ArcGIS® 9

Writing Geoprocessing Scripts With ArcGIS

Published by
ESRI
380 New York Street
Redlands, California 92373-8100

Author
Corey Tucker

Contents

Introduction to scripting in ArcGIS

This chapter reviews what a script is and why it is relevant to today's Geographic Information System (GIS) user. Issues such as what scripting language should be used and how ArcGIS® works with scripting languages are also covered.

Geoprocessing tasks can be time intensive since they are often performed on a number of different datasets or on large datasets with numerous records. Scripting is an efficient method of automating geoprocessing tasks. Scripting allows the execution of simple processes (a single tool) or complex processes (piggybacked, multitool tasks with validation). In addition, scripts are recyclable, meaning they can be data nonspecific and used again.

Any ArcGIS user has the option of writing a script to automate geoprocessing work flows. Even if you have never thought of yourself as a programmer, after reading the first three chapters in this book, you will be able to write scripts to perform geoprocessing. You can also create models in ModelBuilder™, which provides a canvas for you to visually create a geoprocessing work flow. This book will explain how these scripts work and how you can modify them to incorporate loops for batch processing or if statements for conditional control.

Microsoft's Component Object Model (COM) is a mature technology for sharing software components that is language neutral. COM is a protocol for how one binary component connects with another; it is not a language.

ArcGIS 9 introduces scripting support for many of today's most popular scripting environments, such as Python, VBScript, JScript, and Perl. A new ArcObjects™ component, the geoprocessor, manages all the geoprocessing functions available within ArcGIS. It is an object that provides a single access point and environment for the execution of any geoprocessing tool in ArcGIS, including extensions. The geoprocessor implements automation using the COM IDispatch interface, making it possible for interpretative and macro languages to access the more than 400 available tools.

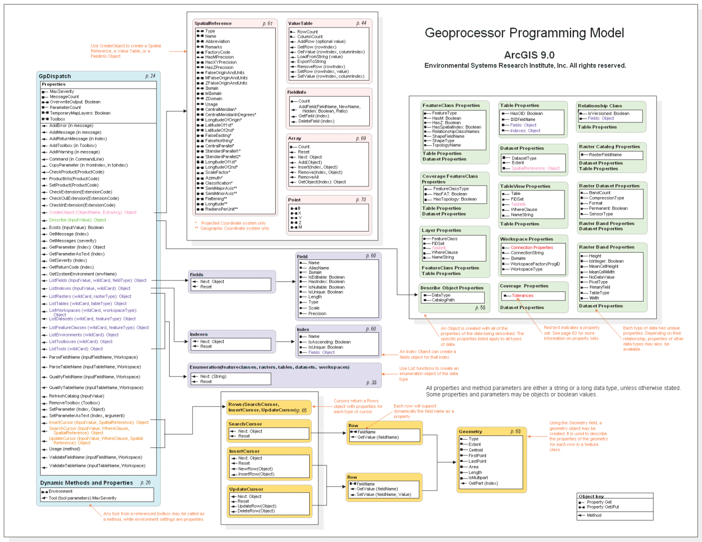
Which scripting language to choose can be an open question. Any scripting language that is COM compliant, interacts well in a Web-based environment, and allows users to complete their tasks is a viable option. Although there are a number of good scripting languages on the market, for simplicity this book will mention three of the more popular languages that meet the necessary criteria: VBScript, JScript, and Python. VBScript and JScript are familiar to many people and are relatively simple languages. Similar to Visual Basic and C they are designed to operate in a Windows environment. Python is an easy-to-learn language similar to C. Python has the ease of use of a scripting language, along with the programming capabilities of a complete developer language. Moreover, Python is platform independent and can operate on a variety of operating systems including UNIX, Linux, and Windows. For more information, visit www.python.org on the Web.

The samples in this book use Python as the scripting language because ESRI sees it as the language that fulfills the needs of our user community. Some advantages of Python are:

- Python is easy to learn because of its clean syntax and simple, clear concepts.
- Python supports object-oriented programming in an easy-to-understand manner.
- Documenting Python is easier because it has readable code.
- Complicated data structures are easy to work with in Python.
- Python is simple to integrate with C++ and Fortran.
- Python can be seamlessly integrated with Java.
- Python is free from the Web and has a widespread community.

Using the Geoprocessor Programming Model diagram

An accompanying diagram, the Geoprocessor Programming Model, gives an overview of the geoprocessor object and all the other objects that may be created from its methods. You will notice that some of the object diagrams found on the diagram are placed in various pages of this book and that there are page numbers on the objects in the diagram. When an object is discussed, its diagram will be displayed in the left margin so you can see all its properties and methods. Use the diagram as a quick reference for names, parameters and relationships when writing your scripts. You may notice that the type of data returned for a property is only specified when it is not a string or number. Most values are simply strings or numbers, so to keep the diagram simple, it only informs you of a data type when it is not a string or number.

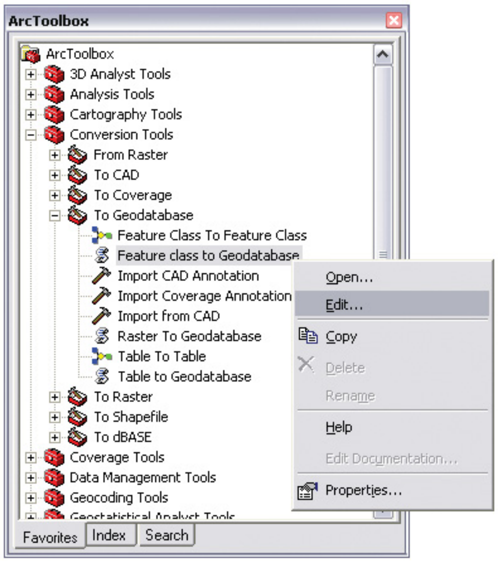

Geoprocessor Programming Model
ArcGIS 9.0
Environmental Systems Research Institute, Inc. All rights reserved.
GpDispatch
SpatialReference
ValueTable
FieldInfo
Array
Point
Fields
Field
Indexes
Index
Enumeration(featureclasses, rasters, tables, datasets, workspaces)
Rows (SearchCursor, InsertCursor, UpdateCursor)
SearchCursor
InsertCursor
UpdateCursor
Row
Geometry
Dynamic Methods and Properties
Object key

The Geoprocessor Programming Model is not based on the Unified Modeling Language (UML) notation used in the ArcObjects object model diagrams. The arrows on the diagram indicate instantiation. GpDispatch is the COM name for the geoprocessor IDispatch object. Any scripting language that can instantiate a COM IDispatch object may create the geoprocessor. In Python, the Dispatch method on the `win32com` module's client object is used to create the geoprocessor, while Visual Basic and VBScript use the CreateObject function. The geoprocessor is the only ArcGIS scripting object that can be directly created using these methods. All other ArcGIS scripting objects are created by a method on the geoprocessor, such as CreateObject, or as a property of an object, such as the fields of a feature class.

Use the object key on the diagram to understand the meaning of the symbols adjacent to property and method names.

The different colors are meant to show logical relationships of objects. For example, the purple objects are created by the various list methods of the geoprocessor. The list methods are purple as well to indicate their relationship with these objects. Some properties, such as Field in a feature class, are colored to indicate they are an object, which, in this case, is the fields object that is purple. The colors should help you connect the various methods and properties with the appropriate objects and make the diagram easy to read.

The GpDispatch object supports the execution of tools as dynamic methods and the use of environment settings as dynamic properties. The diagram does not show these dynamic members of the geoprocessor, as the tools and environments that are accessible depend on what is installed on the desktop machine being used and what toolboxes are being referenced. The geoprocessor object simply refers to tools and environments as generic methods and properties. See Chapter 3 for further details and examples of how to use tools and set environments. Refer to Chapter 6 of *Geoprocessing in ArcGIS* for more information about environment settings.



Different types of tools are indicated using icons. Some tools, such as scripts and models may allow their source to be viewed and edited

Scripting examples

This book contains many examples using Python to coordinate and execute geoprocessing tools. Many of these samples focus on the use of specific methods, properties, or both methods and properties of the geoprocessor and are intended to be consise and easy to follow. More detailed and sometimes complicated examples of how to use the geoprocessor to solve problems may be found as tools in ArcGIS. While most tools delivered with ArcGIS are written in C++ and delivered in a binary form, some are written in Python or ModelBuilder. Some tools execute batch operations, such as loading data into a geodatabase or projecting a set of feature classes, which is perfectly suited for scripting as the solution. The underlying script may be opened at any time to see what happens when the tool is run. Use this capability to learn how to use the geoprocessor in a number of situations, such as creating multiple ring buffers or building pyramids for a set of rasters. Script tools have a unique icon , so you can easily find tools that use scripts to execute. Use the edit option in a tool's context menu to open the underlying script. These scripts may not be edited as they are read-only, but you can copy the tool to a custom toolbox and its underlying script to another folder so you can make edits if desired. Refer to Chapter 5 and the 'Creating models and adding scripts' section of Chapter 5 of *Geoprocessing in ArcGIS* to learn more about script tools.

ModelBuilder is the interface used to create models in ArcGIS. A model is a collection of processes that form a work flow. Refer to Chapter 8 of Geoprocessing in ArcGIS *to learn more about models and ModelBuilder.*

ModelBuilder may also be used to generate scripts, as it will export a model to either Python, VBScript, or JScript. The Help for each geoprocessing tool also contains a section specific to scripting, with a description of each parameter and an example of how to use the tool in a Python script.

Getting help on your computer

In addition to this book, use the online Help system to learn how to use ArcGIS. To learn how to use the online Help system, see *Using ArcMap*. For information on a particular geoprocessing function, see the online geoprocessing command reference or click the Help button on a geoprocessing dialog box.

Contacting ESRI

If you need to contact ESRI for technical support, see the product registration and support card you received with ArcGIS Spatial Analyst or refer to 'Contacting Technical Support' in the 'Getting more help' section of the ArcGIS Desktop Help system. You can also visit ESRI on the Web at *www.esri.com* and *www.arconline.esri.com* for more information on Spatial Analyst and ArcGIS.

ESRI education solutions

ESRI provides educational opportunities related to geographic information science, GIS applications, and technology. You can choose from among instructor-led courses, Web-based courses, and self-study workbooks to find education solutions that fit your learning style and pocketbook. For more information, visit *www.esri.com/education*.

2

Getting started

There are endless uses for scripts and many environments in which to write them. Python for Windows provides an easy-to-use yet powerful integrated development environment (IDE) for writing and debugging scripts. This chapter will demonstrate how to use PythonWin to create a script for batch processing data.

Chapter 1 outlined many of the advantages of using Python for scripting with ArcGIS. This chapter will demonstrate some of these by going through the steps of creating a script. One of the most common reasons for writing a script is to automate a work flow that uses many different datasets as input. Data regularly needs to be manipulated before analysis can be done. Data, such as a set of shapefiles, may need to be projected to the appropriate coordinate system and clipped by a polygon feature class, defining a boundary. Other times, data needs to be added to a geodatabase so it can have its topology validated before more features are added. Scripting is one of the best ways to quickly define and execute a work flow.

Script Description

The Clip geoprocessing tool extracts features from one feature class that overlay the polygon features of another feature class.

The script that will be created is used in Exercise 3 of the *Geoprocessing in ArcGIS* tutorial. The script is called multi_clip.py, and it clips all the feature classes found in a workspace with a polygon feature class, creating new feature classes in an output workspace. In the tutorial, the script is used to clip all the shapefiles in a folder with a polygon shapefile defining a study area. The script may also move the resulting data into a geodatabase if the output workspace is an Access .mdb file or ArcSDE® connection. The script is provided with the tutorial data, but the tutorial does not explain how the script works. That will be done in this chapter, as we will write the script and discuss what each line does.

You will learn how to write a script in PythonWin, which provides the following:

- A script editor
- Integrated debugger
- Interactive command window
- Python COM and Microsoft Foundation Class (MFC) support
- Documentation for working with Python on Windows

Requirements

Python is an Open Source scripting language administered by the Python Software Foundation. Visit www.python.org to learn more about Python and the mission of the foundation.

The requirements for writing this script are that Python 2.1, and PythonWin be installed, which is provided by the ArcGIS Desktop install program, as well as ArcGIS Desktop. The script you create will clip all shapefiles in a folder or standalone feature classes in a geodatabase or geodatabase feature dataset with a polygon feature class that shares a common extent. The tutorial for *Geoprocessing in ArcGIS* provides suitable data for this script, so if you do not already have data that fits this description, you should also install the geoprocessing tutorial for ArcGIS.

Script Editors

You can visit the home page for PythonWin by selecting the About Python option in PythonWin's Help menu. There is a button at the bottom of the dialog box called Home Page that will display the Python for Windows Web site when clicked if you are connected to the Internet.

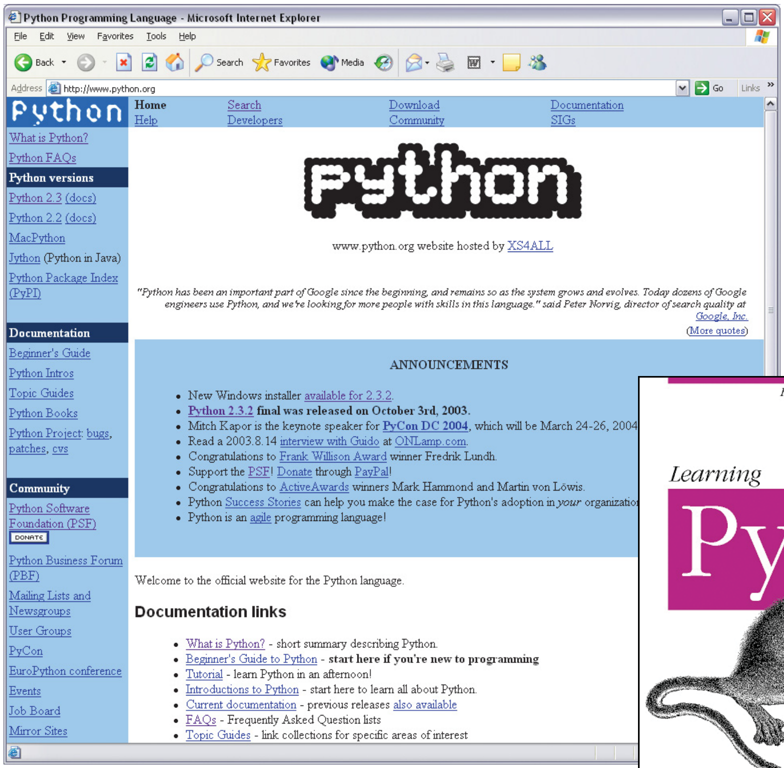
You will work with the PythonWin Editor, Debugger and Interactive Command windows during this exercise. Python scripts may be written in other applications, as PythonWin is not required to write a Python script. Scripts are simply text files; any text editor, such as Notepad or VI, may be used to author a script. The standard Python installation provides a default Python editor called Integrated DeveLopment Environment (IDLE), which also provides search capabilities and a symbolic debugger. IDLE is a good application for writing Python scripts, but the advantage of using PythonWin is the integration of its debugger

See Chapter 3 for more information on how the geoprocessor works.

with the interactive window in a standard Windows style application. PythonWin is required by ArcGIS, as it provides the communication between standard Python and the COM ArcObjects used to write the geoprocessor.

Python References

This book is not a Python language reference. Certain Python syntax and behavior is explained with respect to the examples and concepts used to demonstrate how to write a geoprocessing script. Python is a rich language supporting a number of operating systems and programming libraries. You should obtain a suitable Python reference to augment the information you find in this book. For Python beginners, *Learning Python* by Mark Lutz and David Asher and published by O'Reilly & Associates is a good introduction to the language and is not overwhelming in scope. There are many other books on Python and its particular uses, with new ones being released regularly, so explore what is available. The Python Web site, www.python.org, has full documentation for Python, but it is concise and developer oriented. There is a large online Python community with many online


Python Programming Language - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://www.python.org
Go Links

Home Search Download Documentation
Help Developers Community SIGs

What is Python?
Python FAQs
Python versions
Python 2.3 (docs)
Python 2.2 (docs)
MacPython
Jython (Python in Java)
Python Package Index (PyPI)
Documentation
Beginner's Guide
Python Intros
Topic Guides
Python Books
Python Project: bugs, patches, cvs
Community
Python Software Foundation (PSF)
DONATE
Python Business Forum (PBF)
Mailing Lists and Newsgroups
User Groups
PyCon
EuroPython conference
Events
Job Board
Mirror Sites

www.python.org website hosted by XS4ALL

"Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we're looking for more people with skills in this language." said Peter Norvig, director of search quality at Google, Inc.
(More quotes)

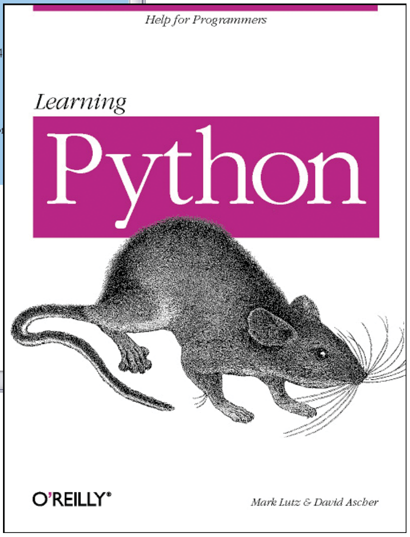
ANNOUNCEMENTS

- New Windows installer available for 2.3.2.
- **Python 2.3.2 final was released on October 3rd, 2003.**
- Mitch Kapor is the keynote speaker for **PyCon DC 2004**, which will be March 24-26, 2004
- Read a 2003.8.14 interview with Guido at ONLamp.com.
- Congratulations to Frank Willison Award winner Fredrik Lundh.
- Support the PSF! Donate through PayPal!
- Congratulations to ActiveAwards winners Mark Hammond and Martin von Löwis.
- Python Success Stories can help you make the case for Python's adoption in *your* organizatio
- Python is an agile programming language!

Welcome to the official website for the Python language.

Documentation links

- What is Python? - short summary describing Python.
- Beginner's Guide to Python - **start here if you're new to programming**
- Tutorial - learn Python in an afternoon!
- Introductions to Python - start here to learn all about Python.
- Current documentation - previous releases also available
- FAQs - Frequently Asked Question lists
- Topic Guides - link collections for specific areas of interest



resources that are accessible from the Python home page.

Double quotes are used to indicate what should be typed in the Interactive window or script. Do not include the double quotes wrapping each code snippet, as it will cause a syntax error. You must include quotes when defining the value of a string variable.

The Interactive Window and variables

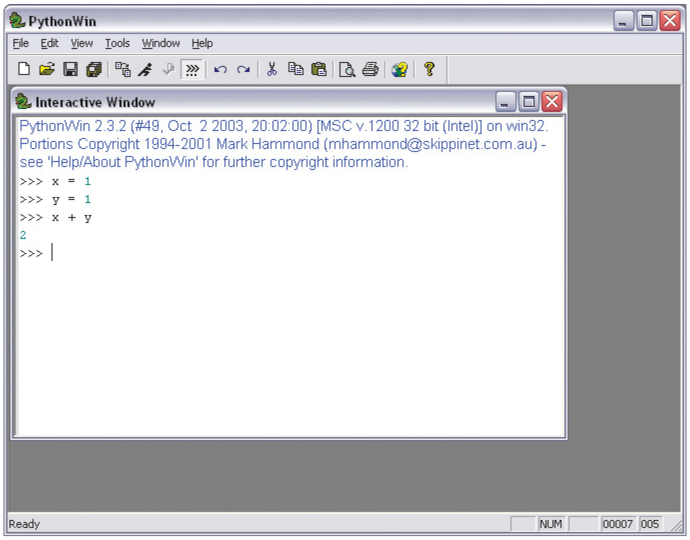
Before creating a script, you need to take a closer look at the PythonWin application itself.

1. Start PythonWin by double-clicking a shortcut installed on your desktop or using the Programs list in your Start menu.

PythonWin starts with the Interactive window open when it is opened for the first time. The Interactive window may be used to execute a single line of Python code, with the resulting messages printed to the window.

To change the contents of a string variable, create a new variable. A variable may be reset by redeclaring its value. For example, a string variable named MyStr with a value of "x" may be set to uppercase using the upper tool. "MyStr = MyStr.upper()"

2. Click the Interactive window and type "x = 1". Press Enter. This creates a variable x with a value of 1. Variables in Python do not have to be declared with a specific type before they can be used. In this case, an integer variable has been created and initialized with a value.



```
PythonWin 2.3.2 (#49, Oct  2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)] on win32.
Portions Copyright 1994-2001 Mark Hammond (mhammond@skippinet.com.au) -
see 'Help/About PythonWin' for further copyright information.
>>> x = 1
>>> y = 1
>>> x + y
2
>>> 
```

3. Type "y = 1" into the next line and press Enter. This creates another integer variable y.

4. Type "x + y" into the next line and press enter. The result, 2, is immediately printed. The immediate window provides direct access to the Python interpreter, as it evaluates each line as it is entered.

Assignment is done using a single equal sign "=", while equivalence is tested using two equal signs "==". The type of variable is defined by the value it is assigned. Python has a number of numeric types such as long, short integer, floating, and complex. Refer to your Python language reference for more information about numeric objects and expressions.

Strings variables are used frequently in scripts, especially when defining geoprocessing tool parameters. Paths or partial paths are used to define a tool's input or output. String operations, such as concatenation, are important when working with datasets in a number of workspaces. Strings are a built-in type in Python and are defined as an ordered collection of characters. Once a string's character sequence has been set, it can not be changed, but there are many ways in Python to work with those characters.

Python can declare and initialize multiple variables on the same line, which is a good way to keep your scripts compact and easy to follow. The previous example could have been declared as "x, y = 1, 1".

Python does not permit the concatenation of strings with other types of variables, such as numbers or booleans. Use the str() tool to convert a nonstring variable to a string during a concatenation statement to avoid a syntax error.

5. Type "a = "Buddy"" into the next line and press Enter. This creates a string variable called a.

6. Type "b = "Holly"" into the next line and press Enter.

7. Type "a + " " + b" into the next line and press Enter to concatenate the strings. The evaluated full name will be printed below the expression.

8. Type "print a[0]" and press Enter. This prints the first character in the variable

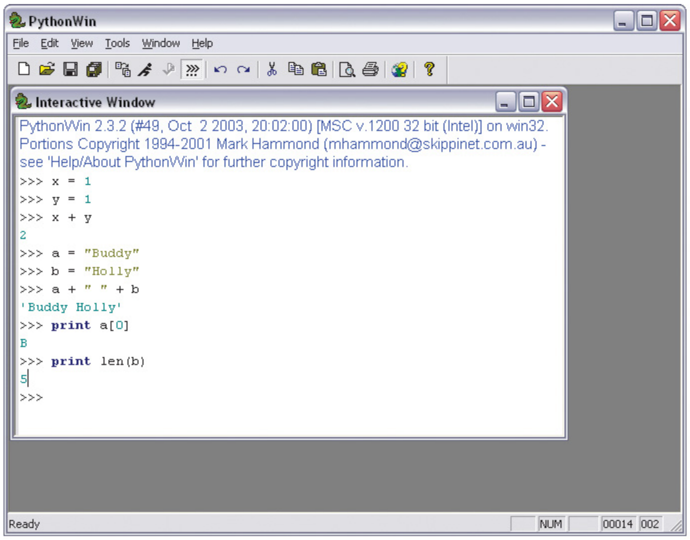
The string module provides many tools for string manipulation, such as searching, conversion, and formatting. You can access these tools by importing the string module within your script. Many of these tools are also found as methods of a string object, so the string module is not always needed.

a. Strings are indexed, with the first character being 0. A set of characters may be accessed by specifying a range. Specifying “a[0:3]” would return “Bud”.

9. Type “print len(b)” and press Enter. This returns the length, or number of characters, in variable b.

There are many tools for working with strings in Python and many techniques for extracting, repeating, and formatting them. Refer to your Python reference for more information on working with strings.

10. Click the Interactive window button on the Standard toolbar to close it.



```
PythonWin 2.3.2 (#49, Oct  2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)] on win32.
Portions Copyright 1994-2001 Mark Hammond (mhammond@skippinet.com.au) -
see 'Help/About PythonWin' for further copyright information.
>>> x = 1
>>> y = 1
>>> x + y
2
>>> a = "Buddy"
>>> b = "Holly"
>>> a + " " + b
'Buddy Holly'
>>> print a[0]
B
>>> print len(b)
5
>>>
```

Importing Modules

By definition, Python scripts are modules, which is Python’s highest level of code organization. Modules serve several purposes; the most obvious is the ability to save code to a reusable form, a script. Scripts may avoid duplication of code by using programs contained in other modules. Using the `import` statement, a program may import any number of functions, variables, or both contained in another module. A module creates a natural grouping and naming structure, which eliminates ambiguity when using commonly named programs.

Each time you create a script, you will import one or more modules. All geoprocessing scripts use the `win32com` module, as it provides the programs needed to support the communication between Python and the COM IDispatch interface, which is used by the geoprocessor. This is covered in more detail in Chapter 3. Other modules that are required to work with math, files, and the operating system may also be imported, depending on the requirements of your script.

Creating a new script module

1. In PythonWin click the File menu and click the New option. Accept the default option of Python Script and click OK.

The Script1 window will open. Script1 is the default name of your script.

2. Click the maximize window button on the Script1 window.

3. Click the File menu and click Save As. Name the script "multi_clip" and save it in a folder of your choice.

Modules should follow the same naming rules as variables. They must start with a letter and can be followed by any number of letters, digits, or underscores. Module names become variable names when they are imported, so to avoid syntax errors, they should not use reserved words such as *while* or *if*. Scripts should always have a .py extension, as this is required when importing a script module. It also defines Python as the executing application for your script.

4. At the top of the code window add the following lines:

```
#Import standard library modules
import win32com.client, sys, os
```

This imports the system, operating system, and Windows 32 modules to the script. The `win32com` module has already been discussed, but the other two are new. The `sys` module refers to the Python system and will be used to access user-specified inputs. The `os` module provides easy access to the most fundamental tools of the operating system. Some of the `os` module's filename manipulation tools are used in this script. Refer to your Python reference material for more information about the tools in these modules.

See Chapter 3 for a complete description of the geoprocessor object and how to work with its properties and methods.

5. Add the following code to declare the geoprocessor object:

```
#Create the Geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GpDispatch.1")
```

This script will have the following arguments so it can be used in a generic fashion:

- An input workspace defining the set of feature classes to process
- A feature class to be used by the Clip tool as the area to be clipped from an input feature class
- An output workspace where the results of the Clip tool will be written
- A cluster tolerance that will be used by the Clip tool

Refer to the Clip tool in the Analysis toolbox for detailed information on how Clip works.

When naming variables, be mindful that Python is case sensitive, so "GP" is not the same as "gp". The names of the geoprocessor's methods and properties (including tool and environment setting names) are NOT case sensitive.

Python automatically removes an object from memory when it is no longer referenced in a program. When a script completes, all memory allocated for objects is released and any open files are closed. The `del` *statement can be used to delete an object so its memory is deallocated during a script's execution.*

6. Add the following code to your script to define and set variables based on the user-defined values passed to the script at execution:

```
#Set the input workspace
GP.workspace = sys.argv[1]

#Set the clip featureclass
clipFeatures = sys.argv[2]
```

```
#Set the output workspace
outWorkspace = sys.argv[3]

#Set the cluster tolerance
clusterTolerance = sys.argv[4]
```

The passed-in arguments are attributes of the sys module and can be sliced to extract the argument you want. The first value of argv, argv[0], is the name of the script. You can apply the same slicing tools used on strings to the argv attribute, so sys.argv[1:] will return the entire list of argument values.

Many geoprocessing scripts are not generic in nature and have no arguments. They use defined dataset names and parameter values and may not require the sys module. Import modules only as needed to avoid unnecessary memory use.

7. Add the following error handling statement and geoprocessor call to the script window:

```
try:
   #Get a list of the featureclasses in the input folder
   fcs = GP.ListFeatureClasses()
```

Python enforces indentation of code after certain statements as a construct of the language. The try statement defines the beginning of a block of code that will be handled by its associated exception handler, or except statement. All code within this block must be indented. Python uses try/except blocks to handle unexpected errors during execution. Exception handlers define what the program should do when an exception is raised by the system or by the script itself. In this case you are only concerned about an error occurring with the geoprocessor, so a try block is started just before you start to use the object. It is good practice to use exception handling in any script using the geoprocessor so its error messages can be propagated back to the user. This also allows the script to exit gracefully and return informative messages instead of simply causing a system error.

Statements that end with a colon indicate the beginning of indented code. Python does not use braces, brackets, or semicolons to indicate the beginning or end of a block of code. Instead Python uses the indentation of the block to define its boundaries. This results in code that is easy to read and write.

The geoprocessor's ListFeatureClasses method returns an enumeration of feature class names in the current workspace. The workspace defines the location of your data and where all new data will be created unless a full path is specified. The workspace has already been set to the first argument's value. Chapter 4 discusses how to work with enumerations in detail.

8. Add the following code to get the first value of the enumeration:

```
   #Loop through the list of feature classes
   fcs.Reset()
   fc = fcs.Next()
```

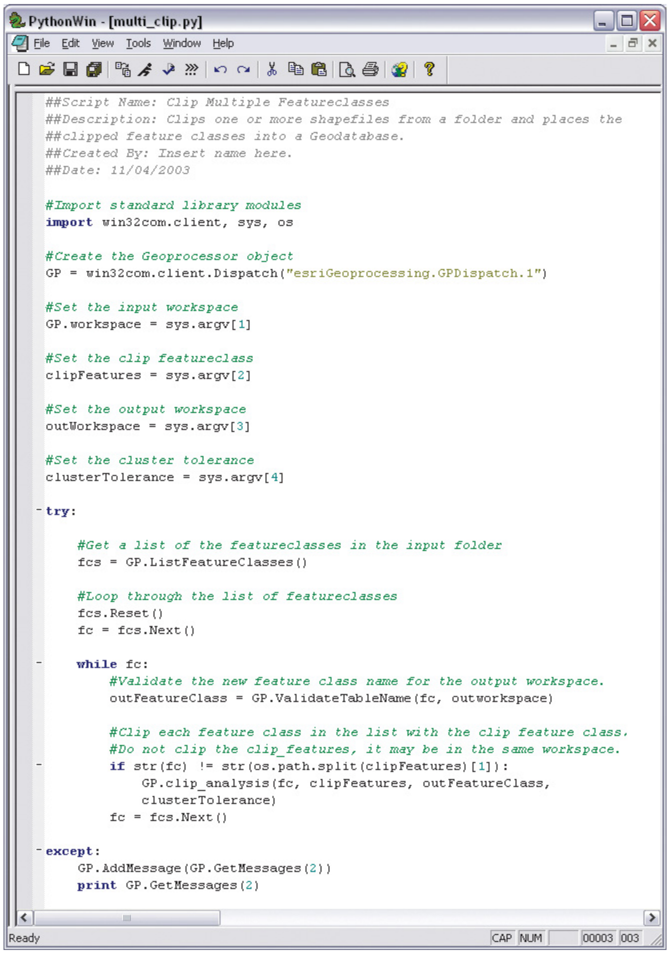
An enumeration is a list that has an undefined number of items. It should always be reset before the first item is extracted so you can be sure you get the first item in the list. You want to get the first item before you start looping through its contents as you will use it as the conditional value for continuing a loop.

9. Add the following code:

```
  while fc:
   #Validate the new feature class name for the output workspace.
   outFeatureClass = outWorkspace + "/" + GP.ValidateTableName(fc,
                     outWorkspace)
   #Clip each feature class in the list with the clip feature class.
```

```
#Do not clip the clipFeatures, it may be in the same workspace.
if str(fc) != str(os.path.split(clipFeatures)[1]):
   GP.Clip(fc, clipFeatures, outFeatureClass,
    clusterTolerance)
fc = fcs.Next()
```

When there are no more names in the enumeration, a null or empty string will be returned. Python will evaluate this as false, which will cause the while loop to exit. The next value from the enumeration must be retrieved before the end of the indented block so it is evaluated correctly at the start of the loop. The ValidateTableName method is used to ensure the output name is valid for the output workspace. Certain characters, such as periods or dashes, are not allowed in geodatabases, so this method will return a name with valid characters in place of invalid ones. It will also return a unique name so no existing data is overwritten. Chapter 7 fully explains how to use this method and how to work with the geodatabase.



```
##Script Name: Clip Multiple Featureclasses
##Description: Clips one or more shapefiles from a folder and places the
##clipped feature classes into a Geodatabase.
##Created By: Insert name here.
##Date: 11/04/2003

#Import standard library modules
import win32com.client, sys, os

#Create the Geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

#Set the input workspace
GP.workspace = sys.argv[1]

#Set the clip featureclass
clipFeatures = sys.argv[2]

#Set the output workspace
outWorkspace = sys.argv[3]

#Set the cluster tolerance
clusterTolerance = sys.argv[4]

try:

    #Get a list of the featureclasses in the input folder
    fcs = GP.ListFeatureClasses()

    #Loop through the list of featureclasses
    fcs.Reset()
    fc = fcs.Next()

    while fc:
        #Validate the new feature class name for the output workspace.
        outFeatureClass = GP.ValidateTableName(fc, outworkspace)

        #Clip each feature class in the list with the clip feature class.
        #Do not clip the clip_features, it may be in the same workspace.
        if str(fc) != str(os.path.split(clipFeatures)[1]):
            GP.clip_analysis(fc, clipFeatures, outFeatureClass,
            clusterTolerance)
        fc = fcs.Next()

except:
    GP.AddMessage(GP.GetMessages(2))
    print GP.GetMessages(2)
```

The os module's path object is used to manipulate the clip feature classes path, so only the feature class name is evaluated in an expression, not the entire path. The Clip tool is accessed as a method of the geoproccessor, using the various string variables as parameter values.

10. Add the following lines to complete the script:

```
except:
   GP.AddMessage(GP.GetMessages(2))
   print GP.GetMessages(2)
```

The `except` statement is required by the earlier `try` statement, otherwise a syntax error will occur. If an error occurs during execution, the code within the except block will be executed. In this case, any message with a severity value of 2, indicating an error, will be added to the geoprocessor in case the script is used as the source of a tool. All error messages are also printed to the standard output in case the script is run outside a tool.

11. Save the script by clicking the Save button on the Standard toolbar.

12. Add the following comments to the top of your script:

```
##Script Name: Clip Multiple Featureclasses
##Description: Clips one or more shapefiles
##from a folder and places the clipped
##feature classes into a Geodatabase.
##Created By: Insert name here.
##Date: 11/04/2003
```

Scripts should be well commented. Each script should contain a heading section that describes what it does, who created it, and when it was created, along with comments throughout the script itself that explain what it is doing. Use the pound symbol (#) to indicate a comment, which is a text string that should not be interpreted by Python. All text following a pound symbol will be ignored by the interpreter.

Coding Guidelines

Python enforces certain coding standards, such as indentation being part of the syntax and variable name restrictions. Other languages may not have such standards, but it is a good idea to follow a general set of rules so your scripts and the scripts of others inside and outside your organization provide good readability, portability, and consistency. The following points are suggestions and can be used to create your own coding guidelines.

- Variable names should start with a character and avoid using special characters such as an asterisk. If the variable name consists of several words, capitalize the first letter of each word (remember that Python and other languages are case sensitive), except for the first word. For example,

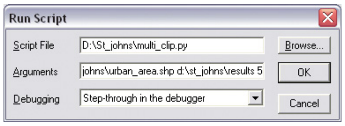
  ```
  myIntegerVariable = 1
  ```
- Use descriptive variable names and avoid using slang terms or abbreviations.
- Script names should follow the variable naming guidelines above. Python scripts should always have a .py extension.
- Use indentation to show the structure of a program. Python enforces this, but other languages do not. Use two or four spaces to define each logical level. Align the beginning and end of statement blocks, and be consistent.
- Use arguments to a script and avoid overuse of global variables, as they can cause unexpected results when they are accessed by a number of modules or scripts.
- Avoid duplication of code by placing commonly used functions in a common module that can be shared between scripts.
- If a script has arguments, it should validate the input values and return a suitable message if an invalid value is specified.
- Scripts should be well commented. Each logical section should have an explanation.
- Each script tools or function should have a header containing the script's name, a description of how it works, its requirements, who wrote it, and when it was written.

Inevitably, errors occur when you write and execute scripts. Syntax errors may be caught by Python before the script is run by running a syntax check, but other problems caused by typing errors, invalid property or method names or invalid parameter values can only be caught during the execution of the script. Without a debugging environment, you are left with the option of inserting `print` statements at critical points of the script so you can trace its execution path and variable values. A debugging environment lets you step through the program and interrogate variables, check object validity, and evaluate expressions.

Executing and debugging your script

1. Click the Check button on the standard toolbar to check for syntax and indentation errors in your script. If an error is found, the cursor will be placed at that location. Correct the syntax and check it again until there are no errors.

2. Click the Run button to open the Run Script dialog box.


Run Script

Script File: D:\St_johns\multi_clip.py

Arguments: johns\urban_area.shp d:\st_johns\results 5

Debugging: Step-through in the debugger

3. Enter the parameters required by the script: an input workspace, a clip feature class, an output workspace, and a cluster tolerance. Below is an example:

```
d:\st_johns d:\st_johns\urban_area.shp d:\st_johns\results 5
```

If you have the geoprocessing tutorial installed, for example, in your C:\Program Files folder, you can use the \ArcTutor\Geoprocessing\San_Diego folder as input. Specify \ArcTutor\Geoprocessing\San_Diego\study_quads.shp as the clip feature class, any writable folder as the output workspace, and 10 for the cluster tolerance. The cluster tolerance is optional in the Clip tool, so it could be omitted if you want to use the default cluster tolerance for each input feature class.

4. Choose “Step-through in the debugger” as the Debugging option and click OK.

Two windows will be opened, one to show the value of expressions you define, while the other displays the value of variables in the system’s call stack. The cursor will be placed on the first line of the script that will be interpreted by Python, which is the `import` statement. Adjust the size of the Python application window as needed to accommodate the new windows.

5. Click the Step button on the Debugging toolbar. This will execute the current line of code and move to the next.

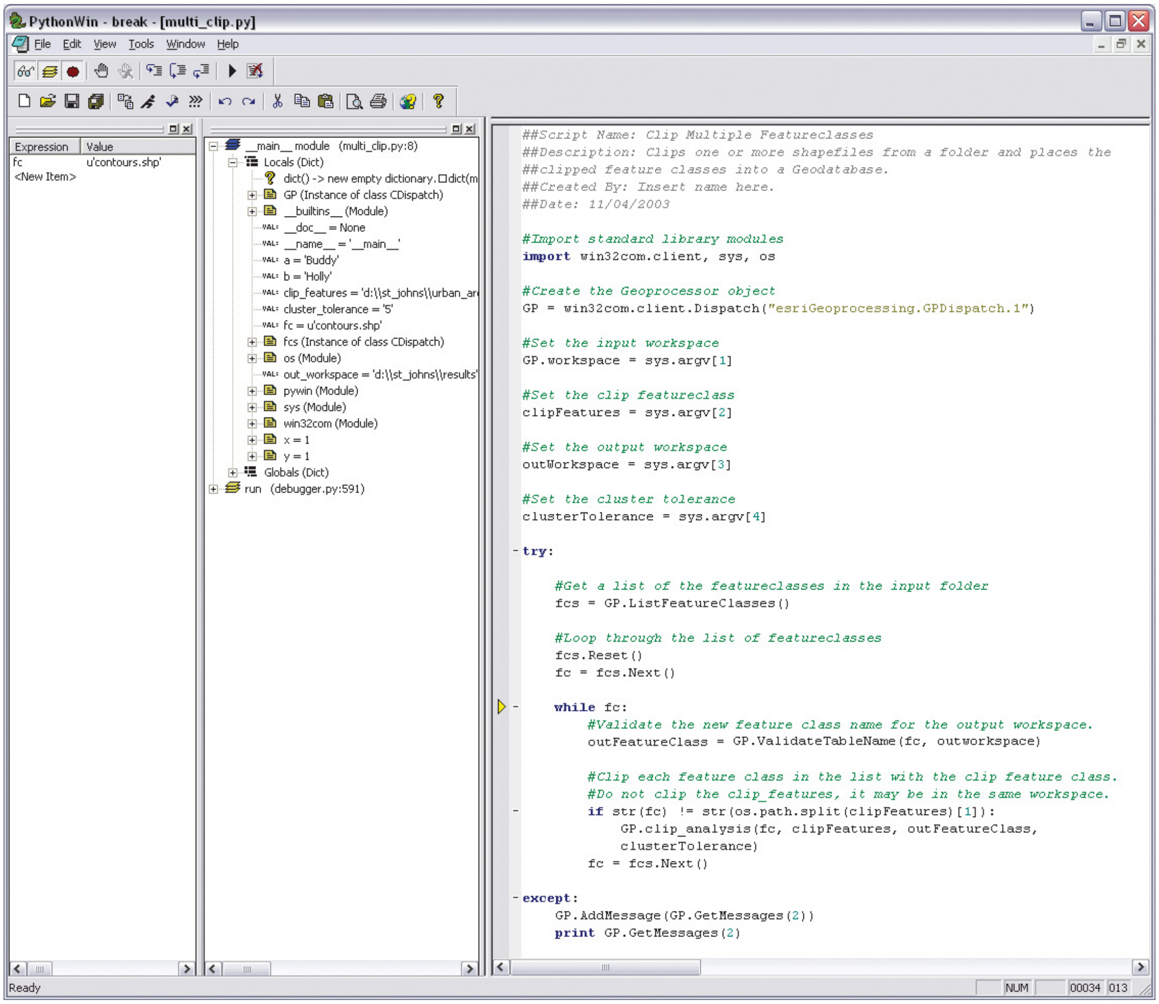
A new window containing the `win32com` module has now opened, as the next line of code to be executed is in that module. Importing a module will, in turn, import all the modules it imports. You do not want to step through the code in this module, as you are only interested in the code contained in the `multi_clip` module.

6. Click the Step Out button twice to step out of the `win32com` module and go back to the multi_clip script.

Use the Step Over button to avoid stepping into any code called by a line and to move on to the next line in the current module.

7. Click the Step Over button to move to the next line. This will create the geoprocessor object and will take several seconds due to its size.

8. Click the Step Over button four times so that the cursor is located at the `try` statement.


PythonWin - break - [multi_clip.py]
File Edit View Tools Window Help
Ready
NUM
00034
013

Expression	Value
fc	u'contours.shp'
<New Item>	

```
__main__ module   (multi_clip.py:8)
    Locals (Dict)
        dict() -> new empty dictionary.□dict(m
        GP (Instance of class CDispatch)
        __builtins__ (Module)
        __doc__ = None
        __name__ = '__main__'
        a = 'Buddy'
        b = 'Holly'
        clip_features = 'd:\\st_johns\\urban_ar
        cluster_tolerance = '5'
        fc = u'contours.shp'
        fcs (Instance of class CDispatch)
        os (Module)
        out_workspace = 'd:\\st_johns\\results'
        pywin (Module)
        sys (Module)
        win32com (Module)
        x = 1
        y = 1
    Globals (Dict)
run   (debugger.py:591)
```

```python
##Script Name: Clip Multiple Featureclasses
##Description: Clips one or more shapefiles from a folder and places the
##clipped feature classes into a Geodatabase.
##Created By: Insert name here.
##Date: 11/04/2003

#Import standard library modules
import win32com.client, sys, os

#Create the Geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

#Set the input workspace
GP.workspace = sys.argv[1]

#Set the clip featureclass
clipFeatures = sys.argv[2]

#Set the output workspace
outWorkspace = sys.argv[3]

#Set the cluster tolerance
clusterTolerance = sys.argv[4]

try:

    #Get a list of the featureclasses in the input folder
    fcs = GP.ListFeatureClasses()

    #Loop through the list of featureclasses
    fcs.Reset()
    fc = fcs.Next()

    while fc:
        #Validate the new feature class name for the output workspace.
        outFeatureClass = GP.ValidateTableName(fc, outworkspace)

        #Clip each feature class in the list with the clip feature class.
        #Do not clip the clip_features, it may be in the same workspace.
        if str(fc) != str(os.path.split(clipFeatures)[1]):
            GP.clip_analysis(fc, clipFeatures, outFeatureClass,
            clusterTolerance)
        fc = fcs.Next()

except:
    GP.AddMessage(GP.GetMessages(2))
    print GP.GetMessages(2)
```

9. In the Stack View window, click the _main_module node.

10. Click the Locals (Dict) node to display all local variables and their values.

String variables have a "u" preceding the value to indicate it is a unicode string. Unicode is a 16-bit character set standard designed to facilitate multiple languages.

You will see all the variables you have set in the Interactive window and the multi_clip script, as well as the modules you have imported. Use the Stack window to check the scope of your variables and their values, as well as the contents of all available modules.

11. Click the Step Over button three times to move the cursor to the `while` statement. The three lines you execute will create the list of available feature classes and populate the variable fc with the first name.

12. In the Watch window, click <New Item> and enter fc. Press Enter to change the expression to the variable name. You will now see the value of the variable.

13. Click the Step Over button twice to first set the output workspace and move to the `if` statement.

14. Click the Step Over button to move into the `if` statement's code block where the Clip tool will be executed.

15. Click the Step Over button twice to execute the tool and retrieve the next feature class name from the feature class names enumeration. Notice the value of the fc variable in the Watch window.

16. Click the Watch window and Stack View window buttons on the Debugging toolbar so they are closed.

17. Click the Interactive window button . Add the following code and press Enter:

```
GP.GetMessages()
```

All the messages from the Clip tool will be printed to the Interactive window. Any object and variable may be used in the Interactive window during a debug session, so it can be used for object interrogation and variable manipulation.

18. Click the Close button on the Debugging toolbar to stop the execution of the script.

Setting breakpoints

There are several other debugging options when you run a script. In the previous exercise, you stepped through the script line by line, but you may want to run the script and only have it stop at defined points called breakpoints.

1. Place your cursor on the following line of code in the multi_clip script:

```
fcs = GP.ListFeatureClasses()
```

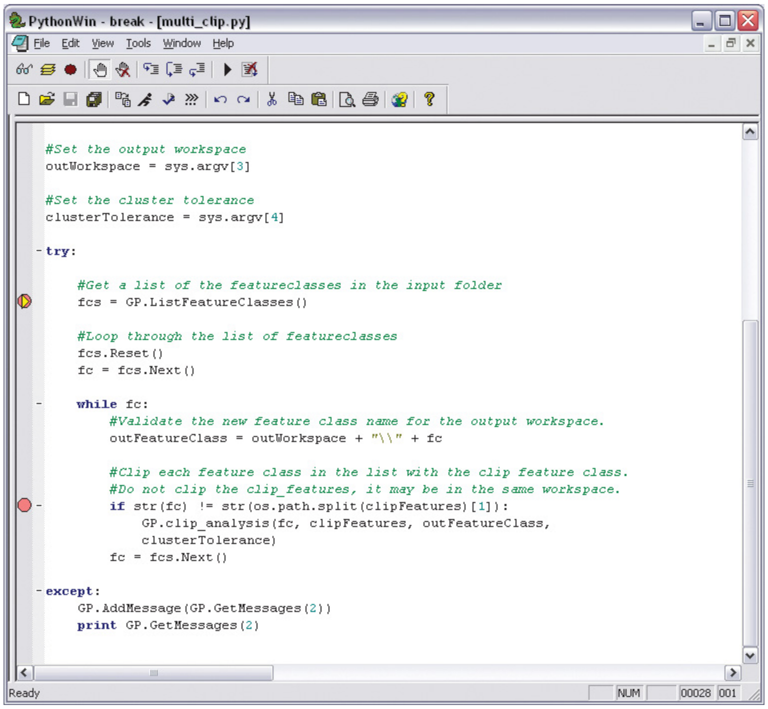
2. Click the Toggle Breakpoint button on the Debugging toolbar to place a breakpoint for that line.

A breakpoint symbol will appear on the far left margin of the script. It can be removed by clicking the Toggle Breakpoint button or clicking the Clear All Breakpoints button.

3. Click the Run button.

4. Click the Debugging dropdown list and click "Run in the debugger" on the Run Script dialog box.

5. Click OK to execute the script. The script will now stop where the breakpoint is set so you can step through the script from that point.


PythonWin - break - [multi_clip.py]
File Edit View Tools Window Help

```
#Set the output workspace
outWorkspace = sys.argv[3]

#Set the cluster tolerance
clusterTolerance = sys.argv[4]

try:

    #Get a list of the featureclasses in the input folder
    fcs = GP.ListFeatureClasses()

    #Loop through the list of featureclasses
    fcs.Reset()
    fc = fcs.Next()

    while fc:
        #Validate the new feature class name for the output workspace.
        outFeatureClass = outWorkspace + "\\" + fc

        #Clip each feature class in the list with the clip feature class.
        #Do not clip the clip_features, it may be in the same workspace.
        if str(fc) != str(os.path.split(clipFeatures)[1]):
            GP.clip_analysis(fc, clipFeatures, outFeatureClass,
            clusterTolerance)
        fc = fcs.Next()

except:
    GP.AddMessage(GP.GetMessages(2))
    print GP.GetMessages(2)
```

6. Click the Close button to stop the script and exit the Debugging window.

Many breakpoints can be added to a script so you can jump from one line to another during execution.

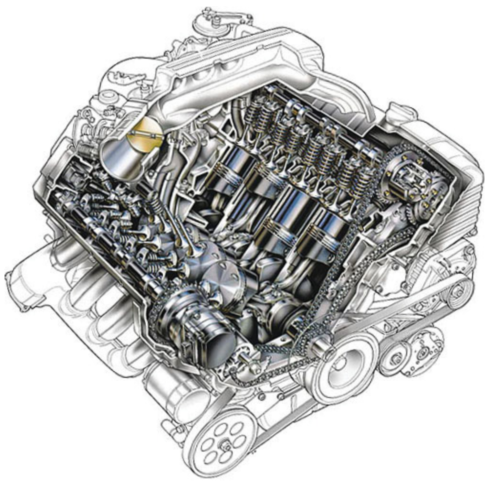
The Go button ▶ will continue executing the script until the next breakpoint or the last line of code.

Congratulations, you have just written and debugged your first geoprocessing script. The remaining chapters will discuss the capabilities of the geoprocessor and provide more examples of how to work in Python.

3

Using tools

Scripts are the engines that run geoprocessing operations for many types of tasks. They can be used to automate tasks, such as data conversion, or generate geodatabases. If scripts are engines, tools can be thought of as the pistons that drive the engine. In this chapter you will learn how to create the geoprocessor and run tools within a script.



The Component Object Model is a software architecture that allows applications to be built from binary software components. COM is the underlying architecture that forms the foundation for ArcGIS.

All geoprocessing tools are accessible through a single object called the Geoprocessor in ArcGIS. This high-level object exposes each tool as a native method. The geoprocessor is an ArcObjects component, accessible in Visual Basic or other COM compliant languages. It also supports the COM interface IDispatch, which enables text-based interpretative languages to utilize COM objects. Most scripting languages support COM using IDispatch, such as VBScript, Python, JScript, and Perl, making the geoprocessing tools available to these languages.

The IDispatch interface exposes objects, methods, and properties to programming tools and other applications that support Automation. COM components implement the IDispatch interface to enable access by Automation clients, such as Python, VBScript, and JScript.

All geoprocessing capabilities of ArcGIS are exposed as methods of the geoprocessor object, as it takes advantage of the dynamic nature of IDispatch to create a single access point for running tools, including user-defined models. The advantage of this approach over the one where a simple-to-use COM interface is written to encapsulate each geoprocessing tool is that there is no need to write the simple interface. Only the geoprocessing COM function needs to be written, and the rest is taken care of by the geoprocessing framework.

Creating the object

Creating the geoprocessor object within the script is easy, using the scripting environments standard call for instantiating IDispatch objects. Before this is done, the Python `win32com` module must be loaded using the Import command. This module enables the COM IDispatch communication within Python. All geoprocessing scripts must import this module to instantiate the geoprocessor object so it can be seen as standard code that all your scripts have at the top.

```
# Import COM Dispatch module
from win32com.client import Dispatch

# Create the geoprocessor object and print the usage of the clip tool
GP = Dispatch("esriGeoprocessing.GPDispatch.1")
print GP.usage("clip_analysis")
```

Once you have the object, you can run any tool as a method. The standard toolboxes that are installed with ArcGIS are immediately available once the geoprocessor is created. These are:

- Analysis Tools
- Conversion Tools
- Data Management Tools
- Linear Referencing Tools
- Geocoding Tools
- Cartography Tools

Adding and removing toolboxes

The geoprocessor only knows about the tools in toolboxes that it has been made aware of. Toolboxes may be found in many different folders or geodatabases, so it would be impossible for the geoprocessor object to be aware of all tools that exist in a system. To make a tool accessible to the geoprocessor, the tool's toolbox must be added to the geoprocessor so it is aware of its location and parameters.

Scripts and models may be added to custom toolboxes to create new tools. System tools may also be added to toolboxes to create custom collections of tools that are stored in a folder or a geodatabase. Refer to Chapter 4 of Geoprocessing in ArcGIS *to learn more about toolboxes.*

```
# Add a toolbox with a model to the geoprocessor and set the workspace
GP.AddToolbox("d:/Myproject/MyTools.tbx")
GP.Workspace = "d:/Myproject"
# Run the model with its required parameters
GP.BestPath("start.shp","destination.shp","results.shp")
```

A toolbox may also be removed from the geoprocessor so its tools are not accessible from the geoprocessor. Keeping the number of known toolboxes to a minimum decreases the chances of having multiple tools with the same name being accessed at the same time, which could result in tool name ambiguity.

```
# Remove toolbox from the geoprocessor
GP.RemoveToolbox("d:/Myproject/MyTools.tbx")
```

Scripting is an ideal way to run a tool many times with different parameters automatically. This is called batch processing.

The geoprocessing object exposes a number of methods and properties to better support the scripting experience. Methods, which may also be called functions, can be used to list certain datasets, retrieve a dataset's properties, validate a table name before adding it to a geodatabase, or perform many others tasks. These methods are only available from the geoprocessor, not as tools in ArcGIS applications, as they are meant for scripting. Each method will be discussed in detail in this book. The example below shows how the ListFeatureClasses method can be used to create a list object that contains the names of all the feature classes in a workspace.

```
GP.workspace = "d:/MyData"
out_workspace = "d:/MyData/Results/"
clip_features = "d:/MyData/TestArea/Boundary.shp"
# Get a list of the feature classes in the workspace
fcs = GP.ListFeatureClasses()

# Loop through the list of feature classes
fcs.reset()
fc = fcs.next()
while fc:
   # Set the outputname for each output to be the same as the input
   output = out_workspace + fc
   # Clip each input feature class in the list.
   GP.clip_analysis(fc, clip_features, output, 5)
   fc = fcs.next()
```

Enumerations are lists without a known count. Using a looping structure, a script can process the values of an enumeration by simply checking to see if there is another value to enumerate. Refer to Chapter 4 for working with enumerations.

Toolbox tools may be accessed as methods directly from the geoprocessor as well. Due to the dynamic nature of the IDispatch COM interface, the geoprocessor can access any tool that has been registered with the system, so geoprocessing operations such as overlays or buffering can be easily executed, as well as models and other scripts. In the example above, the Clip tool from the Analysis Toolbox is used to clip a number of feature classes in a batch operation.

Running a tool

Each geoprocessing tool has a fixed set of parameters that provides the tool with the information it needs for execution. Tools usually have input parameters that define the dataset or datasets that will typically be used to generate new output data. Parameters have several important properties:

- **Name:** Each tool parameter has a unique name.
- **Type:** The type of data expected, such as feature class, integer, string and raster.
- **Direction:** The parameter defines input or output values
- **Required:** Either a value must be provided for a parameter or it is optional

When a tool is used in a script, its parameter values must be correctly set so it can execute when the script is run. The documentation of each tool clearly defines its parameters and properties. Once a valid set of parameter values are provided, the tool is ready to be executed.

Parameters are specified either as strings or objects. Strings are simply text that uniquely identify a parameter value, such as a path to a dataset or a keyword.

All the system toolboxes have a set alias property. Below are the toolbox names and their aliases:

Data Management Tools—Management

Analysis Tools—Analysis

Conversion Tools—Conversion

Geocoding Tools—Location

Coverage Tools—Arc

ArcGIS Spatial Analyst Tools—SA

ArcGIS 3D Analyst™ Tools—3D

Most tool parameters can be specified as a simple string, with only complex parameters, such as a spatial reference, requiring an object. How to create objects and use them as parameters will be discussed in detail later in this chapter. In the following code example, input and output parameters are defined for the Buffer tool. In this case, two string variables are used to define the input and the output so the call to the tool is easier to read. Other parameters are defined by strings, which must always be quoted.

```
Output = "D:/St_Johns/data.mdb/roads_Buffer"
Roads = "D:/St_Johns/data.mdb/roads"
# Set the toolbox to Analysis to avoid conflicts with other Buffer tools
# and run Buffer
GP.Toolbox = "Analysis"
GP.Buffer(Roads, Output, "distance", "FULL", "ROUND", "NONE", "")
```

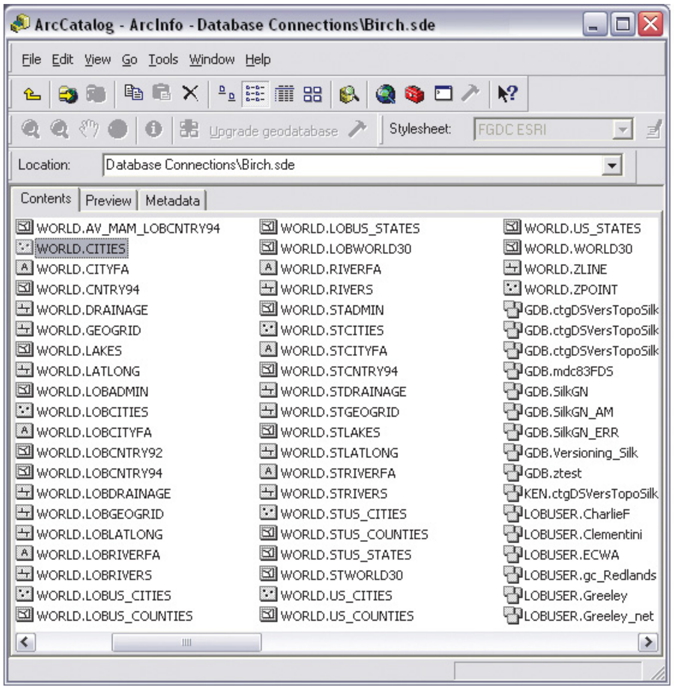
Geodatabase feature datasets and standalone feature classes may have the same ArcCatalog path. Typically tools work with one or the other. For those tools that may work with either, such as the Copy tool, the specific data type may be specified to avoid ambiguity.

ArcCatalog paths

When the path to a dataset is specified as a tool or environment setting parameter, it must be the same as the path reported in the ArcCatalog™ Location toolbar. Tools use ArcCatalog to find geographic data using an ArcCatalog path. This path is a string and is typically unique to each dataset, containing either its folder location, database connection, or URL in the case of streaming Internet Mapping Server (IMS) or server data. If the dataset you want to use as a tool parameter cannot be seen in ArcCatalog, then it cannot be used, except for some file specific parameters, such as an ArcInfo™ interchange file (e00) or raster colormap file.

The ArcCatalog path for a shapefile is simply the path to the folder containing the shapefile and the shapefile's name, including its .shp extension. A shapefile containing roads located in the folder C:\GrosMorne would have an ArcCatalog path of "C:\GrosMorne\roads.shp". A coverage in that same folder containing a study area polygon would have a similar ArcCatalog path of "C:\GrosMorne\StudyArea". A shapefile's extension is used to differentiate between a shapefile and a coverage having the same name in the same folder. INFO and dBASE tables act in the same fashion, except the dBASE table has a .dbf extension.

Feature classes in a personal geodatabase reside in an Access database file, and enterprise geodatabase feature classes are found in a Relational Database Management System (RDBMS). The ArcCatalog path to a personal geodatabase has the disk location of the Access file. A feature class name is simply added to that path if it is standalone, resulting, for example, in a path of "C:\GrosMorne\Data.mdb\rivers". If a feature class is contained by a feature


ArcCatalog - ArcInfo - Database Connections\Birch.sde
File Edit View Go Tools Window Help
Upgrade geodatabase
Stylesheet:
FGDC ESRI
Location:
Contents
Preview
Metadata

Location: Database Connections\Birch.sde

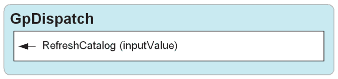
WORLD.AV_MAM_LOBCNTRY94	WORLD.LOBUS_STATES	WORLD.US_STATES
WORLD.CITIES	WORLD.LOBWORLD30	WORLD.WORLD30
WORLD.CITYFA	WORLD.RIVERFA	WORLD.ZLINE
WORLD.CNTRY94	WORLD.RIVERS	WORLD.ZPOINT
WORLD.DRAINAGE	WORLD.STADMIN	GDB.ctgDSVersTopoSilk
WORLD.GEOGRID	WORLD.STCITIES	GDB.ctgDSVersTopoSilk
WORLD.LAKES	WORLD.STCITYFA	GDB.ctgDSVersTopoSilk
WORLD.LATLONG	WORLD.STCNTRY94	GDB.mdc83FDS
WORLD.LOBADMIN	WORLD.STDRAINAGE	GDB.SilkGN
WORLD.LOBCITIES	WORLD.STGEOGRID	GDB.SilkGN_AM
WORLD.LOBCITYFA	WORLD.STLAKES	GDB.SilkGN_ERR
WORLD.LOBCNTRY92	WORLD.STLATLONG	GDB.Versioning_Silk
WORLD.LOBCNTRY94	WORLD.STRIVERFA	GDB.ztest
WORLD.LOBDRAINAGE	WORLD.STRIVERS	KEN.ctgDSVersTopoSilk
WORLD.LOBGEOGRID	WORLD.STUS_CITIES	LOBUSER.CharlieF
WORLD.LOBLATLONG	WORLD.STUS_COUNTIES	LOBUSER.Clementini
WORLD.LOBRIVERFA	WORLD.STUS_STATES	LOBUSER.ECWA
WORLD.LOBRIVERS	WORLD.STWORLD30	LOBUSER.gc_Redlands
WORLD.LOBUS_CITIES	WORLD.US_CITIES	LOBUSER.Greeley
WORLD.LOBUS_COUNTIES	WORLD.US_COUNTIES	LOBUSER.Greeley_net

Use the Location toolbar in ArcCatalog to help determine the path to a dataset. In the example above, the path for the selected feature class is "Database Connections\Birch.sde\WORLD.CITIES".

The geoprocessor uses ArcCatalog to reference geographic data. If the geoprocessor is created in an application that has an ArcCatalog already instantiated, it will use that instance, otherwise, it will create one for the lifetime of the geoprocessing object.

dataset, the feature dataset's name must precede the feature class name. Feature class paths must always include the feature dataset name when applicable, otherwise, a tool will not recognize the path.

Instead of a path to an Access file, paths to data in an enterprise geodatabase contain the location of the file defining the database connection. The default location for this information is Database Connections in ArcCatalog, so a typical path to a standalone feature class in an enterprise geodatabase may appear as "Database Connections\Connection to GrosMorne.sde\reed.roads". Use the Location toolbar in ArcCatalog to check a dataset or workspace path.



The geoprocessor will not see data correctly if a script copies, deletes, or modifies data without using one of the geoprocessor's methods or a geoprocessing tool. Copying a set of shapefiles using the file system can cause the geoprocessor's internal ArcCatalog to become out-of-date, as it does not see these changes. The geoprocessor will not refresh automatically to see these changes, as ArcCatalog does not monitor all changes in a system. In a case such as this, the RefreshCatalog method must be used to make sure the ArcCatalog that the geoprocessor uses references data correctly for a particular location.

To learn more about working with ArcObjects, visit the ArcGIS Developer online website (http://arcgisdeveloperonline.esri.com).

Using ArcObjects as tool input

ESRI® ArcObjects is the development platform for the ArcGIS family of applications such as ArcMap™, ArcCatalog™, and ArcScene™. The ArcObjects software components expose the full range of functionality available in ArcInfo and ArcView® to software developers. These objects may be used to manage geographic data, such as the contents of a geodatabase, shapefiles, and coverages, to name a few. If you are accustomed to working with ArcObjects, you can continue with that object model when working with the geoprocessor. An ArcObject may be used instead of an ArcCatalog path when defining an input parameter to a tool if the parameter accepts layers as valid input. For example, an IFeatureClass object may be used to define the input to the Clip tool in the Analysis toolbox, while an IRasterDataset object may be used as input to the Slope tool. The CreateFeatureclass tool will not accept an ArcObject as the input location, which could be a folder, geodatabase workspace, or geodatabase feature dataset, as none of these data types may be represented as a layer. An ArcCatalog path must be used to define newly created tool output. Append updates existing data, so its output may be defined using an ArcObject, such as an IFeatureLayer. Below is an example of using an ArcObject as input to a tool in VBA:

The geoprocessor object may be used in Visual Basic (VB) or Visual Basic for Applications (VBA) to execute any geoprocessing tool. When used in VBA, any resulting tool messages are captured in the Command Line window. The geoprocessing history model is also updated with a record of any tool that is executed.

```
Public Sub ClipFC()

  Dim filePath As String
  filePath = "D:\st_johns"
  Dim inputName As String
  Dim clipName As String
  inputName = "roads.shp"
  clipName = "urban_area.shp"

  Dim pWorkspace As IWorkspace
  Dim pFact As IWorkspaceFactory
  Set pFact = New ShapefileWorkspaceFactory
  Set pWorkspace = pFact.OpenFromFile(filePath, 0)
  Dim pFWorkspace As IFeatureWorkspace
  Set pFWorkspace = pWorkspace

  Dim pInfc As IFeatureClass
  Dim pClipfc As IFeatureClass
  Set pInfc = pFWorkspace.OpenFeatureClass(inputName)
  Set pClipDS = pFWorkspace.OpenFeatureClass(clipName)

  Dim pGP As Object
  Set pGP = CreateObject("esriGeoprocessing.GPDispatch.1")

 On Error GoTo EH
  pGP.Workspace = filePath
  pGP.Clip_analysis pInfc, pClipfc, filePath & "\clipfc.shp"

  Exit Sub
EH:
  MsgBox pGP.GetMessages(), vbOKOnly, "Clip"
End Sub
```

Tool return values

The geoprocessor always returns the output value(s) of the tool when it is executed. Typically, this is the path to the output dataset produced or updated by the tool, but it may be other value types such as a number or a boolean. If there is more than one output for a tool, the values are returned in a multivalue string, which consists of multiple strings separated by a semicolon.

Output values are always returned to the script from the geoprocessor. Return values are necessary when a tool has no output dataset, instead, it has an output scalar value, such as an integer or boolean. They are also helpful when working with multiuser databases, as output names are qualified with a user's name and table space when outputs are written into the database. This topic is covered in more detail in Chapter 7. Below are several examples of how return values are captured and what their values could be:

A backslash (\) is a reserved character indicating line continuation or an escape character in Python. When specifying a path, use two backslashes instead of one to avoid a syntax error. A forward slash (/) may be used in place of a backslash. A string literal may also be used by placing "r" before a string containing a backslash so it is interpreted correctly. All examples in this book use forward slashes.

Example 1:

```
GP.Toolbox = "Analysis"
GP.Workspace = "D:/St_Johns/data.mdb"
outvar = GP.Clip("roads","urban_area","urban_roads")
# Print output feature class name D:/St_Johns/data.mdb/urban_roads
print outvar
```

Example 2:

```
GP.Toolbox = "Management"
GP.Workspace = "D:/St_Johns/data.mdb"
defgrid = GP.CalculateDefaultGridIndex("urban_roads")
# Print return value, which is a double precision number
print defgrid
```

Example 3:

```
GP.Toolbox = "Analysis"
GP.Workspace = "Database Connections/Connection to st_johns.sde"
outvar = GP.Clip("roads","urban_area","urban_roads")
# Print return value of fully qualified output name which will be
# Database Connections\Connection to jreinhart.sde\gp.city.urban_roads
print outvar
```

Tool names and name conflicts

Tools have a name and label property. A tool name must be unique for the toolbox containing the tool, but the label has no restrictions. For example, a tool may be named "CalculatePath", while its label is "Calculate Best Path". There must be no other tool in that toolbox named "CalculatePath", but other tools may have the same label. The tool label is used for displaying the tool in an ArcGIS application and for labeling the tool's dialog box. The tool name is used to execute the tool in the command line and within scripts. The name must not have spaces or other restricted characters such as percent symbols or slashes.

Setting the toolbox when using a tool is a good practice if the script being written may be used by others or in different contexts. It eliminates the possibility of tool name conflicts.

A script typically uses tools from more than one toolbox, as seen with the toolboxes ESRI delivers with ArcGIS. When using multiple toolboxes, it is possible that two or more toolboxes will contain a tool with the same name. When this happens, the geoprocessor is unable to determine which tool in which toolbox should be executed when the tool is referenced in a script. If the geoprocessor determines that the tool name is ambiguous, an error is returned.

Avoid referencing too many toolboxes in your script, as it increases the possibility of a tool name conflict. Add a toolbox when you need its tools; remove it when you no longer need them.

There are two ways of avoiding these tool name conflicts. The first is to specify which toolbox should be used as the default tool location when a tool is called by the geoprocessor. The second is to add a suffix to the name of the tool using the toolbox's alias when the tool is referenced by the geoprocessor. This will be discussed in the 'Using a toolbox alias' section below.

Setting the current toolbox

Each toolbox has a unique path. This path can be used to specify the first place the geoprocessor should look when a tool is executed. If that toolbox has a tool with the same name as a tool in another toolbox referenced by the geoprocessor, the geoprocessor will select the tool in the toolbox that has been explicitly set for the geoprocessor. A toolbox's path or alias may be used to explicitly set it. A toolbox's alias is a short name that is used to quickly identify it. The alias has the same naming restrictions as a tool and should be unique. The examples below show how to set the default toolbox for the geoprocessor:

Example 1:

```
# Set the default toolbox using the toolbox's path
GP.AddToolbox = "D:/St_Johns/Path Tools.tbx"
GP.Toolbox = "D:/St_Johns/Path Tools.tbx"
GP.Workspace = "D:/St_Johns/data.mdb"
GP.CalculatePath("start","destination","path")
```

All the system toolboxes have a set alias property. Below are the toolbox names and their aliases:

Data Management Tools—Management

Analysis Tools—Analysis

Conversion Tools—Conversion

Geocoding Tools—Location

Coverage Tools—Arc

ArcGIS Spatial Analyst Tools—SA

ArcGIS 3D Analyst Tools—3D

Example 2:

```
# Set the default toolbox using its alias
GP.AddToolbox = "D:/St_Johns/Path Tools.tbx"
GP.Toolbox = "PathTools"
GP.Workspace = "D:/St_Johns/data.mdb"
GP.CalculatePath("start","destination","path")
```

Using a toolbox alias

Often, only a few toolboxes are used in a script, so explicitly setting the toolbox may not be necessary for most tools known by the geoprocessor. In the rare case where a conflict is known to occur, it may be avoided by simply suffixing the tool name with the alias of the toolbox and an underscore. This additional information eliminates ambiguity for the geoprocessor, allowing it to execute the correct tool. Not all toolboxes have an alias, as it is an optional property. There may also be nonunique toolbox alias names, so only use a toolbox's alias when there is no chance of an alias name conflict; otherwise, explicitly set the default toolbox for the geoprocessor using the toolbox's path. In the example below, the script tries to access the Clip tool found in the Analysis Tools toolbox and the Clip tools found in the Coverage Tools toolbox. These tools perform very similar operations, but they are distinct, requiring different parameters. By using the alias of each toolbox, a tool name conflict is avoided.

```
# Clip the geodatabase feature class roads with the urban_area coverage
GP.Workspace = "D:/St_Johns/data.mdb"
# Use the Analysis Tools alias to specify its Clip tool
GP.clip_analysis("roads","urban_area/polygon","urban_roads")
# Clip the coverage rivers with the urban_area coverage
GP.Workspace = "D:/St_Johns"
# Use the Coverage Tools alias to specify its Clip tool
GP.clip_arc("rivers","urban_area","urban_roads")
```

Environment settings are divided into five categories: General, coverage, geodatabase, raster geodatabase and raster analysis. Refer to the ArcGIS Help system to learn about available settings and what tools they affect.

Environment setting names are unique and can be found in the ArcGIS Help system. Environment names have the same restrictions as tool names, so restricted characters, such as spaces or slashes, are not allowed.

Each tool has a set of parameters it uses to execute an operation. Some of these parameters are common among tools, such as a tolerance or output location. These parameters may obtain their default values from a geoprocessing environment that all tools utilize during their operation. When a tool is executed, the current environment settings may also be used as global input parameter values. Settings such as an area of interest, the spatial reference of the output dataset, and the cell size of a new raster dataset may all be specified in the geoprocessing environment. For a complete list and description of the geoprocessing environment settings, refer to Chapter 6 in *Geoprocessing in ArcGIS*.

A script may be executed in several different ways. It may be run by a script tool in an ArcGIS application, such as ArcMap. It may also be run from another script or by itself from a command prompt. When a script is run inside a tool from an ArcGIS application or from another geoprocessing script, the environment settings used by the calling application or script are passed to it. These settings become the default settings used by the tool's script when it is executed. The called script may alter the settings passed to it, but those changes are only used within that script or by any other tool it may call. Changes are not passed back to the calling script or application. The environment model can best be described as cascading, where values flow down to any process that uses the geoprocessing environment.

Environment settings are exposed as properties of the geoprocessor in the scripting environment. These properties may be used to retrieve the current values or to set them. Each environment setting has a name and a label. Labels are displayed on the Environment Settings dialog box in ArcGIS. Names are used in scripts or at the command line in ArcGIS applications. Below are several examples of how to use environment values:

Example 1: Setting Environment Values

```
# Set the workspace environment setting
GP.Workspace = "D:/St_Johns/data.mdb"
# Set the cluster tolerance environment setting
GP.ClusterTolerance = 2.5
# Calculate the default spatial grid index, half it and then
# set the spatial grid 1 environment setting
GP.SpatialGrid1 = pGP.CalculateDefaultGridIndex("roads") / 2
# Clips the roads by the urban area feature class
GP.Toolbox = “Analysis”
GP.Clip("roads","urban_area","urban_roads")
```

Example 2: Getting and Setting an Environment Value

```
# Check the current raster cell size and make sure it is a certain size
# for standard output
GP.Workspace = "d:/avalon/data"
if GP.CellSize < 10:
    GP.CellSize = 10
elif GP.CellSize > 20:
    GP.CellSize = 20
GP.HillShade_sa("island_dem","island_shade",300)
```

Saving and loading settings

Automatic transfer of settings is only done when a script is executed by a geoprocessing tool. When a geoprocessing script calls another geoprocessing script, the environments are not automatically passed to the called script, as there is no way for the first script to know that the second script will contain a geoprocessor.

The environment settings file is text stored in an XML schema, but it is not important for you to understand its format because it is only meant as a vehicle for persistence and transfer of settings from one geoprocessor to the next.

Environment values may be passed between modules as arguments. Saving and loading settings between executing modules is not an efficient way of sharing values. Using a settings file is valid when used by modules that are not run together, which is the intent of the adjacent example.

To facilitate the transfer of environment settings from one script to another and to save settings from one session to the next, settings may be saved to a file. A geoprocessor may then set its environments by loading a settings file. In the first example below, a script transfers its settings to a second script by saving them to a file and passing that file name as a parameter to a second script. The second example loads an environment settings file using a name passed as a script argument.

Example 1:

```
from win32com.client import Dispatch

# Create the GPDispatch object
GP = Dispatch("esriGeoprocessing.GPDispatch.1")
# Set the raster environment settings and the workspace
GP.CellSize = 25
GP.Mask = "D:/St_Johns/Landcover"
GP.Workspace = "D:/St_Johns"
# Save the environment settings
envfile = GP.SaveSettings("D:/St_Johns/settings")
# Call python script and pass file name as argument
os.system('MyHillshade.py ' + envfile)
```

Example 2:

```
import os
from win32com.client import Dispatch

# Create the GPDispatch object
GP = Dispatch("esriGeoprocessing.GPDispatch.1")
# Get the input parameter value
envfile = sys.argv[1]
# Load the environment settings
envfile = GP.LoadSettings(envfile)
# Calculate hillshade
GP.Hillshade("city_dem","city_shade",250)
```

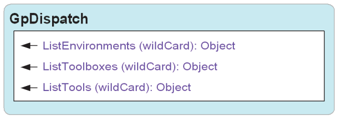
Tool parameters are usually defined using simple text strings. Dataset names, paths, keywords, field names, tolerances, and domain names are all simple to specify using a quoted string. Some parameters are harder to define using a single string, as they define a more complex parameter type that requires many properties. Spatial references, value tables, weighted overlay tables, and remap tables are examples of complex parameters, as they require a number of properties to be set before they can be used. Instead of using simple text strings to define these parameters, they are defined using objects that have all the required information defined as read/write properties. Refer to a tool's documentation to review its parameters and how they are specified in scripts. Each tool's documentation has a specific section for scripting with examples, so it will inform you if an object or a string is expected for a parameter and show how it is defined.

Parameter objects are created using the geoprocessor's CreateObject method. Once an object has been created, its properties must be set before it is used as a parameter. An object's properties are read or set in the same manner as geoprocessing environment settings. Below is an example of how a parameter object is created and used in a tool:

For more information about map projections and spatial references, refer to Understanding Map Projections *and* Building a Geodatabase.

```
# Create spatial reference object for new feature dataset
SR = GP.CreateObject("SpatialReference")
SR.Name = "Lambert Conformal Conic"
SR.StandardParallel1 = "46 00 00"
SR.StandardParallel2 = "52 00 00"
SR.CentralMeridian = "-61 00 00"
SR.LongitudeOfOrigin = "0 00 00"
SR.FalseEasting = "1000000"
SR.Domain = "1000000;0;3000000;5000000"
# Create new feature dataset
GP.CreateFeatureDataset("d:/st_johns/data.mdb/results",SR)
```

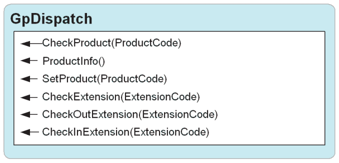
Listing tools, toolboxes, and environments


GpDispatch
ListEnvironments (wildCard): Object
ListToolboxes (wildCard): Object
ListTools (wildCard): Object

Depending on which toolboxes have been added to the geoprocessor, the geoprocessor may have access to several toolboxes, dozens of environment settings and hundreds of tools. There are three appropriately named methods on the geoprocessor to return a list of tools (ListTools), environment settings (ListEnvironments) or toolboxes (ListToolboxes). These methods have one option, a search string, or wild card, and they return an enumeration of name strings that can be looped through. Refer to Chapter 4 for more information about using enumerations. The example below shows how to access all the tools from the geoprocessor and print out their usage.

```
from win32com.client import Dispatch

# Create the Geoprocessor
GP = Dispatch("esriGeoprocessing.GPDispatch.1")

# Create a list of the default set of tools.
tools = GP.ListTools()

# Get the first tool name from the list.
tools.Reset()
tool = tools.Next()

# Loop through the list and print each tool's usage.
while tool:
   print GP.Usage(tool)
   tool = tools.Next()
```

Whenever a tool is executed in a script, an ArcGIS Desktop license is required. Tools from ArcGIS extensions, such as Spatial Analyst, require an additional license for that extension. If the necessary licenses are not available, a tool will fail and return appropriate error messages. The geoprocessor will always assume an ArcInfo license is required for execution of a script, so if a script does not explicitly set the product required by the tools it executes, an ArcInfo license will be initialized. If ArcView or ArcEditor™ are the only available desktop licenses, a script must explicitly set the product to ArcView or ArcEditor. The table below shows the methods available in the geoprocessor for checking, setting, and returning licences:


GpDispatch
CheckProduct(ProductCode)
ProductInfo()
SetProduct(ProductCode)
CheckExtension(ExtensionCode)
CheckOutExtension(ExtensionCode)
CheckInExtension(ExtensionCode)

Method(code)	Return Value	Meaning
SetProduct (ArcInfo, ArcEditor, ArcView)	CheckedOut	Successfully set the license
	AlreadyInitialized	License has already been set in the script
	NotLicensed	The requested license is not valid
	Failed	A system failure occured during the request
ProductInfo	NotInitialized	No license has been set
	ArcInfo	An ArcInfo license has been set
	ArcEditor	An ArcEditor license has been set
	ArcView	An ArcView license has been set
CheckProduct (ArcInfo, ArcEditor, ArcView)	AlreadyInitialized	License has already been set in the script
	Available	The requested license is available to be set
	Unavailable	The requested license is unavailable to be set
	NotLicensed	The requested license is not valid
	Failed	A system failure occured during the request
CheckExtension (Spatial, 3D, Geostats)	Available	The requested license is available to be set
	Unavailable	The requested license is unavailable to be set
	NotLicensed	The requested license is not valid
	Failed	A system failure occured during the request
CheckOutExtension (Spatial, 3D, Geostats)	NotInitialized	No desktop license has been set
	Unavailable	The requested license is unavailable to be set
	CheckedOut	Sucessfully set the license
CheckInExtension (Spatial, 3D, Geostats)	NotInitialized	No desktop license has been set
	Failed	A system failure occured during the request
	CheckedIn	The license has been returned successfully

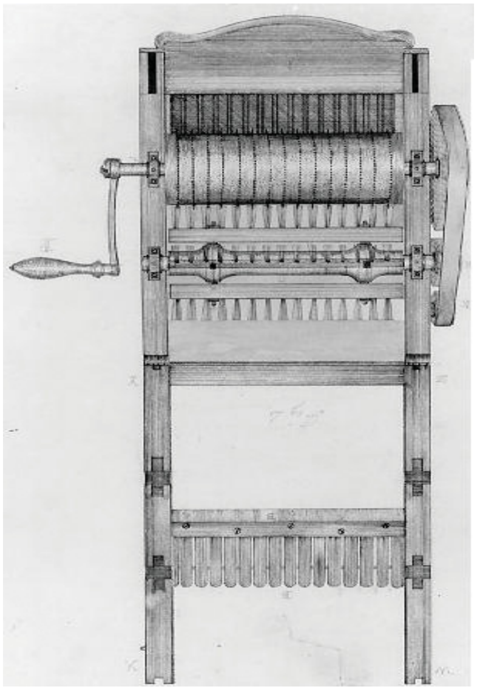
The SetProduct method is used to define the desktop license used by a script. A string is returned by the geoprocessor, indicating whether the license was initialized successfully. The CheckProduct method may be used to see which desktop licenses are available, while the ProductInfo method will report what the current product license is.

Licenses for extensions may be retrieved from a license manager and returned once they are no longer needed. The CheckExtension is used to see if a license is available to be checked out for a specific type of extension, while CheckOutExtension actually retrieves the license. Once the extension license has been retrieved by the script, extension tools will execute. Once a script is done with an extension's tools, the CheckInExtension method should be used to return the license to the license manager so other applications may use it. All checked-out extension licenses and set product licenses are returned to the license manager when a script completes. The following example will execute some Spatial Analyst tools and set the desktop product license to ArcView, as an ArcInfo license is not required to execute tools from an extension. The script would fail if the ArcView license is not explicitly set and no ArcInfo license is available, as a desktop license is required to execute extension tools.

```
# Import COM Dispatch module
from win32com.client import Dispatch

# Create the geoprocessor object
GP = Dispatch("esriGeoprocessing.GPDispatch.1")
GP.SetProduct("arcview")

try:
   if GP.CheckExtension("spatial") == "available":
      GP.CheckOutExtension("spatial")
   else:
      raise "LicenseError"

   GP.Workspace = "D:/GrosMorne"
   GP.HillShade("WesternBrook", "westbrook_hill", 300)
   GP.Aspect("WesternBrook", "westbrook_aspect")
   GP.CheckInExtension("spatial")

except "LicenseError":
   print "Spatial Analyst license is unavailable"
except:
   print GP.GetMessages(2)
```

A returned value of “failed” indicates there was a problem in communication with the license manager. A network error or a stopped license manager process are examples of the failure of some methods.

4 Batch processing



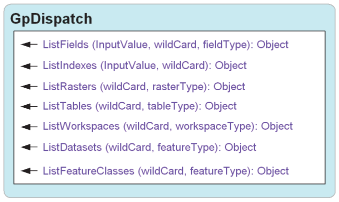
Many geoprocessing tasks are done repetitively and often. An easy and efficient way of automating these tasks is essential.

This chapter will show how to write scripts to work with many inputs and outputs using a number of the geoprocessor's functions and objects that were built for batch processing.

Companies and organizations that use a GIS typically have large amounts of information stored in many different data formats. Hundreds of shapefiles, geodatabase feature classes, coverages, grids, and tables may form the backbone of the organization and are used every day. New data may also be created at a staggering rate by converting data from one format to another. The tasks required to maintain current databases and create data suitable for analysis are often daunting because of the volume of data required for processing.

Scripts may be run outside an ArcGIS application at any time. By using an operating system's scheduling mechanism, you may have a script run at a regular date and time.

Scripts are used to meet these requirements, as they are an effient way of organizing data and executing operations. One of the foremost tasks in a batch processing script is cataloging the data that is available so it can iterate through the data during processing. The geoprocessor has a number of methods built specifically for creating such lists. These methods work with all different types of data and provide flexibility for restricting a search by name or data category. Below is a list of these methods and their syntax:


GpDispatch
ListFields (InputValue, wildCard, fieldType): Object
ListIndexes (InputValue, wildCard): Object
ListRasters (wildCard, rasterType): Object
ListTables (wildCard, tableType): Object
ListWorkspaces (wildCard, workspaceType): Object
ListDatasets (wildCard, featureType): Object
ListFeatureClasses (wildCard, featureType): Object

- **ListDatasets (Wild Card, Dataset Type):** Returns the datasets in the current workspace.
- **ListFeatureClasses (Wild Card, Feature Type):** Returns the feature classes in the current workspace.
- **ListFields (Input Value, Wild card, Field Type):** Returns a list of fields found in the input value.
- **ListIndexes (Input Value, Wild Card):** Returns a list of attribute indexes found in the input value.
- **ListRasters (Wild Card, Raster Type):** Returns a list of rasters found in the current workspace.
- **ListTables (Wild Card, Table Type):** Returns a list of tables found in the current workspace.
- **ListWorkspaces (Wild Card, WorkspaceType):** Returns a list of workspaces found in the current workspace.

The result of each of these methods is an enumeration, which is a list of values without a known count. An enumeration in scripting may contain any type of data, such as string, which could be, for example, a pathname to a dataset, a field, or a row from a table. Once the enumeration has been created with the values you want, you can loop through it in your script to work with each individual value.

*The wild card option uses an asterisk to mean any character. More than one asterisk may be used in the wild card string. For example, the wild card '*road*' can be used to find items that have the word road in their names.*

Parameters

The parameters of these methods are similar. A few, such as ListFields, require an input dataset value, as the items the methods are listing reside within a certain object or dataset. Other methods do not require an input dataset, as they list types of data in the current workspace that are defined in the environment settings. All methods have a wild card parameter, which is used to restrict the objects or datasets listed by name. A wild card defines a name filter, and all the contents in the newly created list must pass that filter. For example, you may want to list all the feature classes in a workspace that start with the letter *G*. The

A workspace is a directory, database, or dataset containing geographic data (for example, geodatabase, geodatabase feature dataset, coverage, folder) and other supporting data.

following example shows how this is done:

```
# Import COM Dispatch module
from win32com.client import Dispatch

# Create the geoprocessor object
GP = Dispatch("esriGeoprocessing.GPDispatch.1")
# Set the workspace. List all of the feature classes that start with 'G'
GP.Workspace = "D:/St_Johns/data.mdb"
fcs = GP.ListFeatureClasses("G*")
```

The list may also be restricted to match certain data properties, such as only polygon feature classes, integer fields, or coverage datasets. This is what the Type parameter is used for in all the methods. In the next example, the feature classes in a workspace are filtered using a wild card and a data type, so only polygon feature classes that start with the letter *G* are in the resulting enumeration:

```
# Set the workspace. List all of the polygon feature classes that
# start with 'G'
GP.Workspace= "D:/St_Johns/data.mdb"
fcs = GP.ListFeatureClasses("G*","polygon")
```

Looping

Once you have the list you want, you must iterate through it utilizing one of the looping mechanisms of the scripting environment you are using. Most languages have a common looping structure, called a While loop, that works well with enumerations. Below is an example of a While loop used to iterate through the list generated in the previous example:

```
# Reset the enumeration to make sure the first object is returned
fcs.reset()
# Get the first feature class name
fc = fcs.next()
while fc: # While the feature class name is not None
    # Copy the features from the workspace to a folder
    GP.Copy(fc,"D:/St_Johns/Shapefiles/" + fc)
    # Get the next feature class name
    fc = fcs.next()
```

The geoprocessor uses enumerations for its list function results and cursor support, as enumerations support the flexibility required for sequential data access and multiple data types.

A While loop is ideal for working with an enumeration because it evaluates a condition before the loop is executed. By setting a value before the loop and setting the next value at the end of the loop, the loop will iterate until the value is set to a null or empty value. Enumerations have two methods and no properties. The Reset method ensures that the first record in the enumeration is returned when the next method is called. The Next method simply returns the currently selected value in the enumeration. Calling Next increments the enumeration's selection. Following is another example of how to use an enumeration created by a list function. The script is used to create raster pyramids for all rasters that are TIFF images within a folder.

```
# Set the workspace. List all of the TIFF files
GP.Workspace= "D:/St_Johns/images"
tiffs = GP.ListRaster("*","TIFF")
```

```
# Reset the enumeration to make sure the first object is returned
tiffs .reset()
# Get the first feature class name
tiff = tiffs.next()
while tiff: # While the raster name is not empty
   # Create pyramids
   GP.BuildPyramids(tiff)
   # Get the next TIFF raster
   tiff = tiffs.next()
```

Types

The default behavior for all list methods is to list all supported types. A keyword is used to restrict the returned list to a specific type. The type keywords for each method are listed in the table below. Refer to the ArcGIS Help system for more information about feature classes, datasets, workspaces, fields, tables, and supported raster formats.

Function	Type Keywords
ListDatasets	*All, Feature, Coverage, RasterCatalog, CAD, VPF, TIN, Topology*
ListFeatureClasses	*All, Point, Label, Node, Line, Arc, Route, Polygon, Region*
ListFields	*All, SmallInteger, Integer, Single, Double, String, Date, OID, Geometry, Blob*
ListWorkspaces	*All, Coverage, Access, SDE, Shapefile*
ListTables	*All, dBASE, INFO*
ListRasters	*All, ADRG, BIL, BIP, BSQ, BMP, CADRG, CIB, ERS, GIF, GIS, GRID, STACK, IMG, JPEG, LAN, SID, SDE, TIFF, RAW, PNG, NITF*

The ArcGIS overlay engine supports the intersection or union of any number of inputs, saving you from having to run these tools in a pair-wise fashion. Read the Overlay toolset documentation for more information about the new overlay engine in ArcGIS 9.

Tools may accept a single input or many inputs, depending on the operation. Tools that convert or overlay data may accept multiple datasets as input because of the nature of the operation. In a script, inputs are passed to these tools as a multivalue string, which uses a semicolon to separate each input within the string. A multivalue string is easy to create within a script using the string manipulation functions built into the scripting language. In this example, a multivalue string is created using one of Python's string concatenation functions:

```
# Import COM Dispatch module
from win32com.client import Dispatch

# Create the geoprocessor object
GP = Dispatch("esriGeoprocessing.GPDispatch.1")
# Set the workspace. List all of the feature classes in the dataset
GP.Workspace= "D:/St_Johns/data.mdb/neighborhoods"
fcs = GP.ListFeatureClasses()

# Create the multi-value string for the Analysis Union tool
fcs.reset
# Get the first feature class name and set the string variable
fc = fcs.next()
inputs = fc
# Get the next name and start the loop
fc = fcs.next()
while fc: # While the fc name is not empty
    inputs = inputs + ";" + fc
    fc= fcs.next()

# Union the feature classes with the land use feature class to create
# a single feature class with all of the neighborhood and land use data
inputs = inputs + ";D:/St_Johns/data.mdb/land_use"
GP.Toolbox = "Analysis"
GP.Union(inputs, "D:/St_Johns/data.mdb/lu_output")
```

Using input options

The Union and Intersect overlay tools in ArcGIS support the use of priority ranks, which are used to preserve features with high accuracy. A rank is assigned to each input feature class as an optional value, where 1 is the highest rank. If no rank is given for a feature class, then it uses the lowest rank available. If no ranks are provided for any input, then they all receive the same rank. In a multivalue string, a rank is specified after the name of the feature class, with a space separating the values. If the feature class name has a space, it must be wrapped by single quotes. The example below shows how to create such a string:

There is no harm in wrapping each item in a multivalue string with single quotes. Single quotes are only required when an item in the string has a space and there are input options, such as a rank or reclass value.

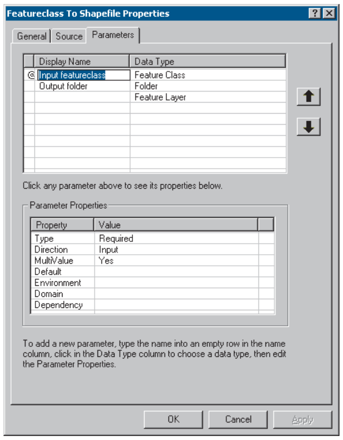
```
GP.Workspace= "D:/St_Johns/data.mdb/neighborhoods"
inputs = "east 1;west 1;south 1;'north end' 2"
inputs = inputs + ";D:/St_Johns/data.mdb/land_use 3"
GP.Toolbox = "Analysis"
GP.Union(inputs, "D:/St_Johns/data.mdb/land_use")
```

If an input has more than one option, separate each option with a space before using a semicolon to indicate the next input.

Refer to Chapter 3 for more information on tool return values and Chapter 5 for how to work with script arguments.

Using a multivalue input

Just as you may create and use a multivalue string for a tool's input, your script may have to use a multivalue string, as one may be returned as an output value of a tool or passed as an input argument for your script. Your script may simply pass the multivalue string as a parameter for a tool, or it may have to split all the values within it so they can be used individually.



Featureclass To Shapefile Properties

General | Source | Parameters

	Display Name	Data Type
@	Input featureclass	Feature Class
	Output folder	Folder
		Feature Layer

Click any parameter above to see its properties below.

Parameter Properties

Property	Value
Type	Required
Direction	Input
MultiValue	Yes
Default	
Environment	
Domain	
Dependency	

To add a new parameter, type the name into an empty row in the name column, click in the Data Type column to choose a data type, then edit the Parameter Properties.

OK | Cancel | Apply

*A script tool will generate a multi-value string for any parameter that has its MultiValue property set to **yes**.*

A script that converts multiple feature classes from one format to another is a good example of why you may want to use a multi-value as input. In ArcGIS, feature conversion can be performed by copying features from one workspace to another using the CopyFeatures tool. The tool will convert from one format to another if the two workspaces are of different types. The CopyFeatures tool copies one feature class at a time, so if your script accepts multiple inputs, it must split the string into its individual feature class names and put the names in a data structure that is suitable for looping, such as a list or array. The following example shows how this could be done:

```
# Import COM Dispatch and sys
from win32com.client import Dispatch
import sys

#Create the Geoprocessor object
GP = Dispatch("esriGeoprocessing.GpDispatch.1")

# Set the output workspace
GP.Workspace = sys.argv[2]

# Split input values using the semicolon
inputs = sys.argv[1]
inputlist = inputs.split(";")

#Loop through the list of inputs
for input in inputlist :
    # Validate the output name for the new workspace
    out_name = GP.ValidateTablename(input)
    # Copy the features to the new workspace
    GP.CopyFeatures(input,out_name)
```

Lists are what is known as a collection type in Python. Lists may contain any type of object, such as a string or number and they have a number of methods for appending, sorting and navigating.

Using value tables

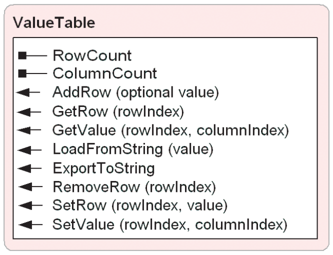
A value table is a flexible object that may be used as a value for any multivalue parameter. The previous examples of multivalue parameter values focus on the text value of the parameter, which may become difficult to use when there are numerous values with complex paths. The value table is used to organize the values into a table, so values may be easily added or removed, eliminating the complexity of parsing a multivalue text string.

A value table only exists during the lifetime of the geoprocessor object that created it. The table may be thought of as a virtual matrix of values that is not persisted as an actual table because it is a device for managing many parameter values in a script.

The number of columns a value table will contain must be specified when it is created. Each column corresponds to a value in the parameter being defined. Union, for example, requires the path to a dataset or a layer name, along with a priority rank for each input entry. A value table for Union will require two columns—one for the data and one for the priority rank. The following example shows how to create and populate a table for Union:

Value tables make it easy to work with multivalue parameters, as they do all the parsing of individual values. Each scripting language has different functions for parsing strings. The value table object provides a standard way to work with multivalue parameters, independent of the scripting language.

```
# Set the workspace. List all of the feature classes in the dataset
GP.Workspace= "D:/St_Johns/data.mdb/neighborhoods"
fcs = GP.ListFeatureClasses()

# Create the value table for the Analysis Union tool with 2 columns
vtab = GP.CreateObject("ValueTable",2)
fcs.reset
# Get the first feature class name and set the string variable
fc = fcs.next()
inputs = fc
# Get the next name and start the loop
fc = fcs.next()
while fc: # While the fc name is not empty
    # Update the value table with a rank of 2 for each record, except
    # roads
    if fc <> "roads":
        vtab.AddRow(fc,"2")
    else:
        vtab.AddRow(fc,"1")
    fc= fcs.next()

# Union the feature classes with the land use feature class to create
# a single feature class with all of the neighborhood and land use data
vtab.AddRow ("D:/St_Johns/data.mdb/land_use","2")
GP.Union_analysis(vtab, "D:/St_Johns/data.mdb/lu_output")
```



Value Table

RowCount	The number of rows in the table
ColumnCount	The number of columns in the table
AddRow	Add a new row to the table
GetRow(Index)	Return a specific row, with all column values
GetValue(RowIndex, ColumnIndex)	Return a value from a specific row and column
LoadValueFromString(String)	Set the table rows and columns using a multivalue string
ExportToString()	Create a multivalue string from all of the rows in the table
RemoveRow(RowIndex)	Remove a row from the table
SetRow(RowIndex, Value)	Set the value of a row using a string containing all column values
SetValue(RowIndex, ColumnIndex, Value)	Set the value of a specific column for a specific row

A value table may be populated with a multivalue string that has been passed to a script as an argument, making it easy to extract each record. The example below shows how to do this:

```
# Set the output workspace
GP.Workspace = sys.argv[2]

# Create the value table 1 column
vtab = GP.CreateObject("ValueTable",1)
```

```
# Set the values of the table with the contents of the first argument
vtab.LoadFromString(sys.argv[1])
x = 1

#Loop through the list of inputs
while x < vtab.RowCount:
    # Validate the output name for the new workspace
    name = vtab.GetRow(x)
    out_name = GP.ValidateTablename(name)
    # Copy the features to the new workspace
    GP.CopyFeatures(name ,out_name)
    x = x + 1
```

5 Messaging and script tools

Tools create messages while executing so the processing status may be known. Messages are also created once a tool has completed executing. There are different types of messages, depending on the processing status of the tool. This chapter will explain the different types of messages and how to use and set them within a script. Scripts may be used as the source of tools in ArcGIS, allowing scripts to be executed using dialog boxes, models, or the command line. The geoprocessor exposes a number of methods for easy communication between scripts and the geoprocessing framework in ArcGIS applications.

During execution of a tool, messages are relayed back to the geoprocessor. These messages include such information as when the operation started, what parameter values are being used, the operation's progress, and warnings of potential problems or errors. All communication between a tool, the geoprocessor, and the user is conducted via this messaging. Scripts are frequently run in an automated fashion without user interaction. Since geoprocessing tools do not assume there is direct interaction with a user, they never use message boxes or other types of dialog boxes during their execution.

Message types and severity

A tool message will be classified as either information, a warning, or an error. A message's type is indicated by its severity property, which is a numeric value. Messages may also be returned when dialog boxes are used to define tool parameters as tools execute a validation routine to check those values. A tool will validate its parameter values during its use.

An informative message may be used to indicate any event that does not reflect a problem or possible error. Typical informative messages indicate a tool's progress, what time a tool started or completed, output data characteristics, or tool results. A severity value of zero is used for informative messages, as they require no action from a user or script.

Warning messages are generated when a tool experiences a situation that may cause a problem during its execution or when the result may not be what the user expects. A user or script may choose to take action when a warning is returned, such as cancelling the tool's execution or making another parameter choice. Defining a coordinate system for a dataset that already has a coordinate system defined, for example, will generate a warning. The tool will still execute, but it may not create the desired result if the user did not intend to alter the existing coordinate system. A severity value of one is given to warning messages.

Error messages indicate a critical event that will prevent a tool from executing. Errors are generated when one or more parameters have invalid values or when a critical execution process or routine has failed. A path to data that does not exist, an invalid keyword, or corrupted data are examples of situations that will cause an error. Errors have a severity value of two, indicating an action is required by the script or user.

Many of the examples in this book have error handling routines. Refer to the scripting language's documentation for a complete discussion of how to use its error handling mechanism.

Most scripting languages have built-in error handling, allowing scripts to continue execution in a logical fashion when an error occurs. When an error message from a tool is returned to the geoprocessor, it generates a system error, which may be caught by a script's error handling routine. If a script does not have an error handling routine, it will fail immediately, which decreases its robustness. Use error handling routines to manage errors and improve a script's usability.

Messages from the last tool the geoprocessor executed are maintained as a list of message objects by the geoprocessor. A message object contains a message's text and its severity. Using a number of geoprocessor methods, you may access the message list to retrieve, add, or clear messages.

Getting messages

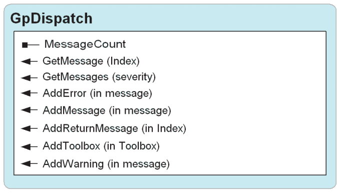
When tools are executed in a script, you may want to access the resulting messages if an error or warning occurs or simply show the progress of an operation. The GetMessages method will return a single string containing all the messages from the tool that was last executed. The returned messages may be filtered to only those with a certain severity using the optional severity option.

Arrays are collections of data with a fixed size. The data is of the same type and may be accessed in any order.

Arrays are typically zero based. The index to the first member is zero and the number of members minus one is the index to the last member.

A tool's second and last messages always give the start and end time for the tool's execution.

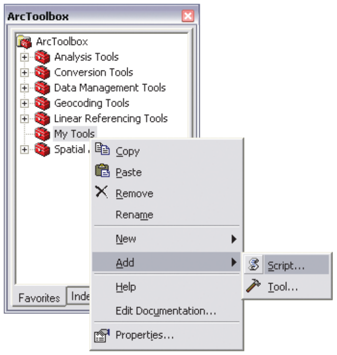
```
# Import COM Dispatch
from win32com.client import Dispatch

# Create the Geoprocessor object
GP = Dispatch("esriGeoprocessing.GpDispatch.1")
# Execute the Clip tool
GP.Toolbox = "Analysis"
GP.Workspace = "D:/St_Johns/data.mdb"
GP.Clip("roads","urban_area","urban_roads")
# Get the resulting messages and print them
print GP.GetMessages()
```



Individual messages may be retrieved using the GetMessage method. This method has one parameter, which is the index of the message in the geoprocessor's message list or array. The MessageCount property maintains the number of messages in the geoprocessor's message array. The example below shows how to print the start and end time for a tool.

```
GP.Clip("roads","urban_area","urban_roads")
# Print the second message.
print GP.GetMessage(1)
# Print the last message.
print GP.GetMessage(GP.MessageCount - 1)
```

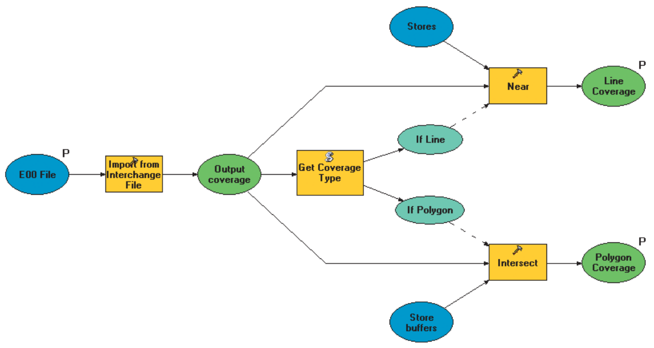
For information on how to add a script to a toolbox, refer to "Creating models and adding scripts" in Chapter 5 of Geoprocessing in ArcGIS.



For more information on using script tools in models, refer to "Controlling the flow of processing" in Chapter 9 of Geoprocessing in ArcGIS

Many scripts are written for specific datasets and scenarios and are never used after they have perfomed the task they were written for. One of the advantages of scripting is its low cost due to its rapid development. Scripts are often written for the task at hand and then deleted when the task is complete. Some scripts perform a generic operation that may be used repeatedly because they use arguments to change tool parameters and behavior. These scripts may rarely change and are used by a number of people.

The geoprocessing framework in ArcGIS accepts scripts as the executable source of a tool. They are a valuable way to quickly and easily customize geoprocessing capabilities. Scripts are also ideal for controlling models, as they can create variables that may be used to manage a model's flow. The ability of scripts to use the capabilities of the geoprocessor to inspect the properties of datasets along with the native statements of the scripting language, such as If, is powerful for model users. The example below shows a model using a script tool, GetType. The tool sets two output variables depending on the coverage's feature classes. The model uses these variables to control what executes after GetType. The code below shows what GetType does.



```
# Check the type of feature classes in a coverage
import win32com.client, sys
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

# Check the feature classes and set the output parameters
GP.Workspace(sys.argv[1]) # Set the workspace to the coverage
if GP.ListFeatureclasses("*","polygon").Next():
   GP.AddMessage("Feature type is polygon")
   GP.SetParameterAsText(1,"true")
   GP.SetParameterAsText(2,"false")
elif GP.ListFeatureclasses("*","arc").Next():
   GP.AddMessage("Feature type is line")
   GP.SetParameterAsText(1,"false")
   GP.SetParameterAsText(2,"true")
else:
   GP.AddMessage("Coverage has neither polygons nor lines")
   GP.SetParameterAsText(1,"false")
   GP.SetParameterAsText(2,"false")
```

Functions, modules, and objects may be imported from other Python scripts to simplify your script and centralize code. The import statement is used to pull everything from another script or just the entities you want.

Setting output messages

When a script tool is executed, messages often need to be returned to the user, especially when problems arise. To support the full integration of scripts as tools, the geoprocessor has several methods for adding messages, which are then available to the user. Messages added to the geoprocessor are immediately returned to the application or script executing the tool. The three methods for adding various types of output messages are:

AddMessage(message string)

AddWarning(message string)

AddError(message string)

Each of these methods take a single string parameter, which is then displayed to the user if the tool is run from a script tool in an ArcGIS application. Once an error message is added, the geoprocessor immediately generates a system error that halts the execution of the tool. The next example copies a list of feature classes from one workspace to another. An automatic conversion takes place if the workspaces are of a different type, such as a geodatabase to a folder. Error handling is used to catch any problems and return messages; otherwise, messages of success are returned during execution.

```
# ConvertFeatures.py
# Converts feature classes by copying them from one workspace to another

# Import the utility module which imports the standard library modules and
# creates the geoprocessing object.
from ScriptUtils import *

# Get the list of feature classes to be copied to the output workspace.
in_feature_classes = sys.argv[1]

# Establish an array of input featureclasses
in_feature_classes = SplitMulti(in_feature_classes)

# Get the output workspace
out_folder = sys.argv[2]

# Loop through the array copying each featureclass to the output workspace
# for each in_feature_class string in the in_feature_classes enumeration.
    try:
        # Create the output name
        feature_class_name = GP.ValidateTableName(in_feature_class,
        out_folder)

        # Add an output message
        GP.AddMessage("Converting: " + in_feature_class + " To " +
        feature_class_name + ".shp")

        # Copy the feature class to the output workspace
        CopyFeatures(in_feature_class, out_folder + os.sep +
```

Exceptions are events that may modify the flow of control through a program. They may be triggered or intercepted within a script using the Try and Raise statements.

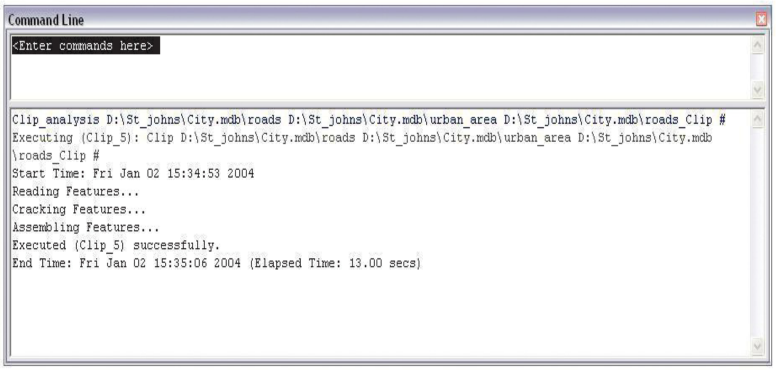
```
            feature_class_name)

            # If successful, add another message
            GP.AddMessage("Successfully converted: " + in_feature_class
            + " To " + out_folder)

        except StandardError, ErrDesc:
            GP.AddWarning("Failed to convert: " + in_feature_class)
            GP.AddWarning(ErrDesc)
        except:
            GP.AddWarning("Failed to convert: " + in_feature_class)
            if not gp.GetMessages(2) == "":
                GP.AddError(GP.GetMessages(2))

# ScriptUtils.py

# Import standard library modules
import win32com.client, sys, string, os

# Create the geoprocessing object
GP = win32com.client.Dispatch("esriGeoprocessing.GpDispatch.1")

def SplitMulti(multi_input):
    try:
        # Remove the single quotes and parenthesis around each input
        # featureclass

        #split input tables
        multi_as_list = string.split(multi_input, ";")
        return multi_as_list
    except:
        ErrDesc = "Error: Failed in parsing the inputs."
        raise StandardError, ErrDesc

def CopyFeatures(in_table, out_table):
    try:
        ErrDesc = "CopyFeatures failed"

        # Copy each feature class to the output workspace
        gp.copyfeatures_management(in_table, out_table)
    except:
        if gp.GetMessages(2) != "":
            ErrDesc = gp.GetMessages(2)
        raise StandardError, ErrDesc
```

StandardError is a built-in exception from which most exception types are based in Python. It is called if a tool error occurs, with the ErrDesc variable set by the tool's error message.

Strings, defined in the String module, are a built-in type used to store and represent text. A number of operations are supported for manipulating strings, such as concatenation, slicing and indexing.

The OS module provides a generic interface to the operating system's basic set of tools.

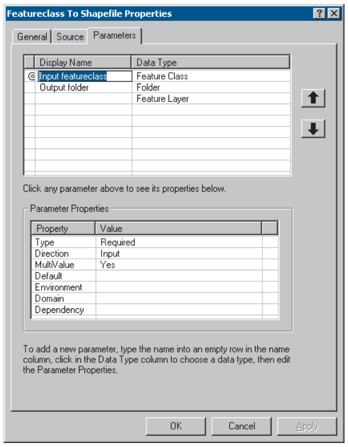
In ArcGIS applications, such as ArcMap or ArcCatalog, messages are displayed in the geoprocessing window whenever a tool is executed.



```
Clip_analysis D:\St_johns\City.mdb\roads D:\St_johns\City.mdb\urban_area D:\St_johns\City.mdb\roads_Clip #
Executing (Clip_5): Clip D:\St_johns\City.mdb\roads D:\St_johns\City.mdb\urban_area D:\St_johns\City.mdb
\roads_Clip #
Start Time: Fri Jan 02 15:34:53 2004
Reading Features...
Cracking Features...
Assembling Features...
Executed (Clip_5) successfully.
End Time: Fri Jan 02 15:35:06 2004 (Elapsed Time: 13.00 secs)
```

There are times when you may want a script to return messages from a tool it has executed. Using an index parameter, the AddReturnMessage method will return a message from the geoprocessor's message array. The example below shows how to return all a tool's messages:

```
GP.Clip_analysis("roads","urban_area","urban_roads")
# Return the resulting messages as script tool output messages
x = 0
while x < GP.MessageCount:
    GP.AddReturnMessage(x)
    x = x + 1
```



Display Name	Data Type
Input featureclass	Feature Class
Output folder	Folder
	Feature Layer

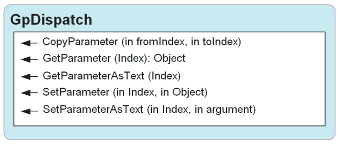
Click any parameter above to see its properties below.

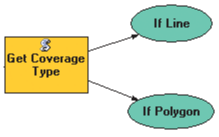
Parameter Properties

Property	Value
Type	Required
Direction	Input
MultiValue	Yes
Default	
Environment	
Domain	
Dependency	

To add a new parameter, type the name into an empty row in the name column, click in the Data Type column to choose a data type, then edit the Parameter Properties.

One of the property pages for a script tool. For information on adding a script to a toolbox, refer to "Creating models and adding scripts" in Chapter 5 of Geoprocessing in ArcGIS.


GpDispatch
CopyParameter (in fromIndex, in toIndex)
GetParameter (Index): Object
GetParameterAsText (Index)
SetParameter (in Index, in Object)
SetParameterAsText (in Index, in argument)



Script tools must populate the values of their output parameters so they can be used as parameters for other tools in a model.

Geoprocessing tools typically have parameters, as described in Chapter 3. Parameters, also called arguments, make scripts more generic and flexible, so they can be used to vary input data, set other tool parameters used in the script, or control the script's processing logic.

When a script is added to a toolbox, a number of parameters may be defined. These parameters correspond to a script's parameters, representing input and output values. Typically scripts only have input values, as they are typically run in an independent fashion. Script tools must define their outputs so tools work as expected in models built with ModelBuilder or on the command line in the geoprocessing window. Models need an output parameter so it can be used as the input for another tool, while tools on the command line allow users to specify the name and location of a tool's output.

System tools, built with ArcObjects, populate the properties of output variables in a model to aid subsequently connected tools. These properties are not required for building models but aid in their construction. System tools do this with a validation routine that is called each time a parameter value is changed for a tool. Script tools do not have a validation routine, just one for execution, which is the script itself. The geoprocessing framework does some validation for script tools automatically, such as ensuring a parameter value matches what is expected for a parameter, including a number for a numeric parameter and a path to a feature class for a feature class parameter. By defining dependencies between parameters, certain behavior may also be built into a tool. For example, a dependency between a feature class parameter and a field parameter in which the field parameter is dependent on the feature class, will mean the field list will automatically be populated on the tool when the feature class is given a valid value. Another script may not actually have a new output, as it alters a dataset specified as an input parameter. In this case, the script tool would still declare an output parameter, with the type set to derived with a dependency on the input dataset. The script tool will automatically set the output parameter's value to be the same as the input dataset when used in a model so the tool may be used in a work flow.

Getting input parameter values

Scripting languages typically provide a mechanism for accessing arguments passed to the script from the caller. VBScript, for example, provides a comma-delimited string of all input arguments, while Python uses its system module. A script must use these mechanisms if it is not the source of a script tool, as shown in the example below:

```
import win32com.client, sys
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

# Set the input workspace
GP.Workspace(sys.argv[1])
```

The argument list in Python is zero based, with the actual script call being the first argument. The second argument is the first user-specified value following the script name. Following is an example of how to call the script in the example above, specifying an input workspace:

```
Clipdata.py "d:\soils\Newfoundland"
```

If a script is the source of a script tool, it may use the geoprocessor to access the input parameter values. The script below, an edited section from an earlier sample, demonstrates how to use the GetParameterAsText method to get input parameter values:

```
# ConvertFeatures.py
# Converts feature classes by copying them from one workspace to another

# Import the utility module which imports the standard library modules and
# creates the geoprocessing object.
from ScriptUtils import *

# Get the list of feature classes to be copied to the output workspace.
in_feature_classes = GP.GetParameterAsText(0)

# Establish an array of input featureclasses
in_feature_classes = SplitMulti(in_feature_classes)

# Get the output workspace
out_folder = GP.GetParameterAsText(1)
```

Setting output parameters

Output parameter values may not be known until the script is executed, as seen in the script tool example at the beginning of this chapter. The script must evaluate or calculate an output value based on its input, so script tools must have a way to specify output values after execution so the values may be used in a model work flow.

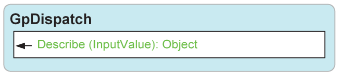
The SetParameter and SetParameterAsText methods will set the value of an output parameter using either an object, such as a value table, or a text string. Both methods require an index value to indicate which parameter will be updated. The GetType script example on page 50 sets output parameter values so the output variables of the GetType tool in ModelBuilder can be used to set precondition values for other processes in the model.

6

Data properties and access

Scripting logic is often determined by the properties of the data being used. Tool parameters may also vary bacause of these properties.

The Geoprocessor provides a number of functions for describing and manipulating data. They may be used to access general properties, such as a dataset's type, or specific values from attributes. This chapter will explain how to use the geoprocessor to focus on all aspects of your data.



Geoprocessing tools work with all types of data, such as geodatabase feature classes, shapefiles, rasters, tables, topologies and networks. Each piece of data has properties that may be used to control the flow of a script or the parameters of a tool. For example, the output feature type of an intersect operation is dependent on the type of data being intersected. When the Intersect tool is run within a script on a list of input datasets, it must be able to determine the data types used so the correct output type can be set.

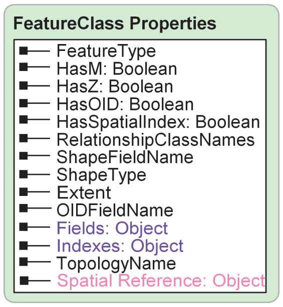
Describe creates what is known as a Dispatch object. Its properties are dynamic, depending on what data type is described. These properties come from the data elements defined by the geoprocessing COM framework. See the Data Elements Object Diagram found in the ArcGIS Developer Help system for more information.

Using the geoprocessor's Describe method, a dataset's properties may be determined then used to make decisions. The output of Describe is an object containing properties such as data type, fields, indices, and so on. Different dataset types have different properties, so the object created by Describe changes its properties depending on what is being described.

```
# Import COM Dispatch and create geoprocessor
from win32com.client import Dispatch
GP = Dispatch("esriGeoprocessing.GpDispatch.1")

# Describe feature class
desc = GP.Describe("D:/St_Johns/data.mdb/roads")
type = desc.FeatureType
```

The following charts show the properties available for each data type:

Any feature class will have the standard feature class properties shown on the right. This includes geodatabase, shapefile, coverage, CAD, Vector Product Format and Smart Data Compression feature classes.



All simple tables in the geodatabase require an ObjectID (OID) type field. It uniquely identifies each object stored in the table in the database. Other table types, such as INFO or dBASE, do not require an OID field.

Feature Class

FeatureType	simple, simple_junction, simple_edge, complex_junction, complex_edge, annotation, coverage_annotation, dimension
TopologyName	The name of the topology this feature class belongs to.
HasM	The M state: true for geometry is M enabled and false for not.
HasZ	The Z state: true for geometry is Z enabled and false for not.
HasSpatialIndex	The SpatialIndex state: true for has a spatial index and false for not.
RelationshipClassNames	A semicolon delimited list of the names of the associated relationship classes.
ShapeFieldName	The name of the shape field.
ShapeType	null, point, multipoint, line, circulararc, ellipticarc bezier3curve, path, polyline, ring, polygon envelope, any, bag, multiPatch, triangleStrip triangleFan, ray, sphere
Extent	XMin; YMin; XMax; YMax
SpatialReference	The spatial reference object.
HasOID	The OID state: true for has OID field and false for not.
OIDFieldName	The name of the object id field name.
Fields	The fields object for this table. This is the same as using the ListFields function.
Indexes	The indexes object for this table. This is the same as using the ListIndexes function.

Coverage FeatureClass Properties

- FeatureClassType
- HasFAT: Boolean
- Topology
- Extent
- Spatial Reference: Object
- HasOID: Boolean
- OIDFieldName
- Fields: Object
- Indexes: Object

Coverage feature classes always exist within a coverage dataset. Some of the coverage feature class properties stem from this relationship.

Coverage Feature Class

FeatureClassType	point, arc, polygon, node, tic, annotation, section, route link, region, label, file
HasFAT	true or false
Topology	notapplicable, preliminary, exists, unknown
Extent	XMin; YMin; XMax; YMax
SpatialReference	The spatial reference object.
HasOID	The OID state: true for has OID field and false for not.
OIDFieldName	The name of the object id field name.
Fields	The fields object for this table. This is the same as using the ListFields function.
Indexes	The indexes object for this table. This is the same as using the ListIndexes function.

Layer Properties

- FeatureClass
- FIDSet
- FieldInfo
- WhereClause
- NameString

Layers and table views are used to limit the features/rows from the underlying data source. This is done using a "where" clause to define an initial set of features, a selection, or both. The FieldInfo property is used to set what fields are available and what their names are.

Layer

Featureclass	The path of the feature class associated with the the layer
FIDSet	A semicolon delimited list of integer ids (record numbers)
FieldInfo	The field info object as: From_name To_name Hidden_flag Ratio; From_name; To_name Hidden_flag Ratio; …
WhereClause	The definition query where clause
NameString	The name of the layer

TableView Properties

- Table
- FIDSet
- FieldInfo
- WhereClause
- NameString

TableView

Table	The path of the table associated with the table view
FIDSet	A semicolon delimited list of integer ids (record numbers).
FieldInfo	The field info object as: From_name To_name Hidden_flag Ratio; From_name; To_name Hidden_flag Ratio; …
WhereClause	The definition query where clause
NameString	The name of the table view

Dataset Properties

- DatasetType
- Extent
- SpatialReference: Object

Datasets are containers that define the spatial reference and extents of their contents. Coverages, geodatabase feature datasets, and CAD datasets are all examples of this.

Dataset

DatasetType	any, container, coverage, geo, featuredataset, featureclass, planargraph, geometricNetwork, topology. text, table, relationshipclass, rasterdataset, rasterband, tin, caddrawing, rastercatalog, toolbox, tool
Extent	XMin; YMin; XMax; YMax
SpatialReference	The spatial reference object.

Relationship Class Properties

- Versioned: Boolean
- Fields: Object

Table Properties

- HasOID: Boolean
- OIDFieldName
- Fields: Object
- Indexes: Object

Workspace Properties

- Connection Properties
- ConnectionString
- Domains
- WorkspaceFactoryProgID
- WorkspaceType

Shapefile, personal geodatabase, and coverage workspaces have no connection properties. Domains are only found in Access and ArcSDE workspaces, as domains are only found in geodatabases.

For more information about enumerations and how to work with them, refer to Chapter 4.

Relationship Class

Versioned	The version state: true for versioned and false for not.
Fields	The fields object for this table. This is the same as using the ListFields function.

Table

HasOID	The OID state: true for has OID field and false for not.
OIDFieldName	The name of the object id field name.
Fields	The fields object for this table. This is the same as using the ListFields function.
Indexes	The indexes object for this table. This is the same as using the ListIndexes function.

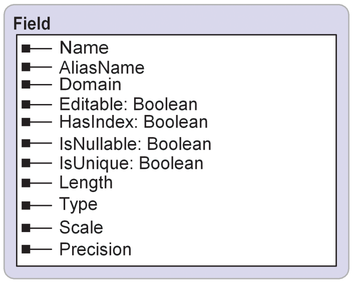
Workspace

ConnectionProperties	A semicolon delimited list of the pair of property name and the property value. Prop1: value1;
ConnectionString	The connection string
Domains	A semicolon delimited list of the names of the domains
WorkspaceFactoryProgID	The id as a string
WorkspaceType	filesystem, localdatabase, remotedatabase

Fields and indexes

The fields or indexes property of a table or feature class is exposed as an enumeration. In this case, the enumeration may contain field or index objects, consisting of properties for each field or index. The ListField and ListIndexes methods may be used to create the same enumerations or limit their contents. The following example shows how to create an enumeration of fields and how to loop through the contents to find a specific field.

```
# Import COM Dispatch and create geoprocessor
from win32com.client import Dispatch
GP = Dispatch("esriGeoprocessing.GpDispatch.1")

# Describe feature class
fc = "D:/St_Johns/data.mdb/roads"
desc = GP.Describe(fc)
fields = desc.Fields
field = fields.next()
while field:
    if field.Name == "Flag":
        # Set the value for the field and exit loop
        GP.CalculateField(fc, "Flag", "1")
        break
    field = fields.next()
```



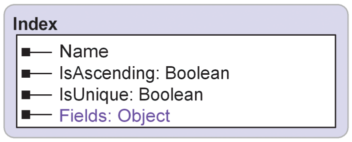
For more information on ListFields and ListIndexes, refer to Chapter 4. These functions may be used to limit the results based on name and type.

The properties of the field and index objects are listed below:

Field

Name	The name of the field
AliasName	The alias name of the field
Domain	The name of the associated domain
Editable	True if the field is editable
HasIndex	True if the field has an index
IsNullable	True if the field is nullable
IsUnique	True if the field is unique
Length	The field's length
Type	SmallInteger, Integer, Single, Double, String, Date, OID Geometry, Blob
Scale	The field's scale
Precision	The field's precision

Index



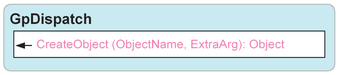
Name	The name of the index
IsAscending	True if the index is sorted in an ascending order
IsUnique	True if the index is unique
Fields	The fields object for this index

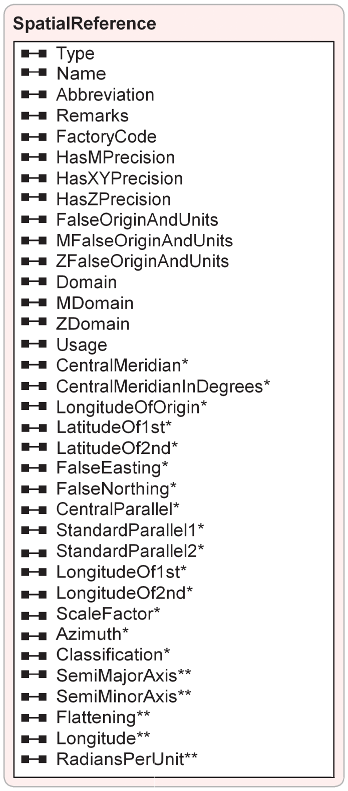
The spatial reference object

Geographic datasets, such as feature classes, coverages, and rasters, have a spatial reference, which defines a dataset's coordinate system, XY domain, M domain, and Z domain. Each part of the spatial reference has a number of properties, especially the coordinate system, which defines what map projection options are used to define horizontal coordinates. All this information is available from the spatial reference property, which is actually another object containing a number of properties.

```
# Describe feature class
fc = "D:/St_Johns/data.mdb/roads"
desc = GP.Describe(fc)
# Get the spatial reference
SR = desc.SpatialReference
# Check if the feature class is in projected space
if SR.Type == "Projected":
   GP.Copy(fc,"D:/St_Johns/data.mdb/UTM")
```

For more information about projected and geographic coordinate systems and ellipsoids, refer to Understanding Map Projections.




SpatialReference

- Type
- Name
- Abbreviation
- Remarks
- FactoryCode
- HasMPrecision
- HasXYPrecision
- HasZPrecision
- FalseOriginAndUnits
- MFalseOriginAndUnits
- ZFalseOriginAndUnits
- Domain
- MDomain
- ZDomain
- Usage
- CentralMeridian*
- CentralMeridianInDegrees*
- LongitudeOfOrigin*
- LatitudeOf1st*
- LatitudeOf2nd*
- FalseEasting*
- FalseNorthing*
- CentralParallel*
- StandardParallel1*
- StandardParallel2*
- LongitudeOf1st*
- LongitudeOf2nd*
- ScaleFactor*
- Azimuth*
- Classification*
- SemiMajorAxis**
- SemiMinorAxis**
- Flattening**
- Longitude**
- RadiansPerUnit**

* Projected Coordinate system only
** Geographic Coordinate system only

The properties of the spatial reference object are listed below:

Spatial Reference

Type	Unknown, Projected, Geographic
Name	The name of the projection
Abbreviation	Abbreviated projection name
Remarks	Any user remarks made when the spatial reference was created
FactoryCode	The factory code of the spatial reference
HasXYPrecision	True if the underlying datasource stores its XY coordinates as integers
HasMPrecision	True if the underlying datasource stores its M values as integers
HasZPrecision	True if the underlying datasource stores its Z values as integers
FalseOriginAndUnits	falseX;falseY;xyUnits
MFalseOriginAndUnits	falseM;MUnits
ZFalseOriginAndUnits	falseZ;ZUnits
Domain	XMin;YMin;XMax;YMax
MDomain	MMin;MMax
ZDomain	ZMin;ZMax
Usage	The usage notes of a projected coordinate system.
CentralMeridian*	Defines the origin of the x-coordinates. Decimal Degrees
CentralMeridianInDegrees*	Defines the origin of the x-coordinates. Degrees, Minutes, Seconds
LongitudeOfOrigin*	Defines the origin of the y-coordinates. Decimal Degrees
LatitudeOf1st*	The latitude of the first point of a projected coordinate system.
LatitudeOf2nd*	The latitude of the second point of a projected coordinate system.
FalseEasting*	A linear value applied to the origin of x-coordinates.
FalseNorthing*	A linear value applied to the origin of y-coordinates.
CentralParallel*	Defines the origin of the y-coordinates. Same as the LongitudeOfOrigin
StandardParallel1*	Used with conic projections to define the latitude lines where the scale is 1.0.
StandardParallel2*	Used with conic projections to define the latitude lines where the scale is 1.0.
LongitudeOf1st*	The longitude of the first point of a projected coordinate system.
LongitudeOf2nd*	The longitude of the second point of a projected coordinate system.

ScaleFactor*	A unitless value applied to the center point or line of a map projection
Azimuth*	Defines the center line of a projection.
Classification*	The classification of a map projection.
SemiMajorAxis**	The longer radius of an ellipsoid.
SemiMinorAxis**	The shorter radius of an ellipsoid.
Flattening**	The difference in length between the SemiMajor and SemiMinor axes expressed as a fraction or a decimal.
Longitude**	The least (left) longitude bounding a 360 degree range.
RadiansPerUnit**	The radians per angular unit.

* Projected coordinate system only
** Geographic coordinate system only

Property sets

Some properties are composed of a set of values. The tolerances of a coverage or the connection properties of a workspace are examples of this. Property sets have named properties that can be called from the property set itself. In the example below, the tolerances of a coverage are printed to the standard output:

```
from win32com.client import Dispatch
GP = Dispatch("esriGeoprocessing.GpDispatch.1")

desc = GP.Describe("D:/St_Johns/freshwater")
covTols = desc.tolerances
print covTols.Fuzzy
print covTols.Dangle
print covTols.TicMatch
print covTols.Edit
print covTols.NodeSnap
print covTols.Weed
print covTols.Grain
print covTols.Snap
```

Property sets are typically used when the properties of the object being described may vary. The connection properties of an enterprise geodatabase workspace will vary depending on the type of ArcSDE database that is being used, so it is well suited to a property set that has no predefined set of values. Refer to ArcObjects documentation for more information about workspace properties.

GpDispatch

← Exists (InputValue): Boolean

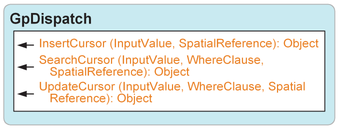
Checking for existence

Scripts often use paths to data, which may be problematic if the data being referenced does not exist. Data may be deleted or moved between executions of a script, which will cause errors if the path is used as a geoprocessing tool parameter. If there is a possibility of a referenced dataset not existing during a script's execution, the geoprocessor's Exists method should be used. The function simply returns a Boolean value for the existence of a dataset or object at the time of

The default behavior for all tools is to overwrite any output that already exists. This behavior may be changed by changing the overwrite data setting to false. An error is returned when the overwrite data setting is false and a tool's specified output already exists.

execution. Objects, such as a cursor, spatial reference, or any other object managed by the geoprocessor, may also be used as input to Exists. Exists will work with any type of data available in ArcCatalog or with any system file or folder. An ArcCatalog path must be used for this and any other method of the geoprocessor when referring to GIS data. If the data resides in an enterprise geodatabase, the name must be fully qualified. See Chapter 7 for more information on working with geodatabases and qualifying names.

```
GP.Workspace = "D:/St_Johns/data.mdb"
# Clip roads feature class if it exists
fc = "D:/St_Johns/data.mdb/roads"
if GP.Exists(fc):
  GP.clip_analysis(fc,"urban_area","urban_roads")
```


GpDispatch

InsertCursor (InputValue, SpatialReference): Object
SearchCursor (InputValue, WhereClause, SpatialReference): Object
UpdateCursor (InputValue, WhereClause, SpatialReference): Object

A cursor is a data access object that can either be used to iterate over the set of rows in a table or insert new rows into a table. Cursors have three forms, referred to as a search, insert, or update cursor. Each type of cursor is created by a corresponding geoprocessor method (SearchCursor, InsertCursor, or UpdateCursor) on a table, table view, feature class, or feature layer. A search cursor can be used to retrieve rows. An update cursor can be used to positionally update and delete rows, while an insert cursor is used to insert rows into a table or feature class.

All three cursor methods create an enumeration of row objects. The methods supported by the row object depend on the type of cursor created. The Next method on a search or update cursor returns the next row in the enumeration. To retrieve all rows in a table containing N rows, the script must make N calls to Next. In Python, a call to Next after the last row in the result set has been retrieved returns None, which is a special data type that acts as a placeholder.

Cursors can only be navigated in a forward direction; they do not support backing up and retrieving rows that have already been retrieved or making multiple passes over data. If a script needs to make multiple passes over the data, the application needs to reexecute the method that returned the cursor. If both executions of a method are made within the same edit session (or database transaction with the appropriate level of isolation), the application is guaranteed not to see any changes made to the data by other concurrently executing applications. This example shows a simple cursor operation. It prints out the value of the each field for the first row in a table.

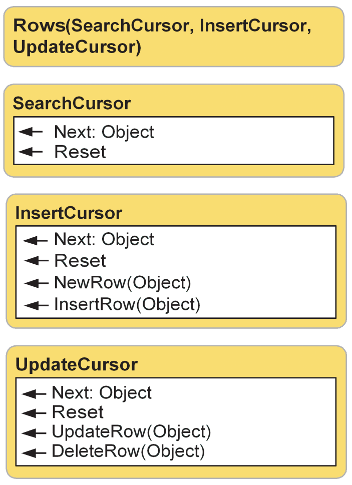
```
# Create search cursor
rows = GP.SearchCursor("D:/st_johns/roads.shp")
row = rows.Next()
fields = GP.ListFields("D:/st_johns/roads.shp")
field = fields.Next()
while field:
    if field.type != "Geometry":
        print field.name + ": Value = " + str(row.GetValue(field.name))
    field = fields.Next()
```

Fields are accessed using the row object. Values are returned using either the field name as a property of the row object or by its position in the table.

Note that no data is fetched from the table until the Next method is called.

When you are using a cursor and changing the underlying data at the same time, you may be concerned about the cursor operation and positioning. The situation, summarized by the table below, is actually quite simple.

Cursor type	Method	Effect on position
Search	Next	Advances position by one
Insert	NewRow followed by InsertRow	No effect (the new row does not belong to the cursor)
	Change values, followed by InsertRow	No effect
	InsertRow	Not applicable—insert cursors do not have position, you may not use *NextRow*
Update	Next	Advances position by one
	UpdateRow	No effect
	DeleteRow	Moves position back by one



All updates or inserts to a table are done outside an edit session in ArcGIS. Changes made are permanent and cannot be undone.

All row objects retrieved from a table logically contain the same ordered set of fields. In particular, the order of fields in a row of a table is the same as the order of fields returned from the ListFields method. The row will only contain the visible fields of the table used to create the cursor, with each field name being a property of the object.

Row enumeration object

The methods of the enumeration object created by the various cursor methods vary depending on the type of cursor created. The following chart shows what methods are supported for each cursor type.

Cursor type	Supported Methods	Description
Search	Next	Retrieves the next row object
Insert	NewRow	Creates an empty row object
	InsertRow	Will insert a row object into the table
	Next	Retrieves the next row object
Update	UpdateRow	Will update the current row with a modified row object
	DeleteRow	Removes the row from the table
	Next	Retrieves the next row object

UpdateRow

The UpdateRow method can be used to update the row at the current position of an update cursor. Making a call to Next on a cursor returns a row and positions the cursor on that row. After fetching a row object using Next, the script can modify the row as needed and call UpdateRow, passing in the modified row.

```
# Create update cursor for feature class
rows = GP.UpdateCursor("D:/St_Johns/data.mdb/roads")
row = rows.Next()
# Update the field used in buffer so the distance is based on the road
# type. Road type is either 1, 2, 3 or 4. Distance is in meters.
while row:
    row.buffer_distance = row.road_type * 100
    rows.UpdateRow(row)
    row = rows.Next()
```

DeleteRow

The DeleteRow method can be used to delete the row at the current position of an update cursor (that is, to delete the row returned by the last call to Next on this cursor). After fetching the row object using Next, the script should call DeleteRow on the cursor to delete the row.

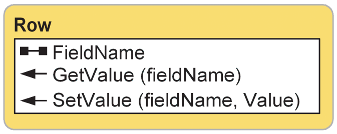
```
# Create update cursor for feature class
rows = GP.UpdateCursor("D:/St_Johns/data.mdb/roads")
row = rows.Next()
while row: # Delete all rows that have a roads type of 4
   if row.road_type == 4:
      rows.DeleteRow(row)
  row = rows.Next()
```

Cursors honor layer/table view definition queries and selections. The row enumeration object will only contain the rows that would be used by any geoprocessing tool during an operation.

InsertRow

Insert cursors are used to bulk insert rows. The InsertRow method takes a row object as an argument. The script obtains a new row object using the NewRow method on the enumeration object into which rows are to be inserted. Each call to InsertRow on the cursor creates a new row in the table whose initial values are set to the values in the input row.

```
# Create insert cursor for table
rows = GP.InsertCursor("D:/St_Johns/data.mdb/roads_lut")
x = 1
# Create 25 new rows. Set the initial row id and distance values
while x <= 25:
    row = rows.NewRow()
    row.rowid = x
    row.distance = 100
    rows.InsertRow(row)
    x = x + 1
```



Not all scripting languages support the evaluation of a variable as an object property or method. In these cases the GetValue and SetValue methods are particularly useful.

Python supports the use of a variable as a property or method name using the Eval function.

GetValue and SetValue

The row object may access and update field values using field names or a field's index position. This approach works well when the field names or order is known, but this may not always be the case. Variables may be used to create generic scripts with no knowledge of what fields exist within a table. The row has two methods that allow the use of variables as field names. The GetValue method simply returns a field's value, with the field name being the only input parameter. The SetValue method has two parameters—the field to be updated and the value to be used. The sample below creates a lookup table for all polygon feature classes in a workspace. The lookup table contains an ID that corresponds to the ObjectID for each feature in the feature class and a distance value that could be used for the Buffer tool.

```
# List the polygon feature classes
fcs = GP.ListFeatureclasses("*","polygon")
# Loop through the results. Get the ObjectID field
fc = fcs.Next
while fc:
   OIDFields = GP.ListFields(fc,"*","OID")
   # Create a search cursor on the feature class and an insert cursor
   # on a new look-up-table
   featcur = GP.SearchCursor(fc)
   # Create look-up-table and add fields
   tab = fc + "_lut"
   GP.CreateTable(tab)
   GP.AddField(tab,"id","long")
   GP.AddField(tab,"dist","long")
   # Open insert cursor on new look-up-table
   tabcur = GP.InsertCursor(tab)
   f_row = featcur.Next()
   # Get the OID field from the feature class
   OIDField = OIDFields.Next()
```

```
# Loop through the rows in the feature class
while f_row:
    # Create a new row for the look-up-table and set its id to be the
    # same as the feature OID and set the distance to 100.
    t_row = tabcur.NewRow()
    t_row.id = f_row.GetValue(OIDField.name)
    t_row.dist = 100
    # Insert the row into the look-up-table and get the next row
    # from the feature class
    tabcur.InsertRow(row)
    f_row = featcur.Next()
fc = fcs.Next
```

Specifying a query

When a query is specified for an update or search cursor, only the records satisfying that query are returned. A SQL query represents a subset of the single table queries that may be made against a table in a SQL database using the SQL SELECT statement. The syntax used to specify the where clause is the same as that of the underlying database holding the data. Refer to the ArcGIS Desktop Help system for more information on defining SQL where clauses. The example below filters the rows of a search cursor to only roads of a certain type:

Structured Query Language (SQL) is a powerful language you use to define one or more criteria that can consist of attributes, operators, and calculations. For example, imagine you have a table of customer data and want to find those who spent more than $50,000 with you last year and whose business type is "Restaurant". You would select the customers with this expression: "Sales > 50000 AND Business_type = 'Restaurant'".

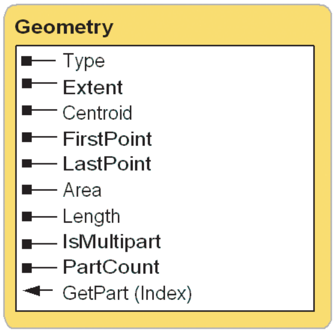
```
# Import COM Dispatch and create geoprocessor
from win32com.client import Dispatch
GP = Dispatch("esriGeoprocessing.GpDispatch.1")
# Create search cursor
rows = GP.SearchCursor("D:/St_Johns/data.mdb/roads",
                        "[type] = 'residential'")
row = rows.Next()
while row:
    # Print the name of the residential road
    print row.Name
    row = rows.Next()
```

The geometry object

Using a geometry object, the geoprocessor supports cursor access of feature geometry. The object, created by the row object when the shape field is specified, exposes a number of properties that describe a feature. The example below shows how to create a geometry object for each line feature in a feature class and sum their length:

All simple feature classes require a geometry type field. It contains the actual geometry of a feature and is typically called Shape. The ListFields or Describe methods may be used to retrieve the geometry field from a feature class. The name of the field can be determined from the field object.

```
# Create search cursor
rows = GP.SearchCursor("D:/St_Johns/data.mdb/roads")
row = rows.Next()
# Calculate the total length of all roads
length = 0
while row:
    # Create the geometry object
    feat = row.shape
    length = length + feat.Length
    row = rows.Next()
print length
```

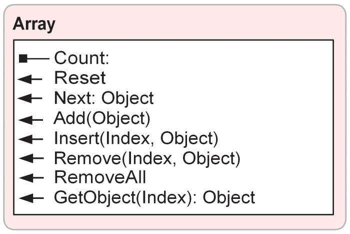


Geometry

Type	null, point, multipoint, line, circulararc, ellipticarc bezier3curve, path, polyline, ring, polygon envelope, any, bag, multiPatch, triangleStrip triangleFan, ray, sphere
Extent	XMin; YMin; XMax; YMax
Centroid	X; Y This is the centroid for polygons, the point closest to the center of the extent of the multipoint, and the midpoint of the line
FirstPoint	The first coordinate of the feature
LastPoint	The last coordinate of the feature
Area	The area of a polygon. Empty for all other feature types
Length	The length of the linear feature. Empty for point, multipoint or polygon feature types
IsMultipart	True if the number of parts for this geometry is more than one.
PartCount	The number of geometry parts for the feature
GetPart(optional index)	Returns an array of point objects for a particular part of geometry or an array containing a number of arrays, one for each part

Reading geometries

Each feature in a feature class contains a set of points defining the vertices of a polygon or line or a single coordinate defining a point feature. These points may be accessed using the geometry object, which returns them in an array of point objects. The array object may contain any number of geoprocessing objects, such as points, geometries, or spatial references.



Array

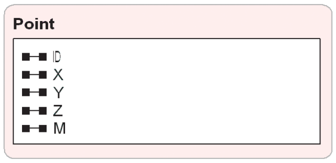
Reset()	Resets the array to the first object
Next()	Returns the next object in the array
Count	The number of objects in the array
Add(Object)	Adds an object to the array in the last position
Insert(Index, Object)	Adds an object to the array in a specific position
Remove(Index)	Removes a specific object from the array
RemoveAll()	Removes all objects and creates an empty array
GetObject(Index)	Returns a specific object from the array

Features in a geodatabase or shapefile may have multiple parts. The geometry object's PartCount property returns the number of parts for a feature. The GetPart method will return an array of point objects for a particular part of the geometry if an index is specified. If an index is not specified, an array containing an array of point objects for each geometry part will be returned. The example below will print the coordinates for all features to the output window:

```
# Create search cursor
rows = GP.SearchCursor("D:/St_Johns/data.mdb/roads")
row = rows.Next()
# Print the coordinates of each road line feature
while row:
   # Create the geometry object
   feat = row.shape
 a = 0
   while a < feat.PartCount:
      # Get each part of the geometry
      roadArray = feat.GetPart(a)
      roadArray.Reset
      # Get the first point object for the feature
      pnt = roadArray.Next()
       while pnt:
          print str(pnt.id) + ";" + str(pnt.x) + ";" + str(pnt.y)
          pnt = roadArray.Next()
        a = a + 1
   row = rows.Next()
```

A multipart feature is composed of more than one physical part but only references one set of attributes in the database. For example, in a layer of states, the State of Hawaii could be considered a multipart feature. Although composed of many islands, it would be recorded in the database as one feature.

Point features return a single point object instead of an array of point objects. All other feature types—polygon, polyline, and multipoint—return an array of point objects or an array containing multiple arrays of point objects if the feature has multiple parts.



Point

X	The horizontal coordinate of the point
Y	The vertical coordinate of the point
Z	The elevation value of the point
M	The measure value of the point
ID	A long value used to unqiuely identify the point

A ring is a closed path that defines a two-dimensional area. A valid ring consists of a valid path such that the from and to points of the ring have the same X and Y coordinates. A clockwise ring is an exterior ring, and a counterclockwise ring defines an interior ring.

A polygon will consist of a number of rings if it contains holes. The array of point objects returned for a polygon will contain the points for the exterior ring and all inner rings. The exterior ring is always returned first, followed by inner rings, with null point objects as the separator. Whenever a script is reading coordinates for polygons in a geodatabase or shapefile, it should contain logic for handling inner rings if this information is required by the script; otherwise, only the exterior ring will be read. The following script prints out the coordinates for polygons in a feature class. It shows how to handle multipart polygons and polygons with multiple rings.

```
from win32com.client import constants, Dispatch
from types import *
import pythoncom, sys
```

```
GP = Dispatch("esriGeoprocessing.GpDispatch.1")
fc = sys.argv[1]

dsc = GP.describe(fc)
print "Describing:", fc
ftype = dsc.ShapeType
rows = GP.searchcursor(fc)

print "Extent:"
print dsc.Extent

# Create search cursor
rows = GP.SearchCursor(fc)
row = rows.Next()
# Print the coordinates of each landuse polygon feature
while row:
   # Create the geometry object
   feat = row.shape
    a = 0
   print " "
   print "Feature: " + str(row.fid) + " number of parts: " +
   str(feat.PartCount)
   while a < feat.PartCount:
      # Get each part of the geometry
      print "Part: " + str(a + 1)
      LUArray = feat.GetPart(a)
      LUArray.Reset()
       b = 1
      # Get the first point object for the polygon
      pnt = LUArray.Next()
      while pnt:
         print str(pnt.id) + "," + str(pnt.x) + "," + str(pnt.y)
         pnt = LUArray.Next()
         # The point may be a null separator, so check to
         # see if another point exists.
          if not pnt:
             pnt = LUArray.Next()
            # If the point does exist, continue printing
            # the coordinates of the inner ring.
             if pnt:
                print "Inner ring: " + str(b)
                 b = b + 1
      print " "
      a = a + 1
   row = rows.Next()
```

Strings may be easily concatenated in Python using the addition operator. The Str function can be used to return the string value of any object so the value can be concatenated with other strings.

Writing geometries

Using insert and update cursors, scripts may create new features in a feature class or update existing ones. A script can define a feature by creating a point object, populating its properties and placing it in an array. That array may then be used to set a feature's geometry. A single geometry part is defined by an array of points, so

Read the upcoming section about locking to understand how cursors affect other applications.

a multipart feature can be created from multiple arrays of points. The following example shows how to read a text file containing a series of linear coordinates and then use them to create a new feature class.

```
# Create a new line feature class using a text file of coordinates.
# The coordinate file is in the format of ID;X;Y.
import win32com.client, sys, fileinput, os, string
# Create the geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")
# Get the name of the input file
infile = sys.argv[1]
# Get the name of the output feature class
fcname = sys.argv[2]
# Get the name of the template feature class.
template = sys.argv[3]
try:
   # Create the feature class
   GP.CreateFeatureclass(os.path.dirname(fcname),os.path.basename(fcname),
                            "Polyline", template)
   # Open an insert cursor for the new feature class.
   cur = GP.InsertCursor(fcname)
   # Create the array and point objects needed to create a feature
   lineArray = GP.CreateObject("Array")
   pnt = GP.CreateObject("Point")
   ID = -1 # Initialize a variable for keeping track of a feature's ID.
   # Assume all IDs are positive.
   for line in fileinput.input(infile): # Open the input file
      # Create a list of input values and set the point properties.
      values = string.split(line,";")
      pnt.id = values[0]
      print pnt.id
      pnt.x = values[1]
      print pnt.x
      pnt.y = values[2]
      print pnt.y
      if ID == -1:
         ID = pnt.id
      # Add the point to the feature's array of points.
      # If the ID has changed create a new feature
      if ID != pnt.id:
         # Create a new row, or feature, in the feature class.
         feat = cur.NewRow()
         # Set the geometry of the new feature to the array of points
         feat.shape = lineArray
         # Insert the feature
         cur.InsertRow(feat)
         lineArray.RemoveAll()
      lineArray.add(pnt)
      ID = pnt.id
except:
    print GP.GetMessages(2)
```

Below is an example of a file that may be processed by this script. The file has a blank line at the end to ensure all inputs are used:

```
1;-61845879.0968;45047635.4861

1;-3976119.96791;46073695.0451

1;1154177.8272;-25134838.3511

1;-62051091.0086;-26160897.9101

2;17365918.8598;44431999.7507

2;39939229.1582;45252847.3979

2;41170500.6291;27194199.1591

2;17981554.5952;27809834.8945

3;17365918.8598;44431999.7507

3;15519011.6535;11598093.8619

3;52046731.9547;13034577.2446

3;52867579.6019;-16105514.2317

3;17160706.948;-16515938.0553
```

An array of points is not necessary when writing point features. A single point object is used to set the geometry of a point feature.

The geoprocessor validates all geometries before they are written to a feature class. Issues such as incorrect ring orientation and self-intersecting polygons, among others, are corrected when the geometry is simplified before its insertion. The geoprocessor will not write invalid geometry.

The spatial reference for a feature class describes its coordinate system (for example, geographic, UTM, and State Plane), its spatial domain, and its precision. The spatial domain is best described as the allowable coordinate range for x,y coordinates, m- (measure) values, and z-values. The precision describes the number of system units per one unit of measure.

Setting a cursor's spatial reference

By default, the spatial reference of the geometry returned from a search cursor or set by an update or insert cursor is the same as the feature class referenced by the cursor. A different spatial reference for the input or output features may be specified when the cursor is created. In the case of a search cursor, specifying a spatial reference that is different from the spatial reference of the input feature class will result in geometries that are projected to the cursor's spatial reference. The example below has a point feature class with a coordinate system of Universal Transverse Mercator (UTM) zone 21 North, defined in its spatial reference. The script will produce a text file with the coordinates of the points in decimal degrees.

```
import win32com.client, sys
# Create the geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

# Describe a feature class with a geographic coordinate system
desc = GP.Describe("D:/St_Johns/data.mdb/latlongbnd")

# Create search cursor. Use the spatial reference object from the
# described feature class so geometries are returned in decimal degrees.
rows = GP.SearchCursor("D:/St_Johns/data.mdb/buildings", "",
desc.SpatialReference)
row = rows.Next()

# Open the file for output. This also creates the file if it does not
exist.
out = open(sys.argv[1],"w")

# Print the coordinates of each building point feature
while row:
   # Create the geometry object
   feat = row.shape
   # Get the geometry's point object.
   pnt = feat.GetPart()
   # Write the XY coordinate to the output file
   out.write(str(pnt.x) + ";" + str(pnt.y) + "\n")
   row = rows.Next()

# Close the output file
out.close()
```

Setting the spatial reference of an insert or update cursor is required when the coordinate system of the input geometries is different from the referenced feature class. Defining an insert or update cursor's spatial reference allows the cursor to project the coordinates on the fly before they are actually written to the feature class. A script that writes geographic Global Positioning System (GPS) coordinates to a feature class with a State Plane Coordinate System is an ideal example of when to set a cursor's spatial reference.

Locking

An edit session in ArcMap will apply a shared lock to data during the edit session. An exclusive lock is applied when edits are saved. A dataset is not editable if an exclusive lock already exists.

Insert and update cursors honor table locks set by ArcGIS. Locks prevent multiple processes from changing the same table at the same time. There are two types of locks—shared and exclusive. A shared lock is applied anytime a table or dataset is accessed. Multiple shared locks can exist for a table, but no exclusive locks are permitted if a shared lock exists. Displaying a feature class in ArcMap or previewing a table in ArcCatalog are examples of when a shared lock would be applied. Exclusive locks are applied when changes are made to a table or feature class. Editing and saving a feature class in ArcMap, changing a table's schema in ArcCatalog or using an insert cursor on a shapefile in PythonWin are examples of when an exclusive lock is applied by ArcGIS.

Update and insert cursors cannot be created for a table or feature class if an exclusive lock exists for that dataset. The UpdateCursor or InsertCursor methods will return an error stating that the methods failed because an exclusive lock exists for the dataset. If these methods successfully create a cursor, they will apply an exclusive lock on the dataset, so two scripts may not create an update or insert cursor on the same dataset.

Locks persist until the application or script releases a dataset, either by closing or releasing the cursor object explicitly. In a script, the cursor object should be deleted so the exclusive lock it placed on the dataset is released. Otherwise, all other applications or scripts could be unnecessarily prevented from accessing a dataset. The sample below shows how to open an update cursor and release it. An error handler is used to check if the UpdateCursor method fails because of another exclusive lock on the table.

When working in a Python editor, such as PythonWin, you may need to clean up object references to remove dataset locks set by cursors. Use the gc (garbage collection) module in the Interactive window to control when unused objects are removed and/or explicitly delete references within your script.

Other scripting languages have different functions or statements for undoing referencing to objects. For example, VBScript uses the Set statement to release an object by setting it to Nothing. For more information regarding object handling, refer to the reference of the language you are using.

```
# Create update cursor for feature class
try:
    rows = GP.UpdateCursor("D:/St_Johns/data.mdb/roads")
    row = rows.Next()
    # Update the field used in buffer so the distance is based on the road
    # type. Road type is either 1, 2, 3 or 4. Distance is in meters.
    while row:
        row.buffer_distance = row.road_type * 100
        rows.UpdateRow(row)
        row = rows.Next()
# Delete the row and cursor
    del row, rows
except:
    if not GP.GetMessages() == "":
        GP.AddMessage(GP.GetMessages(2))
    if row:
        del row
    if rows:
        del rows
```

7

Working with geodatabases

A geodatabase provides a rich single or multiuser environment for managing your data. Updating geodatabases with the results of geoprocessing tools requires some forethought, as the geodatabase has some rules for managing how datasets and fields are named. The geoprocessor provides a number of methods to ensure that correct naming standards are followed.

Personal geodatabases are designed for single use, utilizing Access as the database. Multiuser geodatabases use RDBMS packages, such as Oracle, SQL Server, or IBM DB2. Generic geoprocessing scripts should be written to work with either type of geodatabase using the various validation and qualification geoprocessor methods.

Geodatabases are relational databases that contain geographic information. Geodatabases contain feature classes and tables. Feature classes can be organized into a feature dataset; they can also exist independently in the geodatabase.

Feature classes store geographic features represented as points, lines, or polygons and their attributes; they can also store annotation and dimensions. All feature classes in a feature dataset share the same coordinate system. Tables may contain additional attributes for a feature class or geographic information such as addresses or x,y,z coordinates.

The geodatabase model defines a generic model for geographic information. This generic model can be used to define and work with a wide variety of different user- or application-specific models. By defining and implementing a wide variety of behavior on a generic geographic model, a robust platform is provided for the definition of a variety of user data models.

The geodatabase supports a model of topologically integrated feature classes, similar to the coverage model. However, it extends the coverage model with support for complex networks, topologies, relationships among feature classes, and other object-oriented features. The ESRI ArcGIS Desktop applications (ArcMap, ArcCatalog, and ArcGlobe™) work with geodatabases as well as coverages and shapefiles. To learn how to build and edit data in a geodatabase, see *Editing in ArcGIS*.

Successfully implementing a multiuser GIS system with ArcInfo and ArcSDE starts with a good data model design and database tuning. How the data is stored in the database, the applications that access it, and the client and server hardware configurations are all key factors to a successful multiuser GIS system.

A critical part of a well-performing geodatabase is the tuning of the database management system (DBMS) in which it is stored. This tuning is not required for personal geodatabases; however, it is critical for ArcSDE geodatabases. For more information on tuning your database for ArcSDE and the geodatabase, see the Configuration and Tuning Guide for <DBMS> PDF file.

Designing a geodatabase is a critical process that requires planning and revision until you reach a design that meets your requirements. Once you have a design, you can create the geodatabase and its schema using geoprocessing tools. There are tools for creating, modifying, and analyzing your geodatabase schema, such as Create Featureclass, Compress, and Add Subtype.

Geodatabases use various relational database management systems to maintain the many tables that comprise a geodatabase. All tables in a geodatabase must have a unique name, so a mechanism for checking if a table name is unique is essential when creating data in a geodatabase. The default behavior for geoprocessing tools is to overwrite output that already exists, so there is potential for accidentally overwriting data if the script does not ensure that the new output name is unique.

For more information regarding reserved words and invalid characters, refer to the documentation of the RDBMS you are using. An example of a reserved word is "Select", while "%" is a restricted character in most databases.

Using the ValidateTableName method, a script can determine if a specific name is valid and unique to a specific workspace. Specifying the workspace as a parameter allows the geoprocessor to check all the existing table names and determine if there are naming restrictions imposed by the output workspace. If the output workspace is an RDBMS, it may have reserved words that may not be used as a table name. It may also have invalid characters that are not to be used in a table or field name. ValidateTableName will return a string representing a valid table name that may be the same as the input name if the input name is valid. The example below guarantees that the new output feature class created by the Copy Features tool will have a unique name that is valid in any geodatabase:

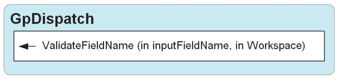
GpDispatch

← ValidateTableName (in inputTableName, in Workspace)

```
# Move all shapefiles from a folder into a geodatabase
# Import COM Dispatch module
import win32com.client

# Create the geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

# Set the workspace. List all of the shapefiles
GP.Workspace = "D:\St_Johns"
fcs = GP.ListFeatureClasses("*")

# Reset the enumeration to make sure the first object is returned
fcs.reset()
# Get the first feature class name
fc = fcs.next()
while fc: # While the feature class name is not empty
    # Copy the features from the workspace to a geodatabase
    GP.Workspace = "Database Connections/Bluestar.sde"
    # Validate the output name so it is unique and valid
    outfc = GP.ValidateTableName(fc)
    GP.Copyfeatures(fc, outfc)
    # Get the next feature class name
    fc = fcs.next()
```

Each database may have naming restrictions for field names in a table. Objects such as feature classes or relationship classes are stored as tables in an RDBMS, so these restrictions affect more than just standalone tables. These restrictions may or may not be common between various database systems, so scripts should check all new field names to ensure that a tool does not fail during execution. The example below will ensure that a field will be added no matter what the input name is, using the ValidateFieldName method:

Whenever a script is updating a dataset, such as a feature class or table, be careful to avoid situations where the dataset is locked. Microsoft Access always locks a database for update operations when a table is accessed, so if you have a personal geodatabase open in ArcCatalog, a script will not be able to update any of the geodatabase's contents until it is deselected and the folder is refeshed or ArcCatalog is closed. This includes script tools.



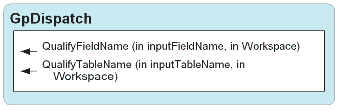
```
# Create a new numeric field containing the ratio of polygon area to
# polygon perimeter. Two arguments, a feature class and field name
# are expected.
# Import COM Dispatch module, system and operating system modules
import win32com.client, sys, os

# Create the geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

# Check the number of arguments
if len(sys.argv) < 2:
   print "Script requires two arguments, feature class and field name"
else:
    try:
        # Get the input feature class and make sure it contains polygons
       input = sys.argv[1]
       dsc = GP.Describe(input)
       if dsc.ShapeType != "polygon":
           raise "ShapeError"

       # Get the new field name and validate it
       fieldname = sys.argv[2]
       fieldname = GP.ValidateFieldName(fieldname,
       os.path.dirname(input))

       # Make sure shape_length and shape_area fields exist
       if GP.ListFields(input,"Shape_area").Next() and \
       GP.ListFields(input,"Shape_length").Next():
           # Add the new field and calculate the value
           GP.AddField(input, fieldname, "double")
           GP.CalculateField(input,fieldname,
           "[Shape_area] / [Shape_length]")
        else:
           raise "FieldError"

   except "ShapeError":
       print "Input does not contain polygons"
   except "FieldError":
       print "Input does not shape area and length fields"
   except:
       print GP.GetMessages(2)
```

Python evaluates an empty data structure as false, and any nonempty data structure as true. A numeric value of zero or an empty string is also evaluated as false.

Geodatabases that reside in an RDBMS, such as Oracle, SQL Server, or IBM DB2, use standard database naming conventions for identifying tables for specific users. An RDBMS may contain thousands of tables from many different users; therefore users of an enterprise geodatabase must understand how to correctly qualify the name of an object so the correct feature class, relationship class, feature dataset, or table is used during an operation. If an unqualified name is specified as input to a tool, the geoprocessor will qualify it automatically using the currently connected user name, which is specified as a property of the connected workspace. If a script needs to access data from a number of users, it should qualify the name of the table using the geoprocessor's QualifyTableName method so the syntax of the qualified name is correct. It requires the name of a table, the username and the path to a geodatabase. Output parameter values do not need to be qualified because a tool's output is always created by the connected user of the workspace. The example below shows how a script qualifies table names so it can use tables from a number of users:

ArcGIS desktop applications, such as ArcMap and ArcCatalog, always present fully qualified feature class, table, and field names. If you are unsure of the owner of a table or which tables are accessible to you, use ArcCatalog to view the contents of the geodatabase.

```
# Clip all of the featureclasses in a feature dataset
# Import COM Dispatch module
import win32com.client

# Create the geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")


# Create the list of input featureclasses
GP.Workspace = "Database Connections/Calgary.sde/transportation"
fcs = GP.Listfeatureclasses()
fc = fcs.Next()

while fc:
   outfc = GP.ValidateTableName(fc,
   "Database Connections/Calgary.sde/clip")
   # Qualify the clip fc name as it is owned by Grace
   clipfc = GP.QualifyTableName("Grace","study_area",
   "Connections/Calgary.sde")
    try:
      GP.Clip(fc, clipfc, outfc)
   except:
      print GP.GetMessages(2)

   print "Successfully clipped " + fc
```


GpDispatch
QualifyFieldName (in inputFieldName, in Workspace)
QualifyTableName (in inputTableName, in Workspace)

Field names such as Area, Entity, and Len are used by ArcSDE system tables, and they require scripts to fully qualify fields with these names. Refer to the ArcSDE Administration Guide *for more information on ArcSDE field names.*

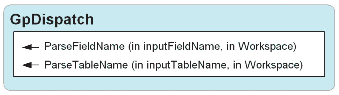
SQL queries may contain the names of a number of fields from two or more tables. It is not uncommon for the same field name to exist in separate tables, especially when working with foreign and primary keys. Qualified field names must also be used when a table contains certain field names that are also used by ArcSDE. To resolve the ambiguity between duplicate names, the field name must be qualified with the table or view name. The QualifyFieldName method allows a database independent mechanism for creating fully qualified field names, using the table and field name as input. The script below is an example of a script that will always create a correct SQL statement, regardless of the underlying database type:

```
# Update a selected set of parcels based on a SQL query
# Import COM Dispatch module
import win32com.client

# Create the geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

try:
   GP.Workspace = "Database Connections/Calgary.sde/LegalFabric"
   # Create the layer for the selection
   GP.MakeLayer("taxparcels","parcels")
   # Join the parcel updates table using parcel_id field
   GP.Addjoin("parcels", "parcel_id", "parcel_updates", "parcel_id")
   # Qualify the identically named fields
   id = GP.QualifyFieldName(parcels, "parcel_id")
   update_id = GP.QualifyFieldName(parcel_updates, "parcel_id")
   # Select the parcels using the qualified names
   GP.SelectLayerByAttribute("parcels","New_Selection",
    id + " = " + update_id)
   # Calculate the update flag to be true for the selected parcels
   GP.CalculateField("parcels", "updateflag", "true")

except:
   print GP.GetMessages(2)
```

Parsing table and field names

Scripts that are used as the source of script tools can make some assumptions about their input argument values. Paths to data are always fully qualified by the geoprocessing framework. Scripts that may be run outside an ArcGIS application should not make the same assumption. Refer to Chapter 5 for more information on script tools.

Scripts should use the geoprocessor's ParseTableName and ParseFieldName methods to split the fully qualified names for a dataset or for a column in a table into its components (database, owner, table, column). Scripts that need to be RDBMS independent should not assume that '.' is the delimiter used to separate the components of a fully qualified dataset name. ArcGIS applications always use fully qualified names, so if your script is being used as the source of a script tool, any feature class name, for example, will need to be parsed if the script needs to determine the name of the feature class without the user and database names. In the example below, the script is checking the input feature class for specific user names. The script could use the parsing functions of Python to split the qualified name, but it would then assume a specific syntax, which may change if another database type is used. ParseTableName returns a single string with the database name, owner name, and table name separated by commas. ParseFieldName returns the table name and the field name, also separated by commas. A script can then reliably parse the returned string, as these methods always return the same formatted string.


GpDispatch
ParseFieldName (in inputFieldName, in Workspace)
ParseTableName (in inputTableName, in Workspace)

A string in Python has a number of native methods for manipulation. In this example, the split method is used to create a Python list, using the comma as the delimiter. Other string methods include upper, lower, strip and count.

A Python list is an ordered collection of any type of object, such as strings or numbers. Lists are zero-based and may be used to effectively handle arrays of values when the order of the list's contents is known.

```
# Append input feature class to another feature class and update field
# Import COM Dispatch module and system module
import win32com.client, sys

# Create the geoprocessor object
GP = win32com.client.Dispatch("esriGeoprocessing.GPDispatch.1")

# Get the name of the input feature class and parse it.
GP.Workspace = os.path.dirname(sys.argv[1])
# Create a list and populate it
fullname = GP.ParseTableName(os.path.basename(sys.argv[1]))
nameList = fullname.split(",")
database = nameList[0]
owner = nameList[1]
featureclass = nameList[2]

# Qualify the name of the feature class that will be appended to and set
# the workspace using the administrator's connection
GP.Workspace = "Database Connections/Trans_admin.sde"
AppendFC = GP.ValidateTableName("common", "roads")

try:
  if owner == "grace":
     GP.CalculateField(fullname, "AppendedBy", owner)
     GP.Append(fullname, AppendFC )
  elif owner == "reed":
     GP.CalculateField(fullname, "AppendedBy", owner)
     GP.Append(fullname, AppendFC )
  else:
     GP.AddError("Unknown user of input feature class")

except:
   GP.AddError(GP.GetMessages(2))
```

Index