of the `Dataset` class created within `StartProcedure-EndProcedure` blocks cannot be set as the active dataset.

The number of variables in the dataset associated with a `Dataset` instance is available using the `len` function, as in:

```
len(datasetObj)
```

Note: Datasets that are not required outside of the data step or procedure in which they were accessed or created should be closed prior to ending the data step or procedure in order to free the resources allocated to the dataset. This is accomplished by calling the `close` method of the `Dataset` object.

Example: Creating a New Dataset

```
*python_dataset_new.sps.
BEGIN PROGRAM.
import spss
spss.StartDataStep()
datasetObj = spss.Dataset(name=None)
datasetObj.varlist.append('numvar',0)
datasetObj.varlist.append('strvar',1)
datasetObj.varlist['numvar'].label = 'Sample numeric variable'
datasetObj.varlist['strvar'].label = 'Sample string variable'
datasetObj.cases.append([1,'a'])
datasetObj.cases.append([2,'b'])
spss.EndDataStep()
END PROGRAM.
```

- You add variables to a dataset using the `append` (or `insert`) method of the `VariableList` object associated with the dataset. The `VariableList` object is accessed from the `varlist` property of the `Dataset` object, as in `datasetObj.varlist`. For more information, see the topic VariableList Class on p. 82.
- Variable properties, such as the variable label and measurement level, are set through properties of the associated `Variable` object, accessible from the `VariableList` object. For example, `datasetObj.varlist['numvar']` accesses the `Variable` object associated with the variable *numvar*. For more information, see the topic Variable Class on p. 84.
- You add cases to a dataset using the `append` (or `insert`) method of the `CaseList` object associated with the dataset. The `CaseList` object is accessed from the `cases` property of the `Dataset` object, as in `datasetObj.cases`. For more information, see the topic CaseList Class on p. 77.

Example: Saving New Datasets

When creating new datasets that you intend to save, you'll want to keep track of the dataset names since the save operation is done outside of the associated data step.

```
*python_dataset_save.sps.
DATA LIST FREE /dept (F2) empid (F4) salary (F6).
BEGIN DATA
7  57  57000
5  23  40200
3  62  21450
3  18  21900
5  21  45000
5  29  32100
7  38  36000
3  42  21900
7  11  27900
END DATA.
DATASET NAME saldata.
SORT CASES BY dept.
BEGIN PROGRAM.
from __future__ import with_statement
import spss
with spss.DataStep():
   ds = spss.Dataset()
   # Create a new dataset for each value of the variable 'dept'
   newds = spss.Dataset(name=None)
   newds.varlist.append('dept')
   newds.varlist.append('empid')
   newds.varlist.append('salary')
   dept = ds.cases[0,0][0]
   dsNames = {newds.name:dept}
   for row in ds.cases:
      if (row[0] != dept):
         newds = spss.Dataset(name=None)
         newds.varlist.append('dept')
         newds.varlist.append('empid')
         newds.varlist.append('salary')
         dept = row[0]
         dsNames[newds.name] = dept
      newds.cases.append(row)
# Save the new datasets
for name,dept in dsNames.iteritems():
   strdept = str(dept)
   spss.Submit(r"""
   DATASET ACTIVATE %(name)s.
   SAVE OUTFILE='/mydata/saldata_%(strdept)s.sav'.
   """ %locals())
spss.Submit(r"""
DATASET ACTIVATE saldata.
DATASET CLOSE ALL.
""" %locals())
END PROGRAM.
```

- The code `newdsObj = spss.Dataset(name=None)` creates a new dataset. The name of the dataset is available from the *name* property, as in `newdsObj.name`. In this example, the names of the new datasets are stored to the Python dictionary *dsNames*.
- To save new datasets created with the `Dataset` class, use the `SAVE` command after calling `spss.EndDataStep`. In this example, `DATASET ACTIVATE` is used to activate each new dataset, using the dataset names stored in *dsNames*.

Example: Modifying Case Values

```
*python_dataset_modify_cases.sps.
DATA LIST FREE /cust (F2) amt (F5).
BEGIN DATA
210 4500
242 6900
370 32500
END DATA.
BEGIN PROGRAM.
import spss
spss.StartDataStep()
datasetObj = spss.Dataset()
for i in range(len(datasetObj.cases)):
   # Multiply the value of amt by 1.05 for each case
   datasetObj.cases[i,1] = 1.05*datasetObj.cases[i,1][0]
spss.EndDataStep()
END PROGRAM.
```

- The `CaseList` object, accessed from the `cases` property of a `Dataset` object, allows you to read or modify case data. To access the value for a given variable within a particular case you specify the case number and the index of the variable (index values represent position in the active dataset, starting with 0 for the first variable in file order, and case numbers start from 0). For example, `datasetObj.cases[i,1]` specifies the value of the variable with index `1` for case number `i`.
- When reading case values, results are returned as a list. In the present example we're accessing a single value within each case so the list has one element.

For more information, see the topic CaseList Class on p. 77.

Example: Comparing Datasets

`Dataset` objects allow you to concurrently work with the case data from multiple datasets. As a simple example, we'll compare the cases in two datasets and indicate identical cases with a new variable added to one of the datasets.

```
*python_dataset_compare.sps.
DATA LIST FREE /id (F2) salary (DOLLAR8) jobcat (F1).
BEGIN DATA
1 57000 3
3 40200 1
2 21450 1
END DATA.
SORT CASES BY id.
DATASET NAME empdata1.
DATA LIST FREE /id (F2) salary (DOLLAR8) jobcat (F1).
BEGIN DATA
3 41000 1
1 59280 3
2 21450 1
END DATA.
SORT CASES BY id.
DATASET NAME empdata2.
BEGIN PROGRAM.
import spss
spss.StartDataStep()
datasetObj1 = spss.Dataset(name="empdata1")
datasetObj2 = spss.Dataset(name="empdata2")
nvars = len(datasetObj1)
datasetObj2.varlist.append('match')
for i in range(len(datasetObj1.cases)):
   if datasetObj1.cases[i] == datasetObj2.cases[i,0:nvars]:
      datasetObj2.cases[i,nvars] = 1
   else:
      datasetObj2.cases[i,nvars] = 0
spss.EndDataStep()
END PROGRAM.
```

- The two datasets are first sorted by the variable *id* which is common to both datasets.
- Since `DATA LIST` creates unnamed datasets (the same is true for `GET`), the datasets are named using `DATASET NAME` so that you can refer to them when calling `spss.Dataset`.
- `datasetObj1` and `datasetObj2` are `Dataset` objects associated with the two datasets *empdata1* and *empdata2* to be compared.
- The code `datasetObj1.cases[i]` returns case number `i` from *empdata1*. The code `datasetObj2.cases[i,0:nvars]` returns the slice of case number `i` from *empdata2* that includes the variables with indexes 0,1,...,*nvars*-1.
- The new variable *match*, added to *empdata2*, is set to 1 for cases that are identical and 0 otherwise.

cases Property

The `cases` property of a `Dataset` object returns an instance of the `CaseList` class. The `CaseList` class provides access to the cases in the associated dataset, allowing you to read existing cases, modify case values, and add new cases. For more information, see the topic CaseList Class on p. 77.

Example

```
import spss
spss.StartDataStep()
datasetObj = spss.Dataset('data1')
caseListObj = datasetObj.cases
```

```
spss.EndDataStep()
```

name Property

The `name` property of a `Dataset` object gets the name of the associated dataset. The name cannot be changed from within the data step. To change the name, use the `DATASET NAME` command following `spss.EndDataStep`.

Example

```
import spss
spss.StartDataStep()
datasetObj = spss.Dataset('data1')
datasetName = datasetObj.name
spss.EndDataStep()
```

varlist Property

The `varlist` property of a `Dataset` object returns an instance of the `VariableList` class. The `VariableList` class provides access to the variables in the associated dataset, allowing you to retrieve the properties of existing variables, modify variable properties, and add new variables to the dataset. For more information, see the topic VariableList Class on p. 82.

Example

```
import spss
spss.StartDataStep()
datasetObj = spss.Dataset('data1')
varListObj = datasetObj.varlist
spss.EndDataStep()
```

dataFileAttributes Property

The `dataFileAttributes` property of a `Dataset` object gets or sets datafile attributes for the dataset. The `dataFileAttributes` property behaves like a Python dictionary in terms of getting, setting, and deleting values. 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 case of datafile attributes, each key is the name of an attribute and the associated value is the value of the attribute, which can be a single value or a list or tuple of values. A list or tuple of values specifies an attribute array.

- When setting attributes, attribute names and values must be given as quoted strings.

Retrieving Datafile Attributes. You retrieve datafile attributes for a dataset from the `dataFileAttributes` property of the associated `Dataset` object. You can retrieve the value of a particular attribute by specifying the attribute name, as in:

```
dsObj = spss.Dataset()
attr = dsObj.dataFileAttributes['attrName']
```

Attribute values are always returned as a tuple.

You can iterate through the set of datafile attributes using the `data` property, as in:

```
dsObj = spss.Dataset()
for attrName, attrValue in dsObj.dataFileAttributes.data.iteritems():
   print attrName, attrValue
```

Adding and Modifying Datafile Attributes. You can add new datafile attributes and modify existing ones. For example:

```
dsObj.dataFileAttributes['attrName'] = 'value'
```

- If the attribute *attrName* exists, it is updated with the specified value. If the attribute *attrName* doesn't exist, it is added to any existing ones for the dataset.

Resetting Datafile Attributes. You can reset the datafile attributes associated with a dataset. For example:

```
dsObj.dataFileAttributes = {'attr1':'value','attr2':['val1','val2']}
```

- You reset the datafile attributes by setting the *dataFileAttributes* property to a new Python dictionary. Any existing datafile attributes are cleared and replaced with the specified ones.

Deleting Datafile Attributes. You can delete a particular datafile attribute or all of them. For example:

```
#Delete a specified attribute
del dsObj.dataFileAttributes['attrName']
#Delete all attributes
del dsObj.dataFileAttributes
```

multiResponseSet Property

The `multiResponseSet` property of a `Dataset` object gets or sets multiple response sets for the dataset. The `multiResponseSet` property behaves like a Python dictionary in terms of getting, setting, and deleting values. 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 case of multiple response sets, each key is the name of a set and the associated value specifies the details of the set.

- The multiple response set name is a string of maximum length 63 bytes that must follow IBM® SPSS® Statistics variable naming conventions. If the specified name does not begin with a dollar sign ($), then one is added. If the name refers to an existing set, the set definition is overwritten.
- When setting a multiple response set, the details of the set are specified as a list or tuple with the following elements in the presented order.

  mrsetLabel. A string specifying a label for the set. The value cannot be wider than the limit for SPSS Statistics variable labels.

  mrsetCodeAs. An integer or string specifying the variable coding: 1 or "Categories" for multiple category sets, 2 or "Dichotomies" for multiple dichotomy sets.

  mrsetCountedValue. A string specifying the value that indicates the presence of a response for a multiple dichotomy set. This is also referred to as the "counted" value. If the set type is numeric, the value must be a string representation of an integer. If the set type is string, the

counted value, after trimming trailing blanks, cannot be wider than the narrowest elementary variable.

varNames. A tuple or list of strings specifying the names of the elementary variables that define the set (the list must include at least two variables).

- When getting a multiple response set, the result is a tuple of 5 elements. The first element is the label, if any, for the set. The second element specifies the variable coding—'Categories' or 'Dichotomies'. The third element specifies the counted value and only applies to multiple dichotomy sets. The fourth element specifies the data type—'Numeric' or 'String'. The fifth element is a list of the elementary variables that define the set.

Retrieving Multiple Response Sets. You retrieve multiple response sets for a dataset from the `multiResponseSet` property of the associated `Dataset` object. You retrieve the value of a particular set by specifying the set name, as in:

```
dsObj = spss.Dataset()
mrset = dsObj.multiResponseSet['setName']
```

You can iterate through the multiple response sets using the `data` property, as in:

```
dsObj = spss.Dataset()
for name, set in dsObj.multiResponseSet.data.iteritems():
   print name, set
```

Adding and Modifying Multiple Response Sets. You can add new multiple response sets and modify details of existing ones. For example:

```
dsObj.multiResponseSet['$mltnews'] = \
                        ["News Sources",2,"1",["Newspaper","TV","Web"]]
```

- If the set *$mltnews* exists, it is updated with the specified values. If the set *$mltnews* doesn't exist, it is added to any existing ones for the dataset.

Resetting Multiple Response Sets. You can reset the multiple response sets associated with a dataset. For example:

```
dsObj.multiResponseSet = \
{'$mltnews':["News Sources",2,"1",["Newspaper","TV","Web"]],
 '$mltent':["Entertainment Sources",2,"1",["TV","Movies","Theatre","Music"]]}
```

- You reset the multiple response sets by setting the *multiResponseSet* property to a new Python dictionary. Any existing multiple response sets are cleared and replaced with the specified ones.

Deleting Multiple Response Sets. You can delete a particular multiple response set or all sets. For example:

```
#Delete a specified set
del dsObj.multiResponseSet['setName']
#Delete all sets
del dsObj.multiResponseSet
```

close Method

.close(). *Closes the dataset.* This method closes a dataset accessed through or created by the `Dataset` class. It cannot be used to close an arbitrary open dataset. When used, it must be called prior to `EndDataStep` or `EndProcedure`.

- If the associated dataset is not the active dataset, that dataset is closed and no longer available in the session. The associated dataset will, however, remain open outside of the data step or procedure in which it was created if the `close` method is not called.
- If the associated dataset is the active dataset, the association with the dataset's name is broken. The active dataset remains active but has no name.

Note: Datasets that are not required outside of the data step or procedure in which they were accessed or created should be closed prior to ending the data step or procedure in order to free the resources allocated to the dataset.

Example

```
import spss
spss.StartDataStep()
datasetObj1 = spss.Dataset()
datasetObj2 = datasetObj1.deepCopy(name="copy1")
datasetObj1.close()
spss.EndDataStep()
```

deepCopy Method

.deepCopy(name). *Creates a copy of the Dataset instance as well as a copy of the dataset associated with the instance.* The argument is required and specifies the name of the new dataset, as a quoted string. The name cannot be the name of the dataset being copied or a blank string. If `'*'` is specified the copy becomes the active dataset with a name that is automatically generated. You can retrieve the dataset name from the `name` property of the new `Dataset` instance.

Example

```
import spss
spss.StartDataStep()
datasetObj1 = spss.Dataset()
# Make a copy of the active dataset and assign it the name "copy1"
datasetObj2 = datasetObj1.deepCopy(name="copy1")
spss.EndDataStep()
```

CaseList Class

The `CaseList` class provides access to the cases in a dataset, allowing you to read existing cases, modify case values, and add new cases. You get an instance of the `CaseList` class from the `cases` property of the `Dataset` class, as in:

```
datasetObj = spss.Dataset('data1')
caseListObj = datasetObj.cases
```

The number of cases in a `CaseList` instance, which is also the number of cases in the associated dataset, is available using the `len` function, as in:

```
len(caseListObj)
```

Note: An instance of the `CaseList` class can only be created within a data step, and cannot be used outside of the data step in which it was created. Data steps are initiated with the `spss.StartDataStep` function.

Looping through the cases in an instance of CaseList. You can loop through the cases in an instance of the `CaseList` class. For example:

```
for row in datasetObj.cases:
   print row
```

- On each iteration of the loop, *row* is a case from the associated dataset.

Note: The `CaseList` class does not provide any special handling for datasets with split groups—it simply returns all cases in the dataset. If you need to differentiate the data in separate split groups, consider using the `Cursor` class to read your data, or you may want to use the `spss.GetSplitVariableNames` function to manually process the split groups.

Accessing specific cases and case values. You can access a specific case or a range of cases, and you can specify a variable or a range of variables within those cases. The result is a list, even if accessing the value of a single variable within a single case.

- System-missing values are returned as the Python data type *None*.
- Values of variables with `TIME` and `DTIME` formats are returned as integers representing the number of seconds in the time interval.
- By default, values of variables with date or datetime formats are returned as integers representing the number of seconds from October 14, 1582. You can specify to convert values of those variables to Python `datetime.datetime` objects with the *cvtDates* argument to the `Dataset` class. For more information, see the topic spss.Dataset Class on p. 69.

Example: Accessing a Single Case

Case values are accessed by specifying the case number, starting with 0, as in:

```
oneCase = datasetObj.cases[0]
```

Case values are returned as a list where each element of the list is the value of the associated variable.

Example: Accessing a Single Value Within a Case

You can access the value for a single variable within a case by specifying the case number and the index of the variable (index values represent position in the active dataset, starting with 0 for the first variable in file order). The following gets the value of the variable with index 1 for case number 0.

```
oneValue = datasetObj.cases[0,1]
```

Note that *oneValue* is a list with a single element.

Example: Accessing a Range of Values

You can use the Python slice notation to specify ranges of cases and ranges of variables within a case. Values for multiple cases are returned as a list of elements, each of which is a list of values for a single case.

```
# Get the values for cases 0,1, and 2
data = datasetObj.cases[0:3]

# Get the values for variables with index values 0,1, and 2
# for case number 0
data = datasetObj.cases[0,0:3]

# Get the value for the variable with index 1 for case numbers 0,1, and 2
data = datasetObj.cases[0:3,1]

# Get the values for the variables with index values 1,2 and 3
# for case numbers 4,5, and 6
data = datasetObj.cases[4:7,1:4]
```

Example: Negative Index Values

Case indexing supports the use of negative indices, both for the case number and the variable index. The following gets the value of the second to last variable (in file order) for the last case.

```
value = datasetObj.cases[-1,-2]
```

Modifying case values. You can modify the values for a specific case or a range of cases, and you can set the value of a particular variable or a range of variables within those cases.

- Values of *None* are converted to system-missing for numeric variables and blanks for string variables.
- Values of numeric variables with a date or datetime format should be specified as Python `time.struct_time` or `datetime.datetime` objects, which are then converted to the appropriate IBM® SPSS® Statistics value. Values of variables with `TIME` and `DTIME` formats should be specified as the number of seconds in the time interval.

Example: Setting Values for a Single Case

Values for a single case are provided as a list or tuple of values. The first element corresponds to the first variable in file order, the second element corresponds to the second variable in file order, and so on. Case numbers start from 0.

```
datasetObj.cases[1] = [35,150,100,2110,19,2006,3,4]
```

Example: Setting a Single Value Within a Case

You can set the value for a single variable within a case by specifying the case number and the index of the variable (index values represent position in the active dataset, starting with 0 for the first variable in file order). The following sets the value of the variable with index 0 for case number 12 (case numbers start from 0).

```
datasetObj.cases[12,0] = 14
```

Example: Setting Ranges of Values

You can use the Python slice notation to specify ranges of cases and ranges of variables within a case. Values for multiple cases are specified as a list or tuple of elements, each of which is a list or tuple of values for a single case.

```
# Set the values for cases 0,1, and 2
datasetObj.cases[0:3] = ([172,'m',27,34500],[67,'f',32,32500],
                         [121,'f',37,23000])


# Set the values for variables with index values 5,6, and 7 for
# case number 34
datasetObj.cases[34,5:8] = [70,1,4]


# Set the value for the variable with index 5 for case numbers 0,1, and 2
datasetObj.cases[0:3,5] = [70,72,71]


# Set the values for the variables with index values 5 and 6 for
# case numbers 4,5, and 6
datasetObj.cases[4:7,5:7] = ([70,1],[71,2],[72,2])
```

Example: Negative Index Values

Case indexing supports the use of negative indices, both for the case number and the variable index. The following specifies the value of the second to last variable (in file order) for the last case.

```
datasetObj.cases[-1,-2] = 8
```

Deleting cases. You can delete a specified case from the `CaseList` object, which results in deleting that case from the associated dataset. For example:

```
del datasetObj.cases[0]
```

append Method

.append(case). *Appends a new case to the associated dataset and appends an element representing the case to the corresponding CaseList instance.* The argument *case* is a tuple or list specifying the case values. The first element in the tuple or list is the value for the first variable in file order, the second is the value of the second variable in file order and so on.

- The elements of *case* can be numeric or string values and must match the variable type of the associated variable. Values of *None* are converted to system-missing for numeric variables and blanks for string variables.
- Values of numeric variables with a date or datetime format should be specified as Python `time.struct_time` or `datetime.datetime` objects, which are then converted to the appropriate IBM® SPSS® Statistics value. Values of variables with `TIME` and `DTIME` formats should be specified as the number of seconds in the time interval.

Example

```
*python_dataset_append_case.sps.
DATA LIST FREE/numvar (F2) strvar (A1).
BEGIN DATA.
1 a
END DATA.
BEGIN PROGRAM.
import spss
spss.StartDataStep()
datasetObj = spss.Dataset()
# Append a single case to the active dataset
datasetObj.cases.append([2,'b'])
spss.EndDataStep()
END PROGRAM.
```

insert Method

.insert(case, caseNumber). *Inserts a new case into the associated dataset and inserts an element representing the case into the corresponding CaseList instance.* The argument *case* is a tuple or list specifying the case values. The first element in the tuple or list is the value for the first variable in file order, the second is the value of the second variable in file order and so on. The optional argument *caseNumber* specifies the location at which the case is inserted (case numbers start from 0) and can take on the values 0,1,...,*n* where *n* is the number of cases in the dataset. If *caseNumber* is omitted or equal to *n*, the case is appended.

- The elements of *case* can be numeric or string values and must match the variable type of the associated variable. Values of *None* are converted to system-missing for numeric variables and blanks for string variables.
- Values of numeric variables with a date or datetime format should be specified as Python `time.struct_time` or `datetime.datetime` objects, which are then converted to the appropriate IBM® SPSS® Statistics value. Values of variables with `TIME` and `DTIME` formats should be specified as the number of seconds in the time interval.

Example

```
*python_dataset_insert_case.sps.
DATA LIST FREE/numvar (F2) strvar (A1).
BEGIN DATA.
1 a
3 c
END DATA.
BEGIN PROGRAM.
import spss
spss.StartDataStep()
datasetObj = spss.Dataset()
# Insert a single case into the active dataset at case number 1
datasetObj.cases.insert([2,'b'],1)
spss.EndDataStep()
END PROGRAM.
```

VariableList Class

The `VariableList` class provides access to the variables in a dataset, allowing you to get and set properties of existing variables, as well as add new variables to the dataset. You get an instance of the `VariableList` class from the `varlist` property of the `Dataset` class, as in:

```
datasetObj = spss.Dataset('data1')
varListObj = datasetObj.varlist
```

The number of variables in a `VariableList` instance, which is also the number of variables in the associated dataset, is available using the `len` function, as in:

```
len(varListObj)
```

Note: An instance of the `VariableList` class can only be created within a data step, and cannot be used outside of the data step in which it was created. Data steps are initiated with the `spss.StartDataStep` function.

Looping through the variables in an instance of VariableList. You can loop through the variables in an instance of the `VariableList` class, obtaining a `Variable` object (representing the properties of a single variable) on each iteration. For example:

```
for var in datasetObj.varlist:
   print var.name
```

- On each iteration of the loop, *var* is an instance of the `Variable` class, representing a particular variable in the `VariableList` instance. The `Variable` class allows you to get and set variable properties, like the measurement level and missing values. For more information, see the topic Variable Class on p. 84.

Accessing a variable by name or index. You can obtain a `Variable` object for a specified variable in the `VariableList` instance. The desired variable can be specified by name or index. For example:

```
#Get variable by name
varObj = datasetObj.varlist['salary']
#Get variable by index
varObj = datasetObj.varlist[5]
```

Deleting a variable. You can delete a specified variable from the `VariableList` instance, which results in deleting it from the associated dataset. The variable to be deleted can be specified by name or index. For example:

```
#Delete variable by name
del datasetObj.varlist['salary']
#Delete variable by index
del datasetObj.varlist[5]
```

append Method

.append(name,type). *Appends a new variable to the associated dataset and appends a corresponding Variable object to the associated VariableList instance.* The argument *name* specifies the variable name. The argument *type* is optional and specifies the variable type—numeric or string. The default is numeric.

- 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).
- The properties of the new variable are set using the `Variable` object created by the `append` method. For more information, see the topic Variable Class on p. 84.

Example

```
*python_dataset_append_variable.sps.
DATA LIST FREE/numvar (F2).
BEGIN DATA.
1
END DATA.
BEGIN PROGRAM.
import spss
spss.StartDataStep()
datasetObj = spss.Dataset()
# Append a string variable of length 1 to the active dataset
datasetObj.varlist.append('strvar',1)
spss.EndDataStep()
END PROGRAM.
```

insert Method

.insert(name,type,index). *Inserts a new variable into the associated dataset and inserts a corresponding Variable object into the associated VariableList instance.* The argument *name* specifies the variable name. The optional argument *type* specifies the variable type—numeric or string. If *type* is omitted, the variable is numeric. The optional argument *index* specifies the position for the inserted variable and `Variable` object (the first position has an index value of 0) and can take on the values 0,1,...,*n* where *n* is the number of variables in the dataset. If *index* is omitted or equal to *n*, the variable is appended to the end of the list.

- 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).
- The properties of the new variable are set using the `Variable` object created by the `insert` method. For more information, see the topic Variable Class on p. 84.

Example

```
*python_dataset_insert_variable.sps.
DATA LIST FREE/var1 (F2) var3 (A1).
BEGIN DATA.
1 a
END DATA.
BEGIN PROGRAM.
import spss
spss.StartDataStep()
datasetObj = spss.Dataset()
# Insert a numeric variable at index position 1 in the active dataset
datasetObj.varlist.insert('var2',0,1)
spss.EndDataStep()
END PROGRAM.
```

Variable Class

The `Variable` class allows you to get and set the properties of a variable. Instances of the `Variable` class for each variable in the associated dataset are generated when the `VariableList` class is instantiated. In addition, the `append` and `insert` methods of a `VariableList` object create associated instances of the `Variable` class for appended and inserted variables. Specific variables can be accessed by name or index (index values represent position in the dataset, starting with 0 for the first variable in file order).

```
datasetObj = spss.Dataset('data1')
# Create a Variable object, specifying the variable by name
varObj = datasetObj.varlist['bdate']
# Create a Variable object, specifying the variable by index
varObj = datasetObj.varlist[3]
```

Note: An instance of the `Variable` class can only be created within a data step, and cannot be used outside of the data step in which it was created. Data steps are initiated with the `spss.StartDataStep` function.

alignment Property

The `alignment` property of a `Variable` object gets or sets the alignment of data values displayed in the Data Editor. 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 variable alignment is specified as an integer with one of the following values: 0 (left), 1 (right), 2 (center).

Example

```
varObj = datasetObj.varlist['gender']
#Get the variable alignment
align = varObj.alignment
#Set the variable alignment
varObj.alignment = 1
```

attributes Property

The `attributes` property of a `Variable` object gets or sets custom variable attributes. It can also be used to clear any custom attributes. The `attributes` property behaves like a Python dictionary in terms of getting, setting, and deleting values. 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 case of variable attributes, each key is the name of an attribute and the associated value is the value of the attribute, which can be a single value or a list or tuple of values. A list or tuple of values specifies an attribute array.

- When setting attributes, attribute names and values must be given as quoted strings.

Retrieving Variable Attributes. You retrieve custom variable attributes for a specified variable from the `attributes` property of the associated `Variable` object. You retrieve the value of a particular attribute by specifying the attribute name, as in:

```
varObj = datasetObj.varlist['gender']
attrValue = varObj.attributes['attrName']
```

Attribute values are always returned as a tuple.

You can iterate through the set of variable attributes using the `data` property, as in:

```
varObj = datasetObj.varlist['gender']
for attrName, attrValue in varObj.attributes.data.iteritems():
   print attrName, attrValue
```

Adding and Modifying Attributes. You can add new attributes and modify values of existing ones. For example:

```
varObj = datasetObj.varlist['age']
varObj.attributes['AnswerFormat'] = 'Fill-in'
```

- If the attribute *AnswerFormat* exists, its value is updated to 'Fill-in'. If the attribute *AnswerFormat* doesn't exist, it is added to any existing ones for the variable *age*.

Resetting Attributes. You can reset the attributes to a new specified set. For example:

```
varObj = datasetObj.varlist['gender']
varObj.attributes = {'DemographicVars':'1','Binary':'Yes'}
```

- You reset the attributes by setting the *attributes* property to a new Python dictionary. Any existing attributes for the variable are cleared and replaced with the specified set.

Deleting Attributes. You can delete a particular attribute or the entire set of attributes for a specified variable. For example:

```
varObj = datasetObj.varlist['gender']
#Delete the attribute Binary
del varObj.attributes['Binary']
#Delete all attributes
del varObj.attributes
```

columnWidth Property

The `columnWidth` property of a `Variable` object gets or sets the column width of data values displayed in the Data Editor. Changing the column width does not change the defined width of a variable. When setting the column width, the specified value must be a positive integer.

Example

```
varObj = datasetObj.varlist['prevexp']
#Get the column width
width = varObj.columnWidth
#Set the column width
varObj.columnWidth = 3
```

format Property

The `format` property of a `Variable` object gets or sets the display format of a variable.

Example

```
varObj = datasetObj.varlist['id']
#Get the variable format
```

```
format = varObj.format
#Set the variable format
varObj.format = (5,5,0)
```

- When getting the format, the returned value is a string consisting of a character portion (in upper case) that specifies the format type, followed by a numeric component that indicates the defined width, followed by a component that specifies the number of decimal positions and is only included for numeric formats. For example, A4 is a string format with a maximum width of four, and F8.2 is a standard numeric format with a display format of eight digits, including two decimal positions and a decimal indicator.
- When setting the format, you provide a tuple or list of three integers specifying the format type, width, and the number of decimal digits (for numeric formats) in that order. The width must include enough positions to accommodate any punctuation characters such as decimal points, commas, dollar signs, or date and time delimiters. If decimal digits do not apply, use 0 for the third element of the list or tuple. The available format types are listed in Appendix A on p. 116.

Notes

- 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 Variable Types and Formats in the Universals section of the *Command Syntax Reference*, available in PDF from the Help menu and also integrated into the overall Help system.
- Setting the width for a string variable will not change the defined length of the string. If the specified value does not match the defined length, it is forced to be the defined length.

index Property

The `index` property of a `Variable` object gets the variable index. The index value represents position in the dataset starting with 0 for the first variable in file order.

Example

```
varObj = datasetObj.varlist['bdate']
index = varObj.index
```

label Property

The `label` property of a `Variable` object gets or sets the variable label.

Example

```
varObj = datasetObj.varlist['bdate']
#Get the variable label
label = varObj.label
#Set the variable label
varObj.label = 'Birth Date'
```

measurementLevel Property

The `measurementLevel` property of a `Variable` object gets or sets the measurement level of a variable. The measurement level is specified as a string. When setting the measurement level the allowed values are: `"NOMINAL"`, `"ORDINAL"`, and `"SCALE"`. When getting the measurement level the additional value `"UNKNOWN"` may be returned 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

```
varObj = datasetObj.varlist['minority']
#Get the measurement level
level = varObj.measurementLevel
#Set the measurement level
varObj.measurementLevel = "NOMINAL"
```

missingValues Property

The `missingValues` property of a `Variable` object gets or sets user-missing values. The missing values are specified as a tuple or list of four elements where the first element specifies the missing value type: 0,1,2, or 3 for that number of discrete values, -2 for a range of values, and -3 for a range of values and a single discrete value. The remaining three elements specify the missing values. When getting missing values, the result is returned as a tuple with this same structure.

- For string variables, returned values are right-padded to the defined width of the string variable.
- To specify *LO* and *HI* in missing value ranges, use the values returned by the spss.GetSPSSLowHigh function.

The structure of the tuple or list that specifies the missing values is shown in the following table.

missingVals[0]	missingVals[1]	missingVals[2]	missingVals[3]
-3	Start point of range	End point of range	Discrete value
-2	Start point of range	End point of range	*None*
0	*None*	*None*	*None*
1	Discrete value	*None*	*None*
2	Discrete value	Discrete value	*None*
3	Discrete value	Discrete value	Discrete value

Examples

In the following examples, *varObj* is an instance of the `Variable` class.

Get the user-missing values.

```
missingVals = varObj.missingValues
```

Specify the discrete missing values 0 and 9 for a numeric variable.

```
varObj.missingValues = [2,0,9,None]
```

Specify the range of missing values 9–99 for a numeric variable.

```
varObj.missingValues = [-2,9,99,None]
```

Specify the range of missing values 9–99 and the discrete missing value 0 for a numeric variable.

```
varObj.missingValues = [-3,9,99,0]
```

Specify two missing values for a string variable.

```
varObj.missingValues = [2,' ','NA',None]
```

Clear all missing values

```
varObj.missingValues = [0,None,None,None]
```

name Property

The `name` property of a `Variable` object gets or sets the variable name.

Example

```
varObj = datasetObj.varlist['bdate']
#Get the variable name
name = varObj.name
#Set the variable name
varObj.name = 'birthdate'
```

role Property

The `role` property of a `Variable` object gets or sets the variable role. Valid values for getting and setting are the following strings: “Input”, “Target”, “Both”, “None”, “Partition” or “Split”.

Example

```
varObj = datasetObj.varlist['var1']
#Get the variable role
role = varObj.role
#Set the variable role
varObj.role = 'Target'
```

type Property

The `type` property of a `Variable` object gets or sets the variable type—numeric or string. The variable type for numeric variables is 0. The variable type for string variables is an integer equal to the defined length of the string (maximum of 32767).

Example

```
varObj = datasetObj.varlist['strvar']
#Get the variable type
type = varObj.type
#Set the variable type to a string of length 10
varObj.type = 10
```

valueLabels Property

The `valueLabels` property of a `Variable` object gets or sets value labels. It can also be used to clear any value labels. The `valueLabels` property behaves like a Python dictionary in terms of getting, setting, and deleting values. 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 case of value labels, each key is a value and the associated value is the label.

- When setting value labels for string variables, values must be specified as quoted strings.

Retrieving Value Labels. You retrieve value labels for a specified variable from the `valueLabels` property of the associated `Variable` object. You retrieve the label for a particular value by specifying the value, as in the following, which retrieves the label for the value 1:

```
varObj = datasetObj.varlist['origin']
valLab = varObj.valueLabels[1]
```

You can iterate through the set of value labels for a variable using the `data` property, as in:

```
varObj = datasetObj.varlist['origin']
for val, valLab in varObj.valueLabels.data.iteritems():
   print val, valLab
```

Adding and Modifying Value Labels. You can add new value labels and modify existing ones. For example:

```
varObj = datasetObj.varlist['origin']
varObj.valueLabels[4] = 'Korean'
```

- If a label for the value 4 exists, its value is updated to ‘Korean’. If a label for the value 4 doesn’t exist, it is added to any existing value labels for the variable *origin*.

Resetting Value Labels. You can reset the value labels to a new specified set. For example:

```
varObj = datasetObj.varlist['origin']
varObj.valueLabels = {1:'American',2:'Japanese',3:'European',
                      4:'Korean',5:'Chinese'}
```

- You reset the value labels by setting the *valueLabels* property to a new Python dictionary. Any existing value labels for the variable are cleared and replaced with the specified set.

Deleting Value Labels. You can delete a particular value label or the entire set of value labels for a specified variable. For example:

```
varObj = datasetObj.varlist['origin']
#Delete the value label for the value 1
del varObj.valueLabels[1]
#Delete all value labels
del varObj.valueLabels
```

spss.DataStep Class

The DataStep class implicitly starts and ends a data step without the need to explicitly call StartDataStep and EndDataStep. In addition, it executes any pending transformations, eliminating the need to check for them prior to starting a data step. The DataStep class is designed to be used with the Python with statement as shown in the following example.

Example

```
*python_DataStep.sps.
BEGIN PROGRAM.
from __future__ import with_statement
import spss
with spss.DataStep():
   datasetObj = spss.Dataset(name=None)
   datasetObj.varlist.append('numvar')
   datasetObj.varlist.append('strvar',1)
   datasetObj.varlist['numvar'].label = 'Sample numeric variable'
   datasetObj.varlist['strvar'].label = 'Sample string variable'
   datasetObj.cases.append([1,'a'])
   datasetObj.cases.append([2,'b'])
END PROGRAM.
```

- The code from __future__ import with_statement makes the Python with statement available to the program block.
- with spss.DataStep(): initiates a block of code associated with a data step. The data step is implicitly started after executing any pending transformations. All code associated with the data step should reside in the block as shown here. When the block completes, the data step is implicitly ended.

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 IBM® SPSS® StatisticsOMS command.

Example

```
handle = 'demo'
spss.DeleteXPathHandle(handle)
```

spss.EndDataStep Function

spss.EndDataStep(). *Signals the end of a data step.*

- EndDataStep must be called to end a data step initiated with StartDataStep.

For an example that uses EndDataStep, see the topic on the Dataset class.

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 IBM® SPSS® Statistics`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 Statistics 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.GetDataFileAttributeNames Function

spss.GetDataFileAttributeNames(). *Returns the names of any datafile attributes, as a tuple, for the active dataset.*

Example

```
import spss
fileattrs = spss.GetDataFileAttributeNames()
```

spss.GetDataFileAttributes Function

spss.GetDataFileAttributes(attrName). *Returns the attribute values, as a tuple, for the specified datafile attribute.* The argument *attrName* is a string that specifies the name of the attribute—for instance, a name returned by `GetDataFileAttributeNames`.

Example

```
# Build a Python dictionary of the datafile attributes
import spss
attrDict = {}
for name in spss.GetDataFileAttributeNames():
   attrDict[name] = spss.GetDataFileAttributes(name)
```

spss.GetDatasets Function

spss.GetDatasets(). *Returns a list of the available Dataset objects.* Each object in the list is an instance of the `Dataset` class. The `GetDatasets` function is intended for use within a data step or a `StartProcedure-EndProcedure` block and will return an empty list if used elsewhere. Data steps are initiated with the `spss.StartDataStep` function and are used to create and manage multiple datasets.

Example

```
import spss
spss.StartDataStep()
```

```
# Create a Dataset object for the active dataset
datasetObj1 = spss.Dataset()
# Create a new and empty dataset
datasetObj2 = spss.Dataset(name=None)
datasetNames = [item.name for item in spss.GetDatasets()]
spss.EndDataStep()
```

spss.GetDefaultPlugInVersion Function

spss.GetDefaultPlugInVersion(). *Returns the default version of the IBM® SPSS® Statistics - Integration Plug-in for Python used for Python programs.* The result is a string specifying a version—for example, `"spss170"` for version 17.0—and is useful when working with multiple versions of the plug-in on a given machine (see Note below). You can change the default using the spss.SetDefaultPlugInVersion function.

Example

```
import spss
version = spss.GetDefaultPlugInVersion()
```

Note: The functions for managing multiple versions of the plug-in (`spss.GetDefaultPlugInVersion`, `spss.SetDefaultPlugInVersion`, and `spss.ShowInstalledPlugInVersions`) operate within a given Python version, not across Python versions. For example, if you are driving IBM® SPSS® Statistics from a Python IDE installed for Python 2.7 then you can view and control the versions of the plug-in installed for Python 2.7.

For more information, see the topic Working with Multiple Versions of IBM SPSS Statistics in Chapter 1 on p. 8.

spss.GetFileHandles Function

spss.GetFileHandles(). *Returns a list of currently defined file handles*. Each item in the list consists of the following three elements: the name of the file handle; the path associated with the file handle; and the encoding, if any, specified for the file handle. File handles are created with the `FILE HANDLE` command.

spss.GetHandleList Function

spss.GetHandleList(). *Returns a list of currently defined dictionary and output XPath DOMs available for use with spss.EvaluateXpath*.

spss.GetImage Function

spss.GetImage(handle,imagename). *Retrieves an image associated with an output XPath DOM*. The argument *handle* specifies the particular XPath DOM and must be a valid handle name defined by a previous IBM® SPSS® Statistics `OMS` command. The argument *imagename* is the filename

associated with the image in the OXML output—specifically, the value of the `imageFile` attribute of the `chart`, `modelView` or `treeView` element associated with the image.

The returned value is a tuple with 3 elements. The first element is the binary image. The second element is the amount of memory required for the image. The third element is a string specifying the image type: "PNG", "JPG", "EMF", "BMP", or "VML".

Example

```
OMS
  /SELECT CHARTS
  /IF COMMANDS=['Frequencies']
  /DESTINATION FORMAT=OXML IMAGES=YES
   CHARTFORMAT=IMAGE IMAGEROOT='myimages' IMAGEFORMAT=JPG XMLWORKSPACE='demo'.

FREQUENCIES VARIABLES=jobcat
  /BARCHART PERCENT
  /ORDER=ANALYSIS.

OMSEND.

BEGIN PROGRAM.
import spss
imagename=spss.EvaluateXPath('demo','/outputTree',
 '//command[@command="Frequencies"]/chartTitle[@text="Bar Chart"]/chart/@imageFile')[0]
image = spss.GetImage('demo',imagename)
f = file('/temp/myimage.jpg','wb')
f.truncate(image[1])
f.write(image[0])
f.close()
spss.DeleteXPathHandle('demo')
END PROGRAM.
```

- The `OMS` command routes output from the `FREQUENCIES` command to an output XPath DOM with the handle name of *demo*.
- To route images along with the OXML output, the `IMAGES` keyword on the `DESTINATION` subcommand (of the `OMS` command) must be set to `YES`, and the `CHARTFORMAT`, `MODELFORMAT`, or `TREEFORMAT` keyword must be set to `IMAGE`.
- The `spss.EvaluateXPath` function is used to retrieve the name of the image associated with the bar chart output from the `FREQUENCIES` command. In the present example, the value returned by `spss.EvaluateXPath` is a list with a single element, which is then stored to the variable *imagename*.
- The `spss.GetImage` function retrieves the image, which is then written to an external file.

spss.GetLastErrorLevel and spss.GetLastErrorMessage Functions

spss.GetLastErrorLevel(). *Returns a number corresponding to an error in the preceding Python Integration Package for IBM® SPSS® Statistics function.*

- For the `spss.Submit` function, it returns the maximum SPSS Statistics error level for the submitted command syntax. SPSS Statistics error levels range from 1 to 5. An 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 99 are from the SPSS Statistics XD API.
- Error codes from 1000 to 1064 are from the Python Integration Package.

SPSS Statistics error levels (return codes), their meanings, and any associated behaviors are shown in the following table.

Table 2-2
IBM SPSS Statistics 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 Statistics processor terminates

spss.GetLastErrorMessage(). *Returns a text message corresponding to an error in the preceding Python Integration Package for SPSS Statistics function.*

- For the `spss.Submit` function, it returns text associated with the highest level error for the submitted command syntax.
- For other functions in the Python Integration Package, it returns the error message text from the SPSS Statistics 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.GetMultiResponseSetNames Function

spss.GetMultiResponseSetNames(). *Returns the names of any multiple response sets for the active dataset.*

Example

```
import spss
names = spss.GetMultiResponseSetNames()
```

spss.GetMultiResponseSet Function

spss.GetMultiResponseSet(mrsetName). *Returns the details of the specified multiple response set.* The argument *mrsetName* is a string that specifies the name of the multiple response set—for instance, a name returned by `GetMultiResponseSetNames`.

- The result is a tuple of 5 elements. The first element is the label, if any, for the set. The second element specifies the variable coding—'Categories' or 'Dichotomies'. The third element specifies the counted value and only applies to multiple dichotomy sets. The fourth element specifies the data type—'Numeric' or 'String'. The fifth element is a list of the elementary variables that define the set.

Example

```
# Build a Python dictionary of the multiple response sets
import spss
dict = {}
for name in spss.GetMultiResponseSetNames():
  dict[name]=spss.GetMultiResponseSet(name)
```

spss.GetOMSTagList Function

spss.GetOMSTagList(). *Returns a list of tags associated with any active OMS requests*. Each `OMS` request has a tag which identifies the request. The tag is specified with the `TAG` subcommand of the `OMS` command, or automatically generated if not specified.

spss.GetSetting Function

spss.GetSetting(setting,option). *Returns the value of an options setting*. Specifically, this function returns values for options that can be set with the `SET` command.

- The argument *setting* is a string specifying the name of the subcommand (of the `SET` command), whose value is desired—for example `"OLANG"`. The case of the specified string is ignored.

  Note: `GetSetting` does not support retrieving the value of the `MTINDEX` subcommand of the `SET` command.

- The argument *option* is a string specifying an option associated with the value of the *setting* argument. It only applies to the `MIOUTPUT` subcommand of `SET`, for which there is a separate setting for each of the keywords `"OBSERVED"`, `"IMPUTED"`, `"POOLED"`, and

"DIAGNOSTICS". When *setting* equals "MIOUTPUT", *option* can be set to any of those four keywords to obtain the associated value of the keyword—'Yes' or 'No'. The case of the string specified for *option* is ignored.

spss.GetSplitVariableNames Function

spss.GetSplitVariableNames(). *Returns the names of the split variables, if any, in the active dataset.*

Example

```
import spss
splitvars = spss.GetSplitVariableNames()
```

spss.GetSPSSLocale Function

spss.GetSPSSLocale(). *Returns the current IBM® SPSS® Statistics locale.*

Example

```
import spss
locale = spss.GetSPSSLocale()
```

spss.GetSPSSLowHigh Function

spss.GetSPSSLowHigh(). *Returns the values IBM® SPSS® Statistics 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. The supported format types are listed in Variable Format Types (the type code shown in the table does not apply to the `GetVariableFormat` function).

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 variable 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

spss.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.GetVariableRole Function

spss.GetVariableRole(index). *Returns a character string containing the role 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 is one of the following strings: "Input", "Target", "Both", "None", "Partition" or "Split".

Example

```
#Find the variable(s) with the role of "Target"
targets=[]
for i in range(spss.GetVariableCount()):
   if spss.GetVariableRole(i)=="Target":
      targets.append(spss.GetVariableName(i))
if len(targets):
   print "Target variables:"
   for i in range(len(targets)):
      print targets[i]
else:
   print "No target variables found"
```

spss.GetVariableType Function

spss.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 2-3
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,'/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. 46.

spss.IsActive Function

spss.IsActive(datasetObj). *Indicates whether the specified dataset is the active one.* The result is Boolean—*True* if the specified dataset is active, *False* otherwise. The argument must be an instance of the `Dataset` class. The `IsActive` function is intended for use within a data step. Data steps are initiated with the `spss.StartDataStep` function and are used to create and manage multiple datasets.

Example

```
import spss
spss.StartDataStep()
datasetObj = spss.Dataset(name="file1")
if not spss.IsActive(datasetObj):
   spss.SetActive(datasetObj)
spss.EndDataStep()
```

spss.IsOutputOn Function

spss.IsOutputOn(). *Returns the status of IBM® SPSS® Statistics 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 the topic spss.SetOutput Function on p. 106.

Example

```
import spss
spss.SetOutput("on")
if spss.IsOutputOn():
    print "The current IBM SPSS Statistics output setting is 'on'."
else:
    print "The current IBM SPSS Statistics output setting is 'off'."
```

spss.Procedure Class

spss.Procedure(procName,omsIdentifier). The `Procedure` class implicitly starts and ends a user procedure without the need to explicitly call `StartProcedure` and `EndProcedure`.

- The argument *procName* is a string and is the name that appears in the outline pane of the Viewer associated with the output from the procedure. It has the same specifications as the *procedureName* argument to the `StartProcedure` function.
- The optional argument *omsIdentifier* specifies the OMS identifier for output from this procedure and has the same specifications as the *omsIdentifier* argument to the `StartProcedure` function. *omsIdentifier* is only necessary when creating procedures with localized output so that the procedure name can be localized but not the OMS identifier. For more information, see the topic Localizing Output from Python Programs in Chapter 1 on p. 10.

The `Procedure` class is designed to be used with the Python `with` statement as shown in the following example.

Example

```
*python_Procedure.sps.
BEGIN PROGRAM.
from __future__ import with_statement
import spss
with spss.Procedure("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])
END PROGRAM.
```

- The code `from __future__ import with_statement` makes the Python `with` statement available to the program block.
- `with spss.Procedure("demoProc"):` initiates a block of code associated with a procedure named *demoProc* and implicitly starts the procedure. All code associated with the procedure should reside in the block as shown here. When the block completes, the procedure is implicitly ended.

spss.PyInvokeSpss.IsUTF8mode Function

spss.PyInvokeSpss.IsUTF8mode(). *Indicates whether SPSS Statistics is running in Unicode mode or code page mode.* The result is 1 if IBM® SPSS® Statistics is in Unicode mode, 0 if SPSS Statistics is in code page mode.

Example

```
import spss
isUTF8 = spss.PyInvokeSpss.IsUTF8mode()
if isUTF8==1:
  print "IBM SPSS Statistics is running in Unicode mode."
else:
  print "IBM SPSS Statistics is running in code page mode."
```

spss.PyInvokeSpss.IsXDriven Function

spss.PyInvokeSpss.IsXDriven(). *Checks to see how the SPSS Statistics backend is being run.* The result is 1 if Python is controlling the IBM® SPSS® Statistics backend or 0 if SPSS Statistics is controlling the backend.

Example

```
import spss
spss.Submit("""
GET FILE
 '/examples/data/employee data.sav'.
""")
isxd = spss.PyInvokeSpss.IsXDriven()
if isxd==1:
  print "Python is driving IBM SPSS Statistics."
```

```
else:
  print "IBM SPSS Statistics is driving Python."
```

spss.SetActive Function

spss.SetActive(datasetObj). *Sets the specified dataset as the active one.* The argument must be an instance of the `Dataset` class. The `SetActive` function can only be used within a data step. Data steps are initiated with the `spss.StartDataStep` function and are used to create and manage multiple datasets.

Example

```
# Set a newly created dataset to be active
spss.StartDataStep()
ds1 = spss.Dataset(name=None)
spss.SetActive(ds1)
spss.EndDataStep()
```

spss.SetDefaultPlugInVersion Function

spss.SetDefaultPlugInVersion(value). *Sets the default version of the IBM® SPSS® Statistics - Integration Plug-in for Python used for Python programs.* This function is useful when working with multiple versions of the plug-in on a given machine (see Note below). The value of the argument is a quoted string or an integer specifying a plug-in version—for example, `"spss160"` or `160` for version 16.0. The strings representing the installed versions of the plug-in are available from the function spss.ShowInstalledPlugInVersions.

- For version 17.0 and higher, `SetDefaultPlugInVersion` also sets the default version of the Integration Plug-in for Python used for Python scripts (Python code that utilizes the `SpssClient` module).

Example

```
import spss
spss.SetDefaultPlugInVersion("spss160")
```

Note: The functions for managing multiple versions of the plug-in (`spss.GetDefaultPlugInVersion`, `spss.SetDefaultPlugInVersion`, and `spss.ShowInstalledPlugInVersions`) operate within a given Python version, not across Python versions. For example, if you are driving IBM® SPSS® Statistics from a Python IDE installed for Python 2.7 then you can view and control the versions of the plug-in installed for Python 2.7.

For more information, see the topic Working with Multiple Versions of IBM SPSS Statistics in Chapter 1 on p. 8.

spss.SetMacroValue Function

spss.SetMacroValue(name, value). *Defines a macro variable that can be used outside a program block in command syntax.* The first argument is the macro name, and the second argument is the macro value. Both arguments must resolve to strings.

- The argument specifying the macro value cannot contain the characters \ or ^ unless they are contained within a quoted string.

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 Statistics output in Python when running SPSS Statistics from Python.* Output is displayed as standard output, and charts and classification trees are not included. When running Python from IBM® SPSS® Statistics within program blocks (`BEGIN PROGRAM-END PROGRAM`), this function has no effect. The value of the argument is a quoted string:

- **"on".** Display SPSS Statistics output in Python.
- **"off".** Do not display SPSS Statistics output in Python.

Example

```
import spss
spss.SetOutput("on")
```

spss.SetOutputLanguage Function

spss.SetOutputLanguage("language"). *Sets the language that is used in IBM® SPSS® Statistics output.* The argument is a quoted string specifying one of the following languages: "English", "French", "German", "Italian", "Japanese", "Korean", "Polish", "Russian", "SChinese" (Simplified Chinese), "Spanish", "TChinese" (Traditional Chinese), or "BPortugu" (Brazilian Portuguese). The setting does not apply to simple text output.

Example

```
import spss
spss.SetOutputLanguage("German")
```

spss.ShowInstalledPlugInVersions Function

spss.ShowInstalledPlugInVersions(). *Returns the installed versions of the IBM® SPSS® Statistics - Integration Plug-in for Python.* This function returns a list of string identifiers for the installed versions of the plug-in—for example, `["spss160","spss170"]` for versions 16.0 and 17.0—and is useful when working with multiple versions of the plug-in on a given machine (see Note below). Use an identifier from this list as the argument to the spss.SetDefaultPlugInVersion function.

Example

```
import spss
versionList = spss.ShowInstalledPlugInVersions()
```

Note: The functions for managing multiple versions of the plug-in (`spss.GetDefaultPlugInVersion`, `spss.SetDefaultPlugInVersion`, and `spss.ShowInstalledPlugInVersions`) operate within a given Python version, not across Python versions. For example, if you are driving IBM® SPSS® Statistics from a Python IDE installed for Python 2.7 then you can view and control the versions of the plug-in installed for Python 2.7.

For more information, see the topic Working with Multiple Versions of IBM SPSS Statistics in Chapter 1 on p. 8.

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 the topic spss.StartProcedure Function on p. 109.

- 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="/examples/data/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.StartDataStep Function

spss.StartDataStep(). *Signals the beginning of a data step.* A data step allows you to create and manage multiple datasets.

- You cannot use the following classes and functions within a data step: the `Cursor` class, the `BasePivotTable` class, the `BaseProcedure` class, the `TextBlock` class, the `StartProcedure` function, the `Submit` function, and the `StartDataStep` function (data steps cannot be nested).
- The `StartDataStep` function cannot be used if there are pending transformations. If you need to access case data in the presence of pending transformations, use the `Cursor` class.
- To end a data step, use the `EndDataStep` function.

For an example of using `StartDataStep`, see the topic on the Dataset class.

To avoid the need to check for pending transformations before starting a data step, use the DataStep class. It implicitly starts and ends a data step and executes any pending transformations.

spss.StartProcedure Function

spss.StartProcedure(procedureName,omsIdentifier). *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, create new datasets, and produce pivot table output and text blocks in the IBM® SPSS® Statistics Viewer. Procedures have almost the same capabilities as built-in SPSS Statistics 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 in the active dataset using the Cursor class, or create new datasets with the Dataset class. Pivot tables are created using the BasePivotTable class. Text blocks are created using the TextBlock class.

- The argument *procedureName* is a string and is the name that appears in the outline pane of the Viewer associated with the output. If the optional argument *omsIdentifier* is omitted, then *procedureName* 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.
- The optional argument *omsIdentifier* is a string and is 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. If *omsIdentifier* is omitted, then the value of the *procedureName* argument is used as the OMS identifier. *omsIdentifier* is only necessary when

creating procedures with localized output so that the procedure name can be localized but not the OMS identifier. For more information, see the topic Localizing Output from Python Programs in Chapter 1 on p. 10.

- In order that names associated with output not conflict with names of existing SPSS Statistics commands (when working with OMS), it is recommended that they have the form *yourcompanyname.com.procedurename*.
- 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.
- Instances of the `Dataset` class created within `StartProcedure-EndProcedure` blocks cannot be set as the active dataset.
- 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.

  Note: You can use the `spss.Procedure` class to implicitly start and end a procedure without the need to call `StartProcedure` and `EndProcedure`. For more information, see the topic spss.Procedure Class on p. 103.

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 */Python27/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='/examples/data/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 */examples/data/Employee data.sav*.

Result

Figure 2-16
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.StartSPSS Function

spss.StartSPSS(). *Starts a session of IBM® SPSS® Statistics.*

- This function starts a session of SPSS Statistics, for use when driving SPSS Statistics from Python. The function has no effect if a session is already running. *Note*: The `spss.Submit` function automatically starts a session of SPSS Statistics.
- This function has no effect when running Python from SPSS Statistics (within program blocks defined by `BEGIN PROGRAM-END PROGRAM`).

spss.StopSPSS Function

spss.StopSPSS(). *Stops IBM® SPSS® Statistics, ending the session.*

- This function is ignored when running Python from SPSS Statistics (within program blocks defined by `BEGIN PROGRAM-END PROGRAM`).
- When running SPSS Statistics from Python, this function ends the SPSS Statistics session, and any subsequent `spss.Submit` functions that restart SPSS Statistics 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 IBM SPSS Statistics from Python

```
#RunSpssFromPython.py
import spss
#start a session and run some commands
#including one that defines an active dataset
spss.Submit("""
GET FILE '/examples/data/employee data.sav'.
FREQUENCIES VARIABLES=gender jobcat.
""")
#shutdown the session
spss.StopSPSS()
#insert a bunch of Python statements
#starting a new session and running some commands without defining
#an active dataset results in an error
spss.Submit("""
FREQUENCIES VARIABLES=gender jobcat.
""")
```

Example: Running Python from IBM SPSS Statistics

```
*run_python_from_spss.sps.
BEGIN PROGRAM.
import spss
#start a session and run some commands
#including one that defines an active dataset
spss.Submit("""
GET FILE '/examples/data/employee data.sav'.
FREQUENCIES VARIABLES=gender jobcat.
""")
#following function is ignored
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 IBM® SPSS® Statistics for processing.* The argument can be a quoted string, a list, or a tuple.

- The argument should resolve to one or more complete SPSS Statistics 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 Statistics commands on multiple lines that more closely resemble the way you might normally write command syntax.

- If SPSS Statistics is not currently running (when driving SPSS Statistics from Python), `spss.Submit` will start the SPSS Statistics backend processor.
- Submitted syntax for `MATRIX-END MATRIX` and `BEGIN DATA-END DATA` blocks cannot be split across `BEGIN PROGRAM-END PROGRAM` blocks.
- The following commands are not supported by `Submit` when driving SPSS Statistics from Python: `OUTPUT EXPORT`, `OUTPUT OPEN` and `OUTPUT SAVE`.

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="/examples/data/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

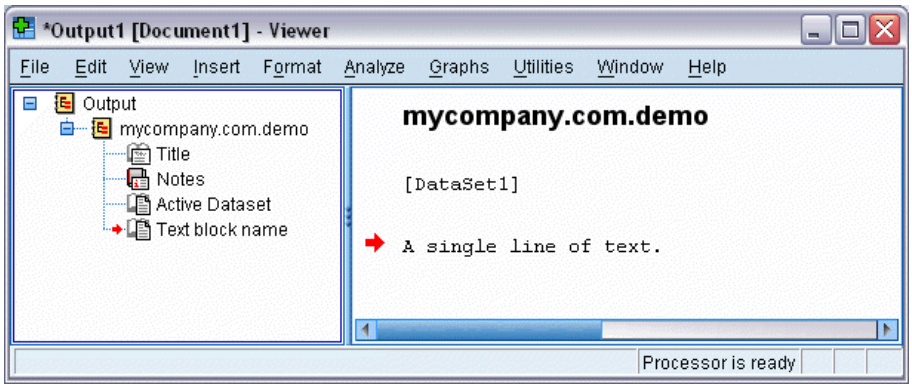
spss.TextBlock(name,content,outline). *Creates and populates a text block item in the Viewer.* The argument *name* is a string that specifies the name of this item in the outline pane of the Viewer. The argument *content* is a string that specifies the text. The string may include the escape sequence `\n` to specify line breaks, but must otherwise be specified as plain text (HTML and rich text formatting are not supported). You can also add lines using 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 2-17
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 the text. The string may include the escape sequence `\n` to specify line breaks, but must otherwise be specified as plain text (HTML and rich text formatting are not supported). 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()
```

Appendix

A

Variable Format Types

Type	Description
1	**A.** Standard characters.
2	**AHEX.** Hexadecimal characters.
3	**COMMA.** Numbers with commas as the grouping symbol and a period as the decimal indicator. For example: 1,234,567.89.
4	**DOLLAR.** Numbers with a leading dollar sign ($), commas as the grouping symbol, and a period as the 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	**WKDAY.** Day of the week.
27	**MONTH.** Month.
28	**MOYR.** Month and year.
29	**QYR.** Quarter and year of the general form qQyyyy.
30	**WKYR.** Week and year.
31	**PCT.** Percentage sign after numbers.
32	**DOT.** Numbers with periods as the grouping symbol and a comma as the decimal indicator. For example: 1.234.567,89.
33	**CCA.** Custom currency format 1.
34	**CCB.** Custom currency format 2.
35	**CCC.** Custom currency format 3.
36	**CCD.** Custom currency format 4.
37	**CCE.** Custom currency format 5.
38	**EDATE.** European date of the general form dd.mm.yyyy.
39	**SDATE.** Sortable date of the general form yyyy/mm/dd.

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