# Chapter 7 Manipulating spatial data

## 7.1 Introduction

This chapter introduces the ArcPy data access module arcpy.da for working with data.

This module allows control of an edit session, editing operations, cursor support, functions for converting tables and feature classes to and from NumPy arrays for additional processing, and support for versioning and replica workflows.

This chapter focuses on cursors, which are used to iterate over rows in a table.

Different types of cursors can be used to search records, add new records, and make changes to existing records.

Search cursors can be used to carry out SQL query expressions in Python.

Validation of text and field names is also covered

## 7.2 Using cursors to access data

In chapter 6, you saw how list functions can be used to iterate over a set of values in a list, including feature classes, tables, and fields.

A similar approach using a cursor is used to work with the rows in a table.

A cursor is a database technology term for accessing a set of records in a table.

Conceptually, it is similar to the way list functions work, where a cursor is used to manipulate a list of records.

A cursor can be used to iterate over the set of records in a table or to insert new records into a table.

Records in a table are also referred to as rows.

In ArcGIS, cursors can be used to read and write geometries to and from records, row by row

There are three types of cursors: search, insert, and update.

These cursors have the following characteristics:

A search cursor is used to retrieve rows.

An insert cursor is used to insert rows.

An update cursor is used to update and delete rows

Cursors navigate in a forward direction.

If a script needs to make multiple passes over the data by iteration, the cursor function must be run again.

Search and update cursors can iterate using a for loop or in a while loop using the cursor's next method to move on to the next row.

For a table with n rows, a script needs to make n calls to this next method.

When the last row is reached, a call to the next method will return a StopIteration exception.

All three cursors have two required arguments: an input table and a list (or tuple) of field names.

Search and update cursors also have several optional arguments.

The general syntax for each cursor is as follows:

Search and update cursors also support with statements.

A with statement has the advantage of guaranteeing closure and release of database locks and resetting iteration, regardless of whether the cursor finished running successfully or not.

The previous example using a with statement is coded as follows:

The following code illustrates how a new row is inserted.

A cursor object is created using the InsertCursor function.

Once the cursor is created, the insertRow method is used to insert a list of values that will be included by the new row.

By default, the new rows are inserted at the bottom of the table.

Fields in the table that are not included in the cursor are assigned the field’s default value, typically "null" (this w ill vary with the database format).

The update cursor is used to update or delete the row at the current position of the cursor.

The updateRow method is used to update the row.

After a row object is retrieved from the cursor, the row is modified as needed and the modified row is passed to the table by calling the updateRow method.

In the following example, the cursor object is created using the UpdateCursor function.

In the for loop, the values of one held (Acres) in the row are updated using the values of another held (Shape\_Area).

If the Shape\_Area held is in square feet, the value needs to be divided by 43,560 to obtain the area in acres.

The row is updated using the updateRow method on the row object.

The deleteRow method is used to delete the row at the current position of the UpdateCursor.

After the cursor retrieves the row object, calling the deleteRow method deletes the row, as follows:

Insert and update cursors support editing operations.

In the ArcG1S geoprocessing framework, a lock is set on the table when the cursor object is created.

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 when a table or dataset is accessed.

For example, opening a feature class in ArcMap and performing a query will result in a shared lock on the dataset.

Multiple shared locks can exist, but no exclusive locks are permitted if there is already a shared lock.

An exclusive lock is applied when changes are being made to a table or dataset.

Examples include editing and saving a feature class in ArcMap, changing the schema of a table or feature class in ArcCatalog, and using an insert or update cursor on a feature class in Python.

Insert and update cursors apply an exclusive lock to a table or feature class, preventing other processes from making changes.

In addition, insert and update cursors cannot be applied if an exclusive lock already exists.

For example, two scripts cannot simultaneously create an insert or update cursor on the same dataset.

Once an exclusive lock is created by a script, the lock persists until the script releases the lock.

This is accomplished using the del statement to delete the cursor object creating the lock.

Without this statement, a lock could needlessly block other applications or scripts from accessing a data set.

A typical script that creates an insert or update cursor should therefore include two del statements - one to delete the row object, such as del r ow, and one to delete the cursor object, such as del cursor.

For example:

>>> TIP

Forgetting to use the de1 statements at the end of a script can lead to errors so be sure to include the del statements after using insert and update cursor.

Search and update cursors suppor with statements, which guarantee closure and release of database locks.

Therefore, when a with statement is used, the del statements are not needed

## 7.3 Using SQL in Python

One of the most common processing steps in geoprocessing is to apply a query using Structured Query Language (SQL).

SQL is used to define one or more criteria based on attributes, operators, and calculations.

For example, SQL is used in the Select By Attributes dialog box in ArcMap, as well as in many different tools in ArcToolbox, including the Select tool

SQL queries can be carried out in Python using the SearchCursor function.

The syntax for SearchCursor is:

The optional where\_clause parameter consists of an SQL expression.

SQL syntax varies slightly depending on the dataset being queried, and the same syntax rules apply when SQL is used in the search cursor parameters.

The following code shows an example of using an SQL expression:

SQL syntax can be cumbersome since it depends on the format of the feature class.

For examp1e, field delimiters for shape files and file geodatabase feature classes consist of double quotation marks (“”), personal geodatabase feature classes use square brackets ([ ]), and ArcSDE geodatabase feature classes use no de1imiters.

To avoid confusion and to ensure the field delimiters are the correct ones, you can use the AddFieldDelimiters function.

The syntax of this function is

The function recognizes the type of dataset being used, such as a shapefile or personal geodatabase, and adds the correct de1imiters.

In the following example code, the field name is first assigned to a variable and the AddFieldDelimiters function is used to add the correct delimiters to the field name prior to using it in an SQL expression:

SQL is commonly used in other functions as well.

A number of bui1t-in tools use SQL-for example, the Select tool.

The generic syntax of the Select tool is:

The where\_clause parameter is an SQL expression, and to avoid confusion over the proper field delimiters, the following example code also uses the AddFieldDelimiters function:

## 7.4 Working with table and field names

When working with many different datasets, it is important to be able to determine whether tables and fields have a valid and unique name.

This is especially important when creating data in a geodatabase to avoid accidentally overwriting data.

The Exists function covered in chapter 6 can be used to determine whether a specific table is unique to a given workspace.

The ValidateTableName function can be used to determine whether a specific table name is valid for a specific workspace.

The syntax of this function is:

The function takes a table name and a workspace and returns a valid table name for the workspace.

If the table name is already valid, the function returns a string that is the same as the input.

Names may have invalid characters, which are replaced by an underscore (\_).

Specific databases may also have words that are reserved and cannot be used in a table name.

For example, the following script determines whether all roads is a valid table name for the file geodatabase called study:

In this example, the va1ue all\_roads is returned for the table name.

An underscore (\_) is added because the tab1e name cannot contain spaces.

Va1idating tab1e names is common1y used when moving data from one work space to another.

For example, the following code uses the Copy Features tool to copy all shapefiles from a folder to a geodatabase.

The .shp file extension is removed from the file name using the basename property and the names are va1idated prior to copying the shape files to feature classes, as follows:

A similar approach can be applied to field names using the ValidateFieldName function.

Fields cannot be added unless their name is valid, so unless field names are validated first, a script may fail during execution.

The syntax of this function is:

The ValidateFieldName function takes a field name and a workspace and returns a valid field name for the workspace.

Invalid characters are rep1aced by an underscore (\_).

The following code validates the name of a new field, and then uses the returned value to add the new field using the Add Field tool:

Similar to the ValidateTableName function, the ValidateFieldName function does not determine whether the field name exists so a script could still fail or an existing field could still be overwritten.

To determine whether a field name exists, the ListFields function can be used to create a list of the fields in a table or feature class, and the new field name can be compared against this list.

An alternative to trying to determine whether a table name exists is to use the Create UniqueName function.

This function creates a unique name in the specified workspace by appending a number to the input name.

This number is increased until the name is unique.

For example, if the name "Clip" already exists, the CreateUniqueName function will change it to "Clip0"; if this name also exists, the function will change the name to "Clip1," and so on.

This function can be used only to create unique table names within a workspace.

It does not work for field names. For example:

When this code is run the first time and the file buffer.shp does not exist, the resulting feature class is called buffer.shp .

When the code is then run a second time, the resulting feature class is called buffer0.shp.

You may have noticed that most geoprocessing tool dialog boxes use a similar approach to providing default names for output datasets.

For example, when the Clip tool is run on a feature class, the output, by default, is the name of the input feature class followed by "Clip,” such as rivers\_Clip.

When the Clip tool is run again on the same feature class and the work space has not been altered, the default name becomes rivers\_Clip1.

## 7.5 Parsing table and field names

The ArcGIS geoprocessing environment always uses fully qualified names for feature classes and tables.

For example, to process a feature class called roads, a geoprocessing tool needs to know not just the name, but also the path, the name of the database, and the owner of the data.

This is essential when working with geodatabases.

The ParseTableName function can be used to split the fully qualified name for a dataset into its components.

The syntax of this function is:

The ParseTableName function returns a single string with the database name, owner name, and table name, each separated by a comma (,).

For example, the following code uses the ParseTableName function on a feature class to obtain the components of the fully qualified name, and then splits these components into a list:

A similar approach can be used to split the fully qualified name of a field in a table into its components using the ParseFieldName function.

This function returns a single string with the database name, owner name, table name, and field name, each separated by a comma (,).

## 7.6 Working with text files

So far, most of the information data you have worked with in this book, such as paths, values, lists of values, and more, is located inside the script itself or inside data files in GIS format, such as shapefiles, geodatabases, and database tables.

In many cases, information data is also located in plain text files.

Python has several functions to work with text files in other formats.

1n many cases, these fil1es come from other applications and Python can be used to manipulate these files for use in ArcGIS.

You can open files using the open function, which has the following syntax:

The only required argument is a file name and the function returns an object.

For example, the following code opens an existing text file (.txt) on disk.

Using only a file name as a parameter returns a file object you can read from.

If you want to do something else, such as write to the file, this must be stated explicitly by specifying a mode.

The most common values for the mode are as follows:

If no mode argument is provided, the read mode is used by default.

Write mode allows you to write to the file.

Read/write mode can be added to any of the other modes to indicate both reading and writing is allowed.

Binary mode allows you to change the way a fi le is handled.

By default, Python assumes you are dealing with text files, which contain characters.

If you are working with some other kind of file such as an image, you can add b to the mode-for example, "rb".

Append mode means that any data written to the file is automatically added to the end of the file.

The buffering parameter controls the buffering of the files.

The default is unbuffered, which means all reads and writes go directly to and from disk.

When buffering is set to True, Python may use memory instead of disk space to improve performance.

For modest file sizes, buffering is normally not needed.

To create a new file object, you can use the open function and specify write mode:

Several file methods exist to manipulate the contents of a text file, including write, read, and close.

Consider the following example:

After you start reading a file, you can use the seek method to set the file’s current position without opening the file again.

For example:

One of the most basic ways of iterating over the contents of a file is to use the read method in a while loop.

For example, the following code loops over every character in the file:

First, files in Python are iterable, which means you can use them directly in a for loop to iterate over their lines.

The code is as follows:

For relatively small files, you can also read the entire file in step using the read method (to read the entire file as a string) or the readlines method (to read the file into a list of strings).

For example:

The following script opens an existing file in read mode and creates a new output file in write mode.

A for loop is used to iterate over the lines in the input file.

1n the block of code that follows, the replace method is used three times to remove specific strings from each line.

The resulting string is written to the output file.

The code is as follows:

# Chapter 10 Map scripting

## 10.1 Introduction

This chapter describes the ArcPy mapping module, also referred to as arc mapping.

The ArcPy mapping module helps automate mapping tasks, including managing map documents, data frames, layer files, and the data within these files.

There is also support for the automation of map export and printing, as well as the creation of PDF map books.

## 10.2 Working with the ArcPy mapping module

The ArcPy mapping module can be used to automate ArcMap workflows to speed up repetitive tasks.

Some typical examples of uses for the ArcPy map ping module are as follows:

* Finding a layer with a particular data source and replacing it with another data source
* Modifying the display properties of a specific layer in multiple ArcMap documents
* Generating reports that describe the properties of ArcMap documents, including data sources, layers with broken data links, information on the spatial reference of data frames , and more

The highly visual ArcMap environment is the recommended application for creating new map documents and creating map layers and map layouts.

Once they are created, however, the ArcPy mapping module can be used for scripting to automate certain mapping tasks, especially repetitive tasks across large numbers of map documents and elements.

The ArcPy mapping module cannot be used to customize the ArcMap interface, but it allows you to automate many of the tasks you wou1d normally carry out there.

Working with the ArcPy mapping modu1e follows the workflow that would be used within an ArcMap session.

For example, a typical workflow cou1d consist of opening a map document, modifying properties of a data frame, adding a layer, modifying the properties of that layer, changing several elements of the page layout, and then exporting the map to a PDF file.

These steps can be automated using scripts that employ the functions and classes of the ArcPy mapping module.

## 10.3 Opening map documents

A map document, or MXD, is stored as an .mxd file on disk-for example, C: \Mapping\Study\_Areas.mxd.

The ArcPy mapping modu1e allows you to open and manipulate .mxd files, in addition to layer (.lyr) files, which contain properties for individual layers.

There are two ways to start working with a map document using the ArcPy mapping modu1e: (1) use the map document from the current ArcMap session or (2) reference an existing .mxd file stored on disk.

The MapDocument function is used to accomplish both.

The syntax of the MapDocument function is

The path is a string representing an .mxd &le on disk. The following code opens a map document:

To use the current map document in ArcMap, the keyword CURRENT (in all uppercase letters) is used:

To use the CURRENT keyword, ArcMap must be running because the MapDocument object will reference the map document current1y 10aded into ArcMap.

Script too1s that use the CURRENT keyword must be run from within ArcMap to run properly.

Creating a script tool is covered in chapter 13.

Background processing must be disab1ed to properly use the current map document.

On creating a script tool, one of the properties that can be set is "Always run in foreground" this is recommended when using the CURRENT keyword because it overrides the default background processing settings of the current ArcMap session.

When an existing .mxd file is used, the script can be run independently of ArcMap.

The use of a system path to open an ArcMap document is recommended, however, because it makes the script more versatile and gives more control over how the script is run.

Still, using the CURRENT keyword can be very useful for quickly testing code in the Python window.

The Map Document object provides access to many different properties and methods of map documents.

The MapDocument object also provides access to the other objects within a map document.

The MapDocument object is a required parameter for many functions in the ArcPy mapping module.

As a result, the MapDocument object is typically one of the first object references created in a mapping script.

Once the MapDocument object is created, properties of the map document can be modified.

Before looking at how these changes are made, first consider how they are saved.

If you are working with a map document in ArcMap and you make a change, such as adding a layer, there are two ways to save the .mxd file: Save and Save As.

When Save is used, the changes are saved to the same .mxd file; when Save As is used, the changes are saved to a new .mxd file that you specify.

In a scripting environment, however, the MapDocument variable always points to the original map document on disk or currently in memory.

So there is no Save As option in the scripting environment and MapDocument uses only the save and saveACopy methods.

However, saveACopy accomplishes the same thing as the Save As option in ArcMap, and also allows you to save the file to a previous version.

After changes are made to the current map document, the map may not automatically be updated with every line of code.

The functions RefreshActiveVew and RefreshTOC can be used to refresh the map document.

This is similar to using the Refresh option in ArcMap (from the menu bar, click View > Refresh).

When a MapDocument object is referenced in a script, the .mxd file is locked.

This prevents other applications from making changes to the file.

It is therefore good practice to remove the reference to a map document when it is no longer needed in a script by using the Python del statement.

A mapping script therefore often has a structure that looks something like the following:

## 10.4 Accessing map document properties and methods

The properties of a MapDocument object include most of the properties found on the Map Document Properties dialog box (from the ArcMap menu bar, click File > Map Document Properties).

This includes properties such as the title and author of the map document, the date the map document was last saved, and whether the Relative paths check box has been selected A complete description of these properties can be found in the ArcPy documentation in ArcG1S Desktop Help.

In addition to properties, the MapDocument object provides a number of methods.

These include the save and saveACopy methods already mentioned, as well as methods for working with thumbnail images (delete Thumbnail and makeThumbnail ), and methods for modifying workspaces (findAndReplaceWorkspacePaths and replaceWorkspaces).

These last two methods are described in more detail in section 10.7 on fixing broken data sources.

In the following example, the CURRENT keyword is used to obtain the map document currently open in ArcMap , and the filePath property is used to print the system path for the .mxd file:

The following example updates the current map document’s title and saves the .mxd file:

## 10.5 Working with data frames

Map documents contain one or more data frames and each data frame typically contains one or more layers.

Data frames and layers are perfect objects for use in lists, which can help automate tasks.

The ListDataFrames function returns a list of DataFrame objects in a map document.

The syntax is as follows:

Once you have a list of data frames in a map document, you can look through them to examine or modify their properties.

Running the following code prints a list of all the data frames in a map document:

If you want to work with just one of the data frames, you can use its index number, as follows:

The order of a list of data frames is the same as the order used in the ArcMap table of contents.

DataFrame object properties, such as map extent, scale, rotation, and spatial reference, use map units.

Other properties use page units to position and size the data frame on the layout page.

Data frames are also used to access other objects-for example, the ListLayers function is used to access the layers in each data frame.

You can then loop through the layers to get their properties.

It is therefore important to uniquely name the data frames within a single map document.

There are quite a number of data frame properties, which are described in the ArcPy documentation in ArcGIS Desktop He1p.

The properties of the DataFrame object are a subset of all the properties on the Data Frame Properties dialog box in ArcMap (right-click a data frame in the Table Of Contents window and click Properties).

Scripting does not provide access to all the properties on the Data Frame Properties dialog box, and conversely, some DataFrame object properties are not on the Data Frame Properties dialog box.

For example, the scale of a data frame can be set using scripting, but in ArcMap, it is accomplished by using a tool on the Standard toolbar.

In most cases when you work with a map document, you are not interested in changing all the properties of a data frame, but only a few selected ones.

For example, in the following code, the spatial reference of all data frames in a map in most cases when you work with a map document, you are not interested in changing all the properties of a data frame, but only a few se1ected ones.

For example, in the following code, the spatial reference of all data frames in a map document is set to the same spatia1 reference as that of a specific shapefile, and the scale of all data frames is set to 1: 24 ,000:

In addition to the properties a1ready discussed, the DataFrame object a1so has two methods: panToExtent and zoomToSelectedFeatures.

The panToExtent method maintains the data frame sca1e but pans and centers the data frame extent based on the properties of an Extent object, which has to be provided as a parameter.

An Extent object is a rectangle specified by providing the coordinates of the lower-left corner and the upper-right corner in map units.

In most cases, this property is derived from an existing object, such as a feature or a layer.

For example, the getExtent method can be used to obtain the extent of a layer.

The following code pans the extent of a data frame called df based on the extent of the features in a layer object called lyr :

The zoomToSelectedFeatures method is similar to the ArcMap operation Selection > Zoom to Selected Features.

Running the following code zooms to the extent of all selected features in a data frame called df:

If no features are selected, the code will zoom to the full extent of all layers.

## 10.6 Working with layers

A data frame typically contains one or more layers and the Layer object is essential to managing these layers.

The Layer object provides access to many different layer properties and methods.

There are two ways to reference Layer objects.

The first approach is to use the Layer function to reference a layer (.lyr ) file on disk.

It is similar to how map document files (.mxd) are referenced.

The syntax of the Layer function is

The parameter of the Layer function is the full path and &le name of an existing .lyr file.

For example:

The second approach is to use the ListLayers function to reference the layers in an .mxd file, or just the layers in a particular data frame in a map document, or the layers within a .lyr file.

The syntax of the List La yers function is

The only required element is a map document or layer file.

An optional wild card can be used to limit the result.

An optional data frame variable can be used that references a specific DataFrame object.

For example, the following code returns a list of all the layers in an ArcMap document, and then prints the names of all the layers:

To access just the layers in a specific data frame, the Data Frame object has to be referenced as a parameter.

In the following example, the ListLayers function returns only the layers in the data frame that have index number 0.

The wild\_card parameter is skipped using an empty string (“”)

The following code illustrates how to reference the layers in a .lyr file on disk and print the name of layer objects:

Once you reference one or more Layer objects using either the Layer or ListLayers function, you have access to many of the common layer properties found on the Layer Properties dialog box in ArcMap.

The Layer object also provides methods for saving layer files.

There are many types of layers in ArcMap and not all of them work in the same manner.

Three of the layer categories are commonly used: feature layers, raster layers, and group layers.

Properties of the Layer object can be used to identify the category you are working with and the supports method can be used to test the properties a layer supports.

For example, a definition query wou1d work only on a feature layer.

But rather than remembering this aspect or checking it manually, you could use the supports method to test whether a particular layer supports a particular property.

There are a number of other more specialized layers, such as annotation subclasses, network datasets, topology datasets, and others.

These layers may also require testing of properties to ensure they are supported.

Layer objects have a number of properties.

These include the name of the layer, the name of the layer dataset, the ability to set a definition query, the ability to turn on the display of labels, and a number of display properties, such as brightness, contrast, and transparency.

A complete description of all the properties of a Layer object can be found in the ArcPy documentation in ArcGIS Desktop Help.

In the first version of ArcPy released with ArcGIS 10.0, emphasis was given to the properties that were most likely to benefit from automation.

Additional properties are included with ArcGIS 10.1, such as layer symbology and access to a layer's time properties.

Other properties may be included as well in future versions of the ArcPy mapping module.

A few examples will serve to illustrate the use of layer properties.

For example, the following code turns on all the labels for the layers in the current map document using the showLabels property:

Instead of changing the properties of all the layers in a map document or a data frame, the layer properties can also be used to find a layer with a particular name.

For example, the following code searches for a layer called "hospitals":

Layer names can be a bit confusing.

The name of a layer is what is shown in the ArcMap table of contents.

This may or may not be the same as the name of the source dataset for the layer

In any case, the name of a layer does not have an extension.

So the name of a feature class could be hospitals.shp, but as a layer in ArcMap, the name of the layer is hospitals.

Also remember that strings are case sensitive, so Hospitals is different from hospitals.

To make your statements insensitive to case, you can use basic string operators.

For example:

Several other layer properties involve names.

The datasetName property returns the name of the layer dataset as it appears in the workspace.

This does not, however, include any file extensions.

The dataSource property returns the full path of the layer dataset.

So for a layer that appears as Hospitals in the ArcMap table of contents, the datasetName property may be hospitals and the dataSource property may be C:\Data\hospitals.shp.

Both datasetName and dataSource are read-only properties, whereas the name property is read/write and can be changed.

Finally, there is the longName property, which is useful for describing group layers because it includes the group layer and sublayer names.

A number of methods exist for Layer objects.

These include the save and saveACopy methods, which save a .lyr file.

The findAndReplaceWorkspacePath and replaceDataSource methods are used to manipulate workspaces and are covered in more detail in the next section.

Because not all types of layers support the same properties, the supports method can be used to determine whether a layer supports a particular property.

This makes it possible to test whether a layer supports a property before trying to get or set its value.

This reduces the need for error checking.

In the earlier example where the labels were shown for all layers in a data frame, it would make sense to first use the supports method to test whether this property is supported for each layer.

The syntax of the supports method is

The parameter, in this case, would consist of one of the Layer object properties, such as brightness contrast, datasetName, or others.

The supports method returns a Boolean value, so the example code to test whether labeling is possible would look as follows:

If you are unsure whether a layer supports a particular property, use the supports method to test it.

Otherwise, you will need to use an error trapping method, such as a try-except statement, which is covered in chapter 11.

In addition to properties and methods of layer objects, there are several functions in the ArcPy mapping module that are specifically designed to manage layers within a data frame.

These include the following:

These functions all must reference an already existing layer.

It can be a layer file on disk, a layer from within the same map document, or a layer from a different map document.

Thus, these functions do not perform the task of adding data to a map document, as Add Data does in ArcMap.

## 10.7 Fixing broken data sources

Consider the following scenario: You open an existing map document that you have not used in some time.

Or perhaps a coworker has given you a disk or drive that has a project on it that contains map documents.

When you open the map document, the layers in the ArcMap table of contents are shown with a red exclamation mark next to them and none of the layers are showing in the data frame.

What happened? The link to the data source(s) has been lost.

This can occur in a number of different scenarios:

* The map document was saved with full paths and the path to the data source has changed-for example, by moving it to a different drive.
* The map document was saved with relative paths, but the .mxd file has been moved relative to the data source.
* The names of data source files have been modified.

These broken data sources can be fixed within ArcMap as follows: right-click layer, click Data > Repair Data Source, and browse to the correct data source.

A few strategies can be used to prevent such broken data sources in the first place, including saving map documents with relative paths and proper file management in general.

Broken data sources are very common, and fixing them manually can be tedious.

Scripting can be used to automate these corrections once the nature of the fix has been identified.

Changes can be made to the map document without even opening it.

Before examining these methods in more detail, a few definitions are in order:

* *Worhspace*

a container for data.

It can be a folder that contains shapefiles, a coverage, or a geodatabase-for example, mydata.

* *Worhspace path*

the system path to a workspace.

It includes the drive letter where the folder is located and any subfolders-for example, C:\mydata.

For a file-based geodatabase, it includes the name of the geodatabase - for example, C:\mydata\project.gdb.

* *Dataset*

the feature class or table in the workspace.

It is the actual name on disk, not the name displayed in the ArcMap table of contents.

For a shapefile, it would be something like hospitals.shp.

For a feature class in a geodatabase, it would be something like hospitals.

* *Data source*

the combination of workspace and dataset-for example, mydata\hospitals.shp or mydata\project.gdb\hospitals

Prior to using a map document, you should check for broken data sources using the ListBrokenDataSources function.

This arcpy.mapping function returns a Python list of layer objects within a map document or layer file that have broken connections to their original data source.

The syntax is

The following code illustrates how this function can be used to print the names of the layers in a map document that have broken data sources:

Running this code returns the names of the layers as they appear in the ArcMap table of contents.

Instead of the name, the layer property dataSource can be used to see the current data source being referenced by the layer, as follows:

This lists the data sources that are broken and could help identify what the correct data sources should be.

Keep in mind, however, that the ListBrokenDataSources function cannot identify what the correct data sources are - this can only be determined by a user with knowledge of the map documents and the data.

Once it is established that data sources need to be updated or fixed, the following methods can be applied to map documents, layers, or tables:

* findAndReplaceWorkspacePaths and replaceWorkspaces use to perform a find-and-replace operation on the workspace path and the workspace, respectively.

This assumes that the datasets are correct.

For example, you can change C:\mydata\hospitals.shp to C:\newdata\hospitals.shp, but the name of the dataset (in this case, hospitals.shp) cannot be modified.

* replaceDataSource -use to perform a find-and-replace operation on the workspace and the dataset.

You can modify both the workspace and the dataset, or just the dataset.

For example, you can change C:\mydata\hospitals.shp to C:\mydata\newhospitals.shp.

The following methods work on three different classes: MapDocument, Layer , and TableView objects.

In total, there are six different methods:

1. MapDocument.findAndReplaceWorkspacePaths

2. MapDocument.replaceWorkspaces

3. Layer.findAndReplaceWorkspacePath

4. Layer.replaceDataSource

5. TableView.findAndReplaceWorkspacePath

6. TableView.replaceDataSource

The syntax of Map Document.findAndReplaceWorkspacePaths is

Running this code searches for and replaces the workspace paths of all layers and tables in a map document that share that workspace.

For example, the following code replaces all the workspace paths in the current map document:

The methods in this group have an optional validation parameter.

This parameter allows you to verify if a workspace or dataset is valid before changing its value.

If validate is set to True (the default value) and the data source is valid, the data source will be updated.

If the data source is not valid, it will remain pointing to the original data source.

If validate is set to False, the workspace path or dataset does not have to be valid (that is, it does not already exist).

This condition would be used when you want to update the data sources prior to the data being created.

When replacing workspace paths, you can replace all or part of a path.

For example, if a workspace has simply moved from one drive to another, you can replace D: \ with C: \.

The MapDocument.findAndReplaceWorkspacePaths method works on multiple workspace types at once, including shapefiles, file geodatabases, and others.

However, the workspace type cannot be modified.

The MapDocument.replaceWorkspaces method can be used to modify both the workspace path and the workspace type-for example, from a folder containing shapefiles to a file geodatabase.

The method works on only one workspace at a time but can be used multiple times if more than one workspace type needs to be replaced.

The syntax of the MapDocument.replaceWorkspaces method is

For example, in the following code, references to shapefiles are redirected to feature classes in a file geodatabase:

Notice exactly what happened here.

The workspace is changed, but not the dataset.

For example, if the data source for a layer was C:\mydata\hospitals.shp, it has been modified to C:\mydata\database.gdb\hospitals.

Because the type of workspace is specified, the .shp extension for the datasets is automatically removed.

The example code assumes that feature classes with the exact same names as the shapefiles exist in the file geodatabase.

Remember that paths are not case sensitive.

* ACCESS\_WORKSPACE
* ARCINFO\_WORKSPACE
* CAD\_WORKSPACE
* EXCEL\_WORKSPACE
* FILEGDB\_WORKSPACE
* OLEDB\_WORKSPACE
* PCCOVERAGE\_WORKSPACE
* RASTER\_WORKSPACE
* SDE\_WORKSPACE
* SHAPEFILE\_WORKSPACE
* TEXT\_WORKSPACE
* TIN\_WORKSPACE
* VPF\_WORKSPACE

Notice that "personal geodatabase" is not specifically included as a type instead ACCESS\_WORKSPACE is used.

When workspaces in a map document are modified, there are a few things that may not work:

* + Joins and relates associated with raster layers and stand-alone tables are not updated.
  + Definition queries may no longer work because a slightly different SQL syntax is used-for example, between file geodatabases and personal geodatabases.

A slight modification to the SQL statement would fix this problem.

* + Label expressions may no longer work for the same reason.

The methods discussed so far work on map documents.

However, data sources can also be modified for individual layers.

The Layer.findAndReplaceWorkspacePath method works on a Layer object and performs a find-and-replace operation on the workspace path for a single layer in a map document or layer file.

The syntax of this method is

The following code modifies the workspace for a layer that references a particular feature class in a personal geodatabase.

Only a portion of the full path of the data source is rep laced-in this case, using a different personal geodatabase:

The Layer.findAndReplaceWorkspacePath method assumes the dataset has not changed.

The replaceDataSource method can be used to change both the workspace and the dataset.

The syntax of this method is

The following code replaces a specific data source.

In this case, the value of the dataSource property is used to determine whether a layer should have its data source updated:

The findAndReplaceWorkspacePath and replaceDataSource methods also exist for TableView objects.

The syntax for using these methods to work with single tables is very similar to the syntax for working with layers.

## 10.8 Working with page layout elements

Map scripting can also be used to work with page layout elements, including graphics, legends, pictures, text, and several others.

Typical properties that can be changed include name, size, position, and sometimes other properties that vary with each element type.

Similar to map documents, layout elements cannot be created using scripting so they have to already exist in a map document.

The ListLayoutElements function can be used to identify which elements exist within the layout of a particular map document.

The syntax of this function is

The ListLayoutElements function returns a Python list of elements.

The optional parameter element\_type can be used to limit the list of elements to only those of a specific type.

The specifc types of elements that can be used in scripting are as follows:

* + DATAFRAME\_ELEMENT
  + GRAPHIC\_ELEMENT
  + LEGEND\_ELEMENT
  + MAPSURROUND\_ELEMENT
  + PICTURE\_ELEMENT
  + TEXT\_ELEMENT

Each of these elements corresponds to a class in the arcpy.mapping module.

Several of these elements are described in this section in a bit more detail.

When getting started with layout elements, however, it can be useful to first create an inventory of what exists.

For example, the following code creates a list of all layout elements and prints their name and type:

The printout may look something like the following:

Notice that several different items are called MAPSURROUND\_ELEMENT.

Technically, any layout element that has an association with a data frame is a MAPSURROUND\_ELEMENT object.

This includes the north arrow, scale bar, and scale text.

A legend element is also associated with a data frame, but since it has some unique properties, it is a separate element type.

Once it is determined what layout elements are available, a specific element can be selected by using: (1) the index number of the element, (2) the element\_type parameter, or (3) the wild\_card parameter.

For example, to work with the title element, the following lines of code can be used to obtain a list with only the object that contains the title.

Using the index number directly:

Using the element\_type parameter:

Using the wild\_card parameter

In the case of the element\_ type and wild\_card parameters, the ListLayoutElements function returns a 1ist with only a sing1e object.

Using an index number of zero([0]) on this list returns the object instead of a list.

*Note: Not all elements have a default name, especially if they have been copied from other applications.*

*To use these elements in scripting, the user has to first manually set the name in the map document.*

Once a specific e1ement is referenced, various properties can be accessed, such as the element's name, type, height, and width, and the x, y coordinates of the element’s anchor position.

Other properties will vary with the type of element.

For example, an important property of the text Element object is the text property.

For a text element, all properties are read/write, with the exception of the type.

The following code modifies the text of the title in a page layout to a new string.

A few more examp1es of code fo11ow to illustrate some unique properties that can be modified using scripting.

A PictureElement object has a sourceImage property, which represents the path to the image data source.

The following code illustrates how this path can be modified:

The LegendElement object has an autoAdd property, which controls whether a layer should be automatically added to the legend when a layer is added to a data frame using the AddLayer function.

The following code illustrates how the autoAdd property can be modified to control which layer gets added:

Another useful property of the LegendElement object is items, which returns a list of the names of the individual legend items.

The LegendElement object also has one method, adjustColumnCount, which allows you to set the number of columns in the legend.

## 10.9 Exporting maps

The ArcPy mapping module has a number of exporting functions.

They are similar to the ArcMap operation File > Export Map.

There is a separate function for each format.

The functions are as follows:

These functions all work in a similar manner.

The only required elements for the export functions are a map document and the path and file name of the output file.

For example, the syntax of the ExportToJPEG function is as follows:

The optional parameters represent the export options, which are also found on the Export Map dialog box in ArcMap.

For example, for the JPEG format, these options look like the General and Format options shown in the two figures.

The dialog box options correspond directly to the parameters in the ExportToJPEG function.

All these parameters have default values, and typically only selected parameters need to be set.

The following code exports the page layout of a map document to a .jpg file, setting a resolution of 600 dpi:

Notice the use of empty strings ("") to skip several optional parameters.

One of the optional parameters in all export functions is the data\_frame parameter.

This parameter makes it possible to reference an individual DataFrame object to export, exporting just the data frame in Data View without any of the layout elements.

By default, the page layout is used for export, including all data frames and layout elements.

Many of the other parameters will vary with the specific format selected.

## 10.10 Printing maps

In addition to exporting maps to files, the ArcPy mapping module contains a basic PrintMap function, which prints a specific data frame or map document to a printer or file.

The syntax of this function is

The only required parameter is a map document.

An optional printer\_ name parameter can be specified to represent the name of a printer on a local computer.

If no printer is specified, the PrintMap function uses the printer that is saved with the map document or the default system printer if no printer is saved with the map document.

An optional data\_frame parameter can be used to reference a specific data frame-by default, the page layout is printed.

## 10.11 Working with PDFs

The PDF format has become widely used in the distribution of cartographic products.

1n addition to the ExportToPDF function, the ArcPy mapping module has a number of classes and functions to work with .pdf files.

First, there is the PDFDocument class.

This object allows for the manipulation of PDF documents, including the merging of pages, setting document behavior, and creating security settings.

The syntax of the PDFDocument class is:

The only parameter is a string that specifies the path and file name of the .pdf file.

A PDFDocument object has only one property: pageCount, which is an integer for the number of pages.

A PDFDocument object has five methods: appendPages, insertPages, saveAndClose, update DocProperties and updateDocSecurity.

There are two PDFDocument functions:

1. PDFDocumentCreate - creates an empty PDFDocument object in memory.

The function receives a path and file name to determine the save location where a new PDF file will be created.

1. PDFDocumentOpen - returns a PDFDocument object from a PDF file on disk

These functions are often used to create a PDF map book.

A number of separate .pdf files can be exported from map documents-for example, using the DataDrivenPages object discussed in the next section.

These are appended in a newly created PDFDocument object and saved as a final PDF map book.

The following code creates an empty PDFDocument using the PDFDocumentCreate function, and appends three existing .pdf files into a single PDF.

The saveAndClose method saves the resulting PDF, as follows:

## 10.12 Creating map books

ArcGIS has a set of too1s to create a map book, which is simply a collection of pages printed together.

A typical example of a map book includes an index page followed by individual maps.

The index page shows the extent of the individual maps.

An example map book is shown in the figure, including an index map and two of the many individual maps.

These map books can be created manually simply by printing each map separately.

However, ArcMap contains a toolbar called Data Driven Pages to automate this process.

More advanced map books require the use of scripting with the DataDrivenPages object in the ArcPy mapping module.

Automating the creation of map books using scripting requires that Data Driven Pages be enabled in the map document.

This can be accomplished using the Data Driven Pages toolbar in ArcMap.

On the Setup Data Driven Pages dialog box, a layer is selected that defines a series of extents-this layer is referred to as an index layer.

The DataDrivenPages object in the ArcPy scripting module can be used to access the properties and methods for managing the individual pages within the map document.

Note: A detailed explanation of how to work with Data Driven Pages and create map books is not provided in this book.

For detailed explanations of these procedures, see "Creating a map book" and "Creating Data Driven Pages" on the Contents tab in ArcGIS Desktop Help (Mapping > Page Layouts).

The exportToPDF method for the DataDrivenPages object can be used to create a map book in POF format.

This is not the same as the ExportToPDF function.

The syntax of the exportToPDF method is as follows:

The following code prints all the pages from a map document that has DataDriven Pages enab1ed to a PDF file and p1aces an existing cover page in front:

The Data Driven Pages toolbar and scripting can be used in combination to effectively produce map books.

Some of the inherent behavior of Data Driven Pages such as page extents, sca1es, dynamic text, and the like are probably easiest to contr01 on the Setup Data Driven Pages dia10g box in ArcMap , a1though printing the pages and merging different PDF files is easiest to control using scripting

## 10.13 Using sample mapping scripts

With the re1ease of ArcGIS 10, Esri started making a number of script to01s available to illustrate the use of ArcPy.

These include a set of mapping script to01s created as representative samp1es of how arcpy.mapping can be used to perform a variety of mapping tasks.

These tools can be found in the Geo processing Mode1 and Script Tool Gallery on the ArcGIS Resource Center.

*Note: To obtain the sample tools, go to* [*http://resources.ArcGIS.com*](http://resources.ArcGIS.com) *and in the Search box, type arcpy.mapping sample script tools.*

*This brings up a link to the sample tools.*

The sample tools consist of three different toolboxes: Cartography Tools (not to be confused with the existing Cartography system toolbox), Export and Printing Tools, and MXD and LYR Management Tools.

Each toolbox contains a number of script tools, each of which references a Python script.

The sample tools include most of the functionality covered earlier in this chapter.

Some of the scripts are quite short and simple.

For example, the Print Map Document(s) tool prints the layout page of one or more map documents to a local printer.

The tool dialog box, shown in the figure, allows you to select multiple map documents and select a local printer.

The Print Map Document(s) tool references the script file called PrintMXDs.py, which I sincluded in the files you get when downloading the tools.

The script tool uses the GetParameterAsText function to get the list of map documents and the local printer from a user.

You will learn about this function in chapter 13 on creating custom tools.

The script tool then uses the PrintMap function in the arcpy.mapping module to print the map documents.

It can be useful to review sample scripts like this to get ideas for your own scripts.

You can use these code examples as is, but you are encouraged to modify them or use parts of the code in your own scripts.

# Chapter 11 Debugging and error handling

## 11.1 Introduction

This chapter discusses debugging procedures and provides a review of the most common Python errors.

Error-handling procedures are also discussed, including how to get the most out of try - except statements No matter how careful you are in writing code, errors are bound to happen.

There are three main types of errors you will encounter in Python: syntax errors, exception, and logic errors.

Syntax errors prevent code from running.

With an exception, a script will stop running midprocess.

A logic error means the script will run but produce undesired results.

## 11.2 Recognizing syntax errors

Syntax errors pertain to spelling, punctuation, and indentation.

Common syntax errors result from misspelled keywords or variables, missing punctuation, and inconsistent indentation.

See if you can spot the error in the following code:

The colon (:) at the end of the first line of the for loop is missing.

The following syntax error is displayed when this code is run in the Python window:

PythonWin has a built-in checking process, which works somewhat like a spell checker in a word-processing application.

The process is enabled by clicking the Check button on the PythonWin Standard toolbar.

This checks the current script file without running it.

Consider the preceding example code, which is shown in the figure.

When you click the Check button, a message appears on the PythonWin status bar: Failed to run scrpt - syntax error-invalid syntax .

The cursor is also moved to the line where the first syntax error was detected-in this case, line 5, as shown in the fingure.

## 11.3 Recognizing exceptions

Syntax errors are frustrating, but they are relatively easy to catch compared to other errors.

Consider the following example that has the syntax corrected:

When you run the script again, it runs without a syntax error.

But what if no count is printed?

Is that an error?

Perhaps the workspace is incorrect, or perhaps there are no feature classes in the workspace.

Rather than referring to these incidents as “error,” it is common for programming languages to discern between a normal course of events and something exceptional.

There might be errors, but there might simply be events you might not expect to happen.

These events are called exceptions.

Exceptions refer to errors that are detected while the script is running.

When an exception is detected, the script stops running unless the detection is handled properly.

Exceptions are said to be thrown.

If the exception is handled properly-that is, it is caught-t he program can continue running Examples of exceptions and proper error-handling techniques are covered later in this chapter.

## 11.4 Using debugging

When code results in exception errors or logic errors, you may need to look more closely at the values of variables in your script.

This can be accomplished using a debugging procedure.

Debugging is a methodological process for finding errors in your script.

There are a number of possible debugging procedures, from very basic to more complex.

Debugging procedures include the following:

* + Carefully reviewing the content of error messages
  + Adding print statements to your script
  + Selectively commenting out code
  + Using a Python debugger

Each of these approaches is reviewed in this section in more detail.

Keep in mind that most of the time, debugging does not tell you why a script did not run properly, but it will tell you where-that – that is, on which line of code it failed.

Typically, you still have to figure out why the error occurred

# Chapter 12: Creating Python functions and classes

## 12.1 Introduction

This chapter describes how to create custom functions in Python that can be called from elsewhere in the script or from another script.

Custom functions make it easy to reduce the code you have written to carry out procedures.

Functions are organized into modules, and modules can be organized into a package.

ArcPy itself is a collection of custom modules and functions organized into a package.

By creating custom functions, you can organize your code into logical parts and reuse frequently needed procedures.

This chapter also describes how to create custom classes in Python, which makes it easier to group together functions and variables.

## 12.2 Creating functions

Functions are small blocks of code that perform a specific task.

Python itself has a great number of built-in functions and the ArcPy site package contains a large number of functions, including all the geoprocessing tools in ArcGIS.

You will use many built-in functions in a typical Python script, and you can import additional functionality from other modules, including ArcPy.

Consider the random module, for example.

You can import this module for access to a number of different functions.

The following code generates a random number between 1 and 100:

The code to generate a random number has already been written, and this code can now be freely used by anyone who needs it.

The code of the random module can be found in a file called random.py and is located in the Python Lib folder.

1n a typical installation of Python 2.7 as part of the ArcG1S 10.1 installation, the path is: C: \Python27\ArcG1S10.1\Lib\random.py.

You can open this script in a Python editor like PythonWin and examine the code.

1nside the code, you will find a reference to the randint function, as shown in the figure.

In this example, the randint function calls another function called randrange.

The random module contains a number of different functions and some of them are closely related.

The point here is that the code to generate random numbers has already been written and shared with the Python user community.

So whenever your script needs a random number, you don't have to write the code yourself.

You can import the random module and use any of its functions.

In addition to using existing functions, you can create your own functions that can be called from within the same script or from other scripts.

Once you write your own functions, you can reuse them whenever needed.

This makes code more efficient since there is no need to write the same task over and over

Python functions are de fined using the def statement, as shown in the figure.

The def statement contains the name of the function, followed by any arguments in parens.

The syntax of the def statement is.

There is a colon (:) at the end of the statement, and the code following a def statement is indented the same as any block of code.

This indented block of code is the function definition.

For example, consider the script helloworld.py as follows:

In this example, the function printmessage has no parameters, but most functions use parameters to pass values.

Elsewhere in the same script, you can call this function directly, as follows:

Typically, functions are quite a bit more elaborate.

Consider the following example: You want to create a list of the names of all the fields in a table or a feature class.

There is no function in ArcPy that does this.

However, the ListFields function allows you to create a list of the fields in a table, and you can then use a for loop to iterate over the items in the list to get the names of the fields.

The list of names can be stored in a list object.

The code is as follows:

Now, say you anticipate that you will be using these lines of code quite often-in the same script or in other scripts.

You can simply copy the lines of code, paste them where they are needed, and make any necessary changes.

For example, it is likely you will need to replace the parameter "streams.shp" with the feature class or table of interest.

Instead of copying and pasting code, you can define a custom function to carry out the same steps.

First, you need to give the function a name - for example, listfieldnames.

The following code defines the function:

You can now call the function from elsewhere in the script by name.

In this example, when calling the function, you want to pass a value to the function-that the name of a table or a feature class.

To make this possible, the function needs to include a parameter to receive these values.

The parameter needs to be included in the definition of the function, as follows:

Following the def statement is an indented block of code that contains what the function actually does.

This is identical to the previous lines of code, but now the hard-coded value of the feature class is replaced by the parameter of the function:

The last thing needed is a way for the function to pass values, also referred to as returning values.

This is necessary to ensure that the function not only creates the list of names, but also returns the list so it can be used by any code that calls the function.

This is accomplished using a return statement.

The completed description of the function is as follows:

Once a function is defined, it can be called directly from within the same script, as follows:

Running the code returns a list of the fields in a table using the function previously defined.

Notice that the new function listfieldnames can be call directly, since it is defined in the same script.

The example function used a parameter called table, which makes it possible to pass a value to the function.

The parameter is also referred to as an argument.

A function can use more than one parameter, and parameters can be made optional.

The arguments for optional parameters should be ordered so that the required ones are listed first, followed by the optional ones.

Arguments can be made optional by specifying default values.

Creating functions can be beneficial in a number of ways:

If a task is to be used many times, creating a function can reduce the amount of code you need to write and manage.

The actual code that carries out the task is written only once as a function, and from that point on, you can call this custom function as needed.

Creating functions can reduce the clutter caused by multiple iterations.

For example, if you wanted to create lists of the held names for all the feature classes in all the geodatabases in a list of workspaces, it would quickly create a relatively complicated set of nested for loops.

Creating a function for creating a list of field names removes one of these for loops and places it in a separate function.

Complex tasks can be broken into smaller steps.

By defining each step as a function, the complex task does not appear so complex anymore.

Well-defined functions are a good way to organize longer scripts.

Custom functions can be called not only directly from the same script, but also from other scripts, which is covered in the next section

## 12.3 Calling functions from other scripts

Once functions are created in a script, they can be called from another script by importing the script that contains the function.

For relatively complex functions, it is worthwhile to consider making them into separate scripts or script tools, especially if they are needed on a regular basis.

So rather than defining a function within a script, the function becomes a script in itself that can be called from other scripts.

Consider the earlier example of the helloworld.py script:

The printmessage function can be called from another script by importing the helloworld.py script.

For example, the script print.py does it, as follows:

The script print.py imports the helloworld module.

A module name is equal to the name of the script minus the .py extension.

The function is called using the regular syntax to call a function-that is, <module>.<function>

In the example script, the helloworld module is imported into the print.py script.

Notice that there is no path associated with the module, but just the name itself, which is the name of another script.

So, the import statement causes Python to look for a file named helloworld.py.

No paths can be used in the import statement, and thus it is important to recognize where Python looks for modules.

The first place Python looks for modules is the current folder, which is the folder where the print.py script is located.

The current folder can be obtained using the following code, where sys.path is a list of system paths:

The current folder can also be obtained using the os module, as follows:

Next, Python looks at all the other system paths that have been set during the installation or subsequent configuration of Python itself.

These paths are contained in an environment settings variable called PYTHONPATH.

This variable can be set to a list of paths that will be added to the beginning of the sys.path list.

To view a complete list of these paths, use the following code:

In a typical scenario, the list will include paths to both the Python installation and the ArcGIS installation.

The list will include paths like the following:

What if the module you want to import is in a different folder-that is, not in the current folder of the script or in any of the folders in sys.path?

You have two options, as follows:

## 12.4 Organizing code into modules

By creating a script that defines a custom function, you are turning the script into a module.

All Python script files are, in fact, modules.

That's why you can call the function by first importing the script (module), and then using a statement such as <module>.<function>.

Recall the example:

The random module consists of the random.py file and is located in one of the folders that Python automatically recognizes, C:\Python27\ArcGIS10.1\lib The random.py script (module) contains a number of functions, including randint.

This makes it easy to create new functions in a script and call them from another script.

However, it also introduces a complication: how do you distinguish between running a script by itself and calling it from another script? What is needed is a structure that provides control of the execution of the script.

If the script is run by itself, the function is executed.

If the module is imported into another script, the function is not executed until it is specifically called.

Consider the example hello.py script, which contains a function as well as some test code to make sure the function works:

This type of testing is reasonable, because when you run the script by itself, it confirms that the function works.

However, when you import this module to use the function, the test code runs, as follows:

When you import the script file as a module, you don't want the test code to run automatically, but only when you call the specific function.

You want to be able to differentiate between running the script by itself and importing it as a module into another script.

This is where the variable \_\_name\_\_ comes in (there are two underscores on each side).

For a script, the variable has the value of "\_\_main\_\_".

For an imported module, the variable is set to the name of the module.

Using an if statement in the script that contains the function will make it possible to distinguish between a script and a module, as follows:

|  |
| --- |
|  |

In this case, the test of the module will be run only if the script is run by itself.

If you import the script, no code will be run until you call the function.

This structure is not limited to testing.

In some geoprocessing scripts, almost the entire script consists of one function or more, and only the very last lines of code actually call the function if, indeed, the script is run by itself.

The structure is as follows:

|  |
| --- |
|  |

This structure provides control of the running of the script and makes it possible to use a script in two different ways - running it by itself or calling it from another script.

Consider the earlier example of the random module.

The very last lines of code look like the example in the figure.

If the random.py script is run by itself, it will run the test function, as shown in the figure.

Running the script produces output than can be examined to ensure the random function performs as expected.

The output from running the random.py script is as follows:

These results are printed only when the random.py script is run by itself and not when it is imported as a module into another script.

## 12.5 Using classes

In the previous sections, you saw how to create your own functions and organize your code into modules.

This substantially increases code reusability because you can write a section of code and use it many times by calling it from within the same script or from another script.

However, these functions and modules have their limitations.

The principal limitation is that a function does not store information the way a variable does.

Every time a function is run, it starts from scratch.

In some cases, functions and variables are very closely related.

For example, consider a land parcel with a number of variables, such as the land-use type, total assessed value, and total area.

The parcel may also have procedures associated with it, such as how to estimate the property taxes based on land- use type and total assessed value.

These functions require the value of the variables.

These values can be passed to a function as arguments.

What if a function needs to change the variables?

The values could be returned by the function.

However, the passing and returning of variables can become quite cumbersome.

A better solution is to use a class.

A class provides a way to group together functions and variables that are closely related so they can inter act with each other.

A class also makes it possible to work with multiple objects of the same type.

For example, each land parcel is likely to have the same attributes.

The concept of grouping together functions and variables related to a particular type of data is called object-oriented programming (OOP).

Classes are the container for these related functions and variables.

Classes make it possible to create objects that have specific properties as defined by these functions and variables.

You have seen several ArcPy classes, such as the env class, which can be used to access and set environment settings, and the Result class, which defines the properties and methods of result objects that are returned by geoprocessing tools.

Being able to create your own classes in Python, however, opens up many new possibilities.

To make a class in Python, you use the keyword class.

Take a look at a simple example:

|  |
| --- |
|  |

The class keyword is used to create a Python class called Person.

The class contains two method definitions-these are like function definitions, except that they are written inside a class statement and are therefore referred to as methods.

The self parameter refers to the object itself.

You can call it whatever you like, but it is almost always called "self" by convention.

A class can be thought of as a blueprint.

It describes how to make some- thing and you can create many instances from this blueprint.

Each object created from a class is called an instance of the class.

Creating an instance of a class is sometimes referred to as instantiating the class.

Next, you will see how this class can be used.

|  |
| --- |
|  |

Using an assignment statement creates an instance of the Person class.

Creating this instance looks like calling a function.

Once an instance is created, you can use the properties and methods of the class, as follows:

|  |
| --- |
|  |

Running this code prints the following:

|  |
| --- |
|  |

This example is relatively simple, but it illustrates some key concepts.

First, a class can be created using the class keyword.

Second, properties of the class are defined as methods-they look like functions but are called methods when they are defined inside a class.

Third, a class can contain multiple properties and methods.

Now return to the example of a parcel of land.

You want to create a class called parcel that has two properties (land-use type and total assessed value) and a procedure (calculating tax) associated with it.

For the purpose of this example, assume the property tax is calculated as follows:

For single-family residential, tax = 0.05 \* value

For multifamily residential, tax = 0.04 \* value

For all other land uses, tax = 0 .02 \* value.

Creating the Parcel class is coded as follows:

|  |
| --- |
|  |

The class called \_\_Parcel\_\_ is created using the class keyword.

The class contains two methods: \_\_init\_\_ and assessment.

The \_\_init\_\_ method is a special method reserved for initializing objects inside a class-that is, constructing objects before they can be used.

This method has three arguments: self, landuse, and va1ue .

When the class is called, however, the first argument (self) is not used.

The argument self represents the object and is provided for implicitly by calling the class.

The assessment method is w here the actual calculation occurs.

Next, take a look at how to use this class.

The following code create s an instance of the parcel object:

|  |
| --- |
|  |

With the instance created, you can use its object properties and methods, as follows:

|  |
| --- |
|  |

Running this code prints:

|  |
| --- |
|  |

You can create multiple instances of this object.

In a typical scenario, you could run the property tax calculation for every parcel in a database, creating a new instance for each parcel.

In many cases, you m ay want to use the class in more than one script.

This can be accomplished by putting it in a module- that is, creating a separate script with the definition of the class, which can then be called from another script.

This is analogous to creating a separate script for a function, which can be called from other scripts, as described earlier in this chapter.

In this example, the script containing the class is called parceJclass.py and is as follows:

|  |
| --- |
|  |

In this example, the script that uses the class is called parceltax.py and is as follows.

|  |
| --- |
|  |

## 12.6 Working with packages

When you have a number of different functions and classes, it often makes sense to put them in separate modules (scripts).

As your collection of modules grows, you can consider grouping them into packages.

A package is essentially another type of module, but it can contain other modules as well.

A regular modu1 e is stored as a .py file, but a package is a fo1der (or directory).

Technically speaking, a package is a folder with a file called ”\_\_init\_\_.py” in it.

This file defines the attributes and methods of the package.

It doesn’t actually need to define anything; it can just be an empty file, but it must exist.

If \_\_init\_\_.py does not exist, the directory is just a directory, and not a package, and it can't be imported.

The \_\_init\_\_.py file makes it possib1e to import a package as a module.

For example, to import ArcPy, you use the import arcpy statement, but there is no script file called "arcpy.py.

" However, there is an arcpy folder with a &le called "\_\_init\_\_.py”.

For examp1e, if you had a package you wanted to call "mytools," you wou1d need to have a fo1der called "mytools" and inside this folder wou1d need to be a file called "\_\_init\_\_.py".

The structure of a package called mytoo1s with two modules (analysis and model) wou1d 100k as follows:

|  |
| --- |
|  |

To use the package, your code would 100k as follows:

|  |
| --- |
|  |

You may wonder what a site package is.

A site package is a locally installed package that is available to all users using that computer.

The "site" is the local computer.

What makes a package a site package has to do with how it is installed, and not its actual contents.

During the installation of a site package, the path to the package is added to the PYTHONPATH variable As a result, the package can be directly imported without first having to add the path.

Python has a number of built-in site packages, which can be found in the Lib\site-packages folder.

You will see PythonWin listed there.

Parts of the PythonWin editor are installed as a site package, although the actual application is a file called PythonWin.exe, which is located outside the package.

Another commonly used site package is NumPy, which is used to manipulate large arrays of data.

ArcPy is referred to as a site package because a typical ArcGIS installation includes both ArcPy and Python, and the folder where ArcPy is located is automatically recognized by Python through the Desktop 10.1.pth file located in the Lib\site-packages folder.

Where exactly is ArcPy installed?

Typically the location is C:\Program Files\ArcGIS\Desktop10.1\arcpy.

*Note: Although the preceding path is the default location for the installation of ArcGIS, it can vary depending on the operating system and the user-defined selections during installation.*

*However if you can find the ArcGIS installation folder, you will also be able to find the copy folder.*

When you explore the contents of this folder, you will find a subfolder called arcpy (which gives ArcPy its name), as shown in the figure, and which contains a file called \_\_init\_\_.py, which makes it a Python package, in addition to many files whose names sound familiar (analysis.py, cartography.py, geocod.py and more).

Normally, you should never work with these files directly, but for educational purposes, it is OK to examine them.

Just don’t make any changes! As part of the installation of ArcPy, the path C:\Program Files\ArcGIS\D esktopl0.l \arcpy is added to the PYTHONPATH environment variable in Windows and you can start using ArcPy immediately.

## Points to remember

# Chapter 13 Creating custom tools

## 13.1 Introduction

This chapter describes the process of turning a Python script into a tool.

Tools make it possible to integrate your scripts in ArcGIS.

Tools can be run from ArcToolbox, can be used within a model, and can be called by other scripts.

Tools have a tool dialog box, which typically contains the parameters that are passed to the script.

Developing tools is relatively easy and greatly enhances the experience of using a script.

Tool dialog boxes reduce user error because parameters can be specified using drop-down lists, check boxes, combo boxes, and other mechanisms.

This provides substantial control of user input, greatly reducing the need to write a lot of error-checking code.

Creating tools also makes it easier to share scripts with others.

## 13.2 Why create your own tools?

Many ArcGIS workflows consist of a sequence of operations in which the output of one tool becomes the input of another tool.

ModelBuilder and scripting can be used to automatically run these tools in a sequence.

Any model created and saved using ModelBuilder is a tool because it is located in a toolbox (.tbx file) or a geodatabase.

A model, therefore, is always run from within an ArcGIS for Desktop application, such as ArcMap or ArcCatalog.

A Python script (.py file), however, can be run in two ways:

1. As stand-lone script.

This means the script is run from the operating system or from within a Python editor, such as PythonWin.

For a script to use geoprocessing tools, ArcGIS for Desktop needs to be installed and licensed, but no ArcGIS for Desktop application needs to be open for the script to run.

For example, you can schedule a script to run at a prescribed time directly from the operating system.

1. As tool within ArcGIS.

This means the script is turned into a tool to be run from within an ArcGIS for Desktop application.

Such a tool is like any other tool: it is located in a toolbox, can be run from a tool dialog box, and can be called from other scripts, models, and tools.

There are a number of advantages to using tools instead of stand-alone scripts.

* A tool includes a tool dialog box, which makes it easier for users to enter the parameters using built-in validation and error checking.
* A tool becomes an integral part of geoprocessing.

This makes it possible to access the tool from the Catalog and Search windows in ArcMap.

It also makes it possible to use the tool in ModelBuilder and in the Python window, and to call it from another script.

* A tool is fully integrated with the application it was called from.

This means any environment settings are passed from the application, such as ArcMap, to the tool.

* The use of tools makes it possible to write messages to the Results window
* Documentation can be provided for tools, which can be accessed like the documentation for system tools.
* Sharing a tool makes it easier to share the functionality of a script with others.
* A well-designed tool means a user requires no knowledge of Python to use the tool’s functionality.

## 13.3 Steps to creating a tool

A tool is created using the following steps:

1. Create a Python script and save it as a .py file

2. Create a custom toolbox (.tbx file) where the tool can be stored.

3. Add a tool to the custom toolbox using the Add Script wizard

4. Modify the script with input and output variables so that it is seamlessly integrated into the geoprocessing framework.

You can create a new custom toolbox in ArcCatalog or in the Catalog window inside another ArcGIS for Desktop application.

Navigate to Toolboxes, right click My Toolboxes, and click New > Toolbox.

Give the toolbox a name.

*Note: Do not click New > Python Toolbox, because a Python toolbox is created entirely in Python and not in GIS.*

*A new generic toolbox is all that is needed at this point.*

This section describes how to create a script tool using a custom toolbox.

Although ArcGIS 10.1 has introduced Python toolboxes, which support additional capabilities to custom toolbox, it is more convenient to use a custom toolbox when starting to create your first script tools.

Your empty custom toolbox can now be added to ArcToolbox.

You can drag it from the Catalog window into ArcToolbox or right lick inside ArcToolbox and click Add Toolbox, which allows you to browse for a tool box in any folder.

A toolbox, which consists of a single .tbx file, can be located anywhere on your computer.

The folder My Toolboxes is one logic allocation to organize custom toolboxes, but they can also be located in any folder w here datasets and other files for a particular project are organized-for example, C:\EsriPress\Python\Data\MyCoolTools.tbx.

Custom toolboxes can also be located inside a geodatabase.

Like other elements, a toolbox inside a geo database no longer has a file extension-for example, C:\EsriPress\Python\Data\study.gdb\MyCoolTools.

To create a tool, in ArcToolbox, right-click a custom toolbox and click Add > Script.

Write access to the toolbox is needed to be able to add a new tool.

As a result, you cannot add tools to any of the system toolboxes in ArcToolbox.

The example script that comes next illustrates how to create a tool.

This script creates a list of all the feature classes in a workspace and copies these feature classes to an existing file geodatabase, as follows:

This script is written as a stand-alone script.

Both the current workspace and the file geodatabase are hard-coded in the script.

Although the script will run correctly, it will require modification to be useful as a tool.

To start the Add Script wizard, in ArcToolbox, right click a custom toolbox and click Add > Script.

This brings up the first pane1 of the wizard, as shown in the figure.

The first panel of the wizard is used to specify the tool name, label, description, and style sheet as follows:

The name of a tool is used when you want to run a tool from Python.

The name cannot contain any spaces.

The label of the tool is the display name in ArcToolbox.

The label name can have spaces.

Consider the example of the Get Count tool.

In ArcToolbox, the tool appears with its label, Get Count (with a space), but for the tool to be called from Python, its name, GetCount (without a space), is used.

The description is an optional held to provide a customized description.

The text is automatically used to provide the contents of the Help panel on a tool dialog box

An optional style sheet can be selected.

If none is selected, the default style sheet is used.

Style sheets are used to control the properties of items on a tool dialog box.

A style sheet provides style and layout information.

All the system tools in ArcToolbox use the default style sheet.

Typically you want your custom tools to look just like the system tools so the default style sheet is usually sufficient.

Optionally, the "Store relative path names" check box can be selected.

When it is selected, relative paths are used instead of absolute paths to reference the location of the script file in relation to the location of the custom toolbox (.tbx) file.

Only the path to the script file can be stored as a relative path; paths within the script itself will not be converted.

If you are going to share the tool with others, it is a good idea to select this check box.

Optionally, the "Always run in foreground" check box can be selected.

This will ensure the script is run using foreground processing, even if background processing has been enabled under Geoprocessing Options.

(Background processing allows you to continue to work in the ArcG1S for Desktop application while the tool is running.)

Some scripts require foreground processing-for example, mapping scripts, which us the CURRENT keyword to obtain the active map document in ArcMap.

For other scrip selecting foreground or background processing is a matter of preference

In the second panel of the Add Script wizard, you can set the following:

The complete path of the Python script file to be run.

You can browse to an existing file or type the path of a file.

If you specify a script file that does not exist, you will be prompted to create a new (empty) script file.

Alternatively, the script file field can be left blank and added later.

The "Show command window when executing script" check box, which is cleared by default.

When the box is selected, an additional window appears during tool execution to show messages that are not part of the regular geoprocessing messages but are written to the standard output for Python.

For example, the Python print statement w rites to the standard output, such as the Interactive Window in PythonWin.

If such statements are part of your script, the messages would not appear unless this box is selected.

Scripts that are referenced by a tool should normally use geoprocessing messaging and not write to the standard output.

So the box remains clear unless you have very specific needs for viewing messages.

The "Run Python script in process" check box, which is selected by default.

Python scripts run faster if they are run "in process, " so typically you’ll want this option selected.

Running in process requires that Python modules in your script be designed to run in process, which is the case for standard modules such as OS, math, and string.

Nonstandard modules from third parties may not be designed for this process, which can result in performance issues.

So if you are using third-party modules in your script, which appears to result in unexpected problems, you can try running the script out of process instead.

The third panel of the Add Script wizard is used to specify the tool parameters.

By default, no parameters are listed, but most tools need at least one input parameter and one output parameter.

The top half of the panel allows you to create parameters and the bottom half allows you to set the properties for each of these parameters.

Setting parameters is covered in detail later in this chapter, including parameter properties.

To complete the Add Script wizard, click the Finish button.

Completing this wizard adds a tool to your custom toolbox.

All the settings in the Add Script wizard can be modified by right-clicking the tool and clicking Properties.

The Properties dialog box of the tool includes tabs for General, Source, and Parameters, which correspond to the three panels of the Add Script wizard.

Two additional tabs of the dialog box are Validation and Help.

These tabs are revisited later in this chapter.

Your new tool can be accessed just 1ike a regular tool.

Right lick the tool and click Open or simply doub1e-click the tool in its toolbox.

The tool dialog box seems empty at this point because no parameters were set in the third panel of the Add Script wizard.

The Help panel on the right shows the tool's description, but there are no parameters yet.

Clicking OK runs the tool – that is, runs the script - but without parameters, it does not provide the user with any control over its execution.

One of the most critica1 steps in creating too1s is to create input and output parameters and add them to the tool’s dialog box.

Remember, however, that the script was written to run as a stand-alone tool.

Setting parameters therefore requires modifying the code in the script so it can receive the parameters set by the tool dialog box.

## 13.4 Editing tool code

When you create a tool, you typically have to make changes to the script so that the tool dialog box and the script can interact seamlessly.

When testing a tool, you will alternate between running the tool and editing the script until the tool works as desired.

You can 1eave the Python editor open whi1e you do it.

You can open a script from within the Python editor, but there is a1so a shortcut in ArcGIS.

Right-click the too1 in the too1box and click Edit This will open the Python script in a Python editor such as IDLE or PythonWin.

The Python editor that is used is determined by the geoprocessing options.

To change these settings, on the menu bar in an ArcGIS for Desktop app li cation, click Geoprocessing > Geoprocessing Options.

To select PythonWin as the editor, browse to the location of the PythonWin app1ication.

Typically, this application is located at C:\Python27\ArcGIS10.l\Lib\ site-packages\PythonWin\PythonWin.exe, but this may vary depending on how Python was installed on your computer.

Once you set a specific editor, any script file opened from within ArcGIS will open in this editor.

Now that a tool is created and you can access its code, it is time to take a closer look at parameters.

## 13.5 Exploring tool parameters

As you have seen in several earlier chapters, all geoprocessing tools have parameters.

Parameter values are set on the tool dialog box.

In a stand-alone script, parameters are typically set within the script unless user input is expected.

For tools, parameters can be set using the tool dialog box.

When a tool runs, the parameter values from the tool dialog box are passed to the script.

The script reads these values and uses them in the code.

Creating and exposing parameters requires the following steps:.

Including code in the script to receive the parameter values

Setting up the parameters in the tool’s properties

Next, you will try this out by using one of the built in tools, the MultipleRingBuffer tool.

The tool dialog box is shown in the figure.

The Multiple Ring Buffer tool has seven parameters total, three of which are optional.

The Parameters tab on the tool properties dialog box lists the same seven parameters, in the same order, as the tool dialog box.

It also shows the data type of each parameter.

For example, the input features consist of a feature layer, and the buffer unit is a string value.

Note: Because the MultipleRingBuffer tool is a built-in tool, you can see the list of parameters but you cannot see or edit the parameter properties.

And because the parameters are read-only, the entire panel appears dimmed.

If you want to read more about the parameters you can copy the tool to a custom toolbox, which provides read/write permission to access it properties.

Once a user specifies the parameter values on the Multiple Ring Buffer tool dialog box, the tool can be run.

Once the tool is run, the user-specified parameter values will be passed to the script.

Take a look at the script’s code to see how these parameter values are received by the script.

The example MultiRingBuffer.py script, which is shown in the PythonWin editor in the figure, includes the import of a number of modules and some introductory comments, and then contains a section of code where the parameter values are received.

The tool’s parameters are received by the script using the GetParameterAsText and the GetParameter functions.

This script uses the ArcGISscripting module from version 9.3, but the concept is the same.

The syntax of the GetParameterAsText function is

The only argument of this function is an index number on the tool's dialog box, which indicates the numeric position of the parameter.

The parameters set on the tool dialog box are sent to the script as a list and the GetParameterAsText function assigns these parameter values to variables in the script.

The two figures (see below and facing page) show how each parameter on the tool dialog box matches the code index number-for example, Input Features is (0) , Output Feature class is (1) , and Distances is (2).

The GetParameterAsText function receives parameters as a text string, even if the parameter on the tool dialog box is a different data type.

Numerical values, Boolean values, and other data types are all converted to strings and additional code is included to correctly interpret these strings.

For example, the code of the Outside Polygons Only parameter is as follows:

The Outside Polygons Only parameter on the tool dialog box is a Boolean value of True or False.

These values are converted to strings, and as a result, the conditional statement uses the string value "true" instead of the Boolean value True.

The Distances parameter on the tool dialog b ox is received by the script using the GetParameterfunction.

This is because the parameter consists of a list of values (doubles) instead of a single value.

The GetParameter function reads this list as an object.

Note: An alternative to using the GetParameterAsText and GetParameter functions is to use sys.argv, or system arguments.

The use of sys.argv has certain limitations, including that it accepts a limited number of characters.

Using the GetParameterAsText and GetParameter functions is therefore preferred.

Prior to ArcGIS 9.2, these functions would work only for tools referencing a script, and stand-alone scripts could use only sys.argv.

The latter is therefore relatively common in older scripts.

The index number for sys.argv starts at 1, so sys.argv[1] is equivalent to GetParameterAsText (0).

Every tool parameter has an associated data type.

One of the benefits of data types is that the tool dialog box will not send values to the script unless they are the correct data type.

User entries for parameters are checked against the parameter data types before they are sent to the script.

This is one advantage of using tools over stand-alone scripts because the script does not have to check for invalid parameters.

The data types of the parameters of the Multiple Ring Buffer tool include a feature layer, a feature class, a double, three strings, and a Boolean.

Many more data types are possible for the parameters of a custom too l, from an address locator to a Z domain Data types of parameters should be selected carefully because they control the interaction between the tool dialog box and the script .

After parameters are assigned a data type, the tool dialog box uses this data type to check the parameter value.

For example, if you enter a path to an element of a different data type, the tool dialog b ox will generate an error.

In the example in the two figures, the Input Features parameter is a feature layer (upper right) , so typing the path for a raster, such as C:\Data\dem , will generate an error (lower right) an d prevent the tool from running.

This built-in error- checking mechanism prevents from using incorrect parameters to run a tool.

When the tool runs, the dialog box has already validated the parameter Input Features as a feature layer, and no additional code is needed in the script to verify it.

The data type property is also used to browse through folders for data.

Only data that matches the parameter’s data type will be shown.

This prevents the entering of incorrect paths to data.

## 13.6 Setting tool parameters

Tool parameters can be set in the Add Script wizard when cleating the tool.

They can also be edited after the tool has been created by accessing the tool's properties dialog box.

Set ting parameters is the same, no matter which method is used.

A parameter is added by placing the cursor in the first empty cell in the Display Name column, under the Parameters tab, and typing a name for the parameter, which is displayed on the tool dialog box.

Next, the data type is specified by selecting from an extensive drop down list.

You can add more parameters by repeating these steps.

Once multiple parameters are created, the order can be changed by selecting one and using the arrow keys to move the row up or down.

Each parameter has a number of properties, as shown in the bottom half of the dialog box.

When each parameter is created, its properties are set to default values based on the parameter's data type.

Some of the key parameters are discussed as follows.

A complete description of all the parameters can be found in ArcGIS Desktop Help, under the topic "Setting script tool parameters."

### Type

There are three choices for Type: Required, Optional, and Derived Required means that a parameter value needs to be specified for a tool to run.

Optional means that a value is not required for a script to run.

Typically, this means that a default value is specified.

Derived parameters are used for output parameters only and do not appear on the tool dialog box.

Derived parameters are used in a number of cases, including the following:

When a tool outputs a single value instead of a dataset.

Such a single value is often referred to as a scalar.

When a tool creates outputs using information from other parameters.

When a tool modifies an input without creating a new output.

All tools should have outputs so that the tool can be used in a model.

Sometimes the only way to accomplish this is by using derived parameters.

Examples of tools with derived parameters include the Get Count tool and the Add Field tool.

The input parameter of the Get Count tool is a feature class, table, layer, or raster, and the output is a count of the number of rows.

This count is a scalar variable and is returned as a result object.

It comprises an output parameter and is a derived parameter that does not appear on the tool dialog box.

The Add Field tool adds a new held to an input table.

The input table is a required parameter, and so is the name of the new held.

The rest of the parameters are optional, as shown in the figure.

|  |
| --- |
| Note: Running the Get Count tool as a single tool from ArcToolbox is not common.  Although the count is printed to the Results window, this tool is typically used within a model or a script where the output is used as the input to another step.  The Get Count tool is also commonly used in conditional statements.  For example, a particular procedure can be stopped if the count of rows is zero(0). |

When the tool runs, a new field is added based on the input parameters.

The output of the tool is the modified table or feature class.

Because this table or feature class is an input parameter, there is no need to specify the output on the tool dialog box.

The output of the tool is therefore specified as a derived parameter.

An easy way to visualize it is by using the tool in a model, as shown in the figure.

When you inspect the properties of the input feature class and the output, you will notice that they reference the exact same feature class.

In fact, you cannot specify or change the name of the output (you can change the display name in the model but not the underlying data).

### Direction

The Direction property defines whether the parameter is an input of the tool or an output of the too1.

For derived parameters, the parameter direction will automatically be set to output.

Every tool should have output parameters.

This makes it possible to use the tool in ModelBuilder.

Although technically a script can run without output parameters, for ModelBuilder to work, every tool needs an output so it can be used as the input to another tool in the model.

### MultiValue

Some tool parameters consist of a list of values rather than a single value.

When the MultiValue property is set to No, only a single value can be used.

When the MultiValue property is set to Yes, a list of values can be used.

Multivalue parameters are quite common for built-in geoprocessing tools.

For example, the Union tool uses a list of input feature classes.

The Union tool uses the default multivalue parameter control, which is simply a list of inputs that can be added, removed, and reordered.

The second type of multivalue parameter is a list of check boxes.

This is commonly used for fields, as illustrated by the Delete Field too1.

Check boxes can also be used if a Value List filter is applied, which is discussed later in this section.

Multivalue parameters are passed to the script as a delimited string, with the individual list elements separated by semicolons (;).

The Python split method can be used to create a list of the elements from the string.

The syntax is as follows:

As an alternative, you can use GetParameter to obtain a ValueTable object instead of a string.

In a ValueTable, the values are stored in a virtual table of rows and columns.

ValTable objects are specially designed for multivalue parameters.

This means that when writing the script, you need to be aware of the parameter types being passed to the script from the tool dialog box.

### Default

The default value of a parameter is the contents of the parameter when the script tool’s dialog box is open.

If no default value is specified, the parameter value will be blank on the tool’s dialog box.

If a default value is specified, the Environment property will be disabled.

### Environment

A default value for a parameter can also be specified using the Environment property.

By right-clicking the cell next to Environment, you can choose the name of the environment setting.

Once this property is set, the default value is obtained from the environment settings of the geoprocessing frame work.

If the Environment property is specified, the default property will be ignored-so you need to specify one or the other.

### Filter

The Filter property allows you to limit the values of dataset types to be entered for a parameter.

There are a number of filter types, and the type depends on the data type of the parameter.

The different filter types are Value List, Range, Feature Class, File, Field, and Workspace.

For most data types, there is only one filter type.

For example, if the data type of a parameter is set to Feature Class, the only possible filter type is Feature Class.

The only exceptions are the Long and Doubles data types, which have Value List and Range as possible filter types.

Many data types have no filter type at all.

The different filter types provide specific control of which parameters are valid inputs.

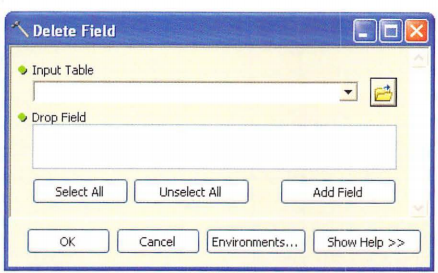
Carefully setting the filter type will improve the robustness of the tool.

The different filter types are discussed in more detail in ArcGIS Desktop Help, under the topic "Setting script tool parameters."

### Obtained from

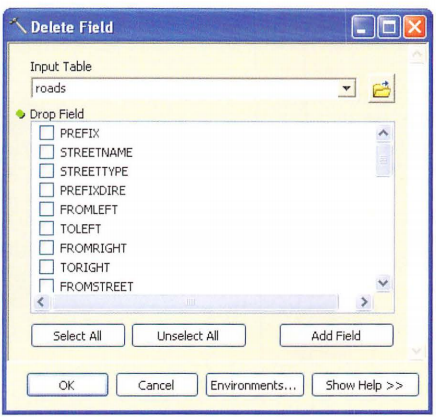
In many cases, a tool parameter is closely related to another one.

For example, consider the Delete Field tool.



The first parameter is an input table, and the second parameter, Drop Field, is a list of fields.

The list of fields is populated only when the input table is selected, as shown in the figure.



This dependency of a parameter on another parameter in the same tool is controlled using the "Obtained from" property.

In the example of the Delete Field tool, the Obtained from property of the Drop Field parameter is set to the input table.

A second reason to use the Obtained from property is to work with derived output parameters.

For example, when an input parameter is modified by a tool, the Obtained from property of the derived output parameter is set to the input parameter.

1n the case of the Delete Field tool, the Obtained from property of the output parameter is set to the input table.

*Note: Remember that the derived output parameter is not visible on the tool dialog box.*

### Symbology

By default, the output of a geoprocessing tool is added to the ArcMap tab le of contents.

This behavior can be set on the Geoprocessing Options dialog box under Display/Temporary Data.

The symbology of a layer added in this way follows the same rules as when data is added using the Add Data option in ArcMap-in other words, there is no customized symbology.

The Symbology property can be set to a custom layer file (.lyr).

This option is available only for outputs where layer files make sense, such as feature classes, rasters, TINs, and the like.

The parameter type can be required or derived, but the parameter direction has to be set to output for the Symbology property to be accessible.

Notice that setting the Symbology property does not control w h ether the output will be added to the table of contents, because this is controlled by Geoprocessing Options in ArcMap.

## 13.7 Examining an example script tool

The following example illustrates the steps to convert a stand-alone script to a script too1.

The following stand-alone script was introduced at the beginning of the chapter.

The script creates a list of all the shapefiles in a workspace and copies them to a geodatabase.

For the purpose of this example, the script is located in the C:\Sharedscripts folder.

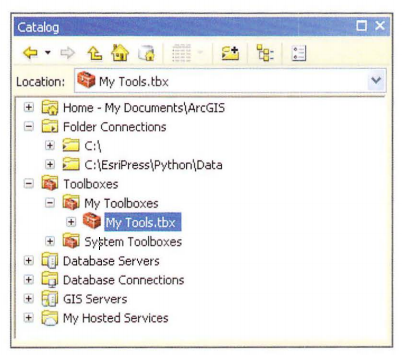
The script is as follows:

Two workspaces hard-coded into the script have to be modified to parameters using the GetParameterAsText function.

The revised script (without comments) is as follows:

A custom toolbox is created in the My Toolboxes folder.

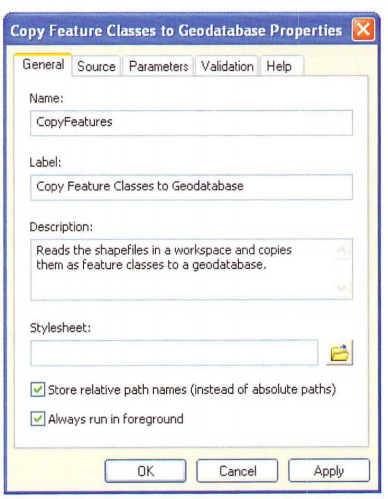
This empty toolbox can be dragged to ArcToolbox.



To create a new script tool, right-click the custom too1box and click Add > Script.

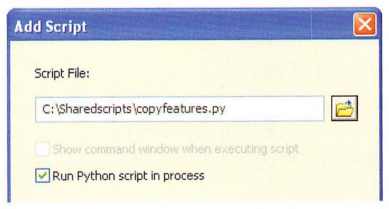
In the first panel on the Add Script dialog box, specify the name, label, and description of the script tool.

Select the check boxes for storing relative path names and foreground processing.



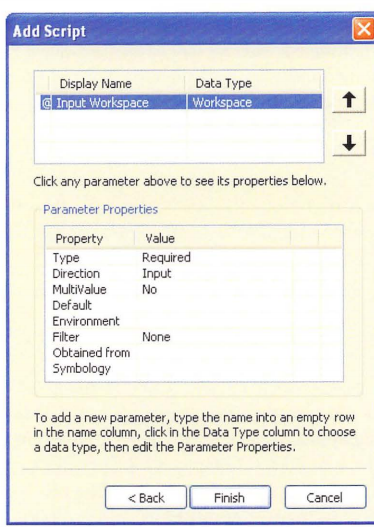
In the second pane1 on the Add Script wizard, browse to the location of the script file-in this case, C: \Shared scripts\copyfeatures.py.

Leave the other settings to their defaults.



In the third panel on the Add Script wizard, you can specify the parameters.

For the first parameter, enter the name Input Workspace and select Workspace for Data Type.

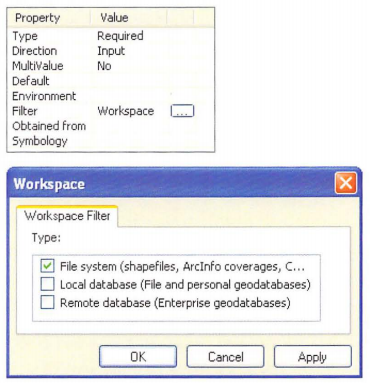


Under Parameter Properties, click the Filter property.

Under Value, select Workspace.

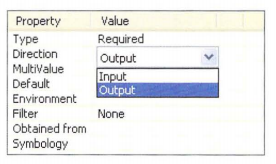
This brings up the Work space filter.

Under Type, clear the check boxes for "Local database" and "Remote database."



For the second parameter, enter the name Output Workspace and select Workspace for Data type.

Under Parameter Properties, set Direction to Output.



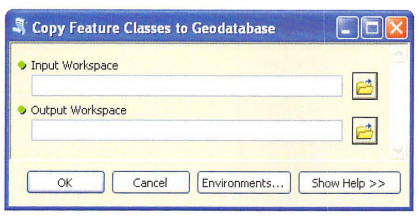
In the Add Script wizard, click Finish to complete creating the script tool.

After the parameters are added and the properties are set, the tool dialog box should look like the example in the figure.

The tool is now ready to run.

The tool copies the shape files to a geodatabase, and the output of the tool is the geodatabase workspace.

The copied feature classes themselves will not be added to the ArcMap table of contents - this is by design because the number of feature classes could conceivably be very large.



## 13.8 Customizing tool behavior

Once a tool’s parameters are specified, you can add custom behavior. Examples of custom behavior include the following:

Certain parameters may need to be enabled or disabled based on the values contained in other parameters.

Some parameters may benefit from having a default value specified based on the values in other parameters

If there are a lot of parameters, it may be more effective to organize parameters into different categories.

Warning and error messages may need to be customized.

Tool behaviors can be set on the Validation tab on the Script Properties dialog box.

In the Validation panel, you can use Python code that uses a Python class called ToolVa1idator.

The Too1Va1idator class controls how the tool dialog box is changed based on user input.

It is also used to describe the output data the tool produces, which is important for using tools in ModelBuilder.

The Too1Va1idator class was introduced in ArcG1S 9.3, providing more possibilities for creating robust tools.

A detailed description of customizing tool behavior is not provided here.

Details on the ToolVa1idator class can be found in ArcG1S Desktop Help, under the topic "Customizing script tool behavior."

## 13.9 Working with messages

One of the advantages of running a script as a tool is writing messages that appear on the progress dialog box and in the Results window.

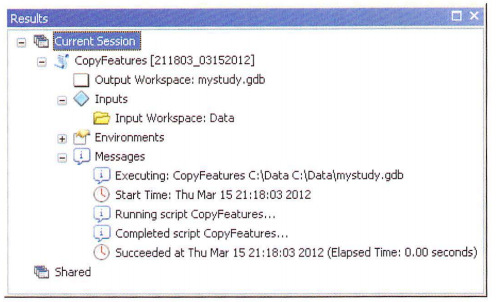
Tools and scripts that call a tool also have access to these messages.

When scripts are run as stand-alone scripts, messages are printed only to the Interactive Window-there is no progress dialog box and no Results window where messages can be retrieved later.

There is also no sharing of messages between stand-alone scripts.

However, because script tools work like any other tool, they automatically print messages to the Results window.

For example, when the Copy Feature Classes to Geodatabase tool is run, it prints very simple messages that indicate when the script is running and note when it is completed.



Since this is just the bare bones, there are a number of ArcPy functions for writing messages.

These include the following:

Since this is just the bare bones, there are a number of ArcPy functions for writing messages.

These include the following:

AddMessage -for general information messages (severity = 0)

AddWarning -for warning messages (severity = 1)

AddError-for error messages (severity = 2)

AddIDMessage-for both warning and error messages

AddReturnMessage-for all messages, independent of severity

The AddReturnMessage function can be used to retrieve all messages returned from a previously run tool, regardless of severity.

The original severity of the geoprocessing messages is preserved-for example, an error message is printed as an error message.

Some of the other message functions create a custom message.

For example, the AddError and AddMessage functions are used in the following code to print custom messages to the Results window based on the result of a particular tool:

In the case of a feature class without any features, running this code will produce an error message, as shown in the figure.

Calling the AddError function a1so results in a Failed to execute message.

However, it does not add an exception, and the code will keep running after the AddError call.

When the AddWarning function is used instead, it results in a warning message, but the script will finish running

Another level of control can be accomplished using the AddIDMessage function.

This function makes it possible to use system messages within a tool.

The syntax of the function is

The message type can be set to Error, Informative , or Warning .

The message 1D number indicates the specifìc Esri system message.

Depending on the message, additional arguments may be necessary.

1n the following example code, an error message, with the message 1D number 12, is produced if the output feature class already exists:

The syntax of error message 12 is

This message has one argument, which in this case is the name of a feature class.

A complete list of tool error and warning messages can be found in ArcGIS Desktop Help, under Geoprocessing > Tool errors and warnings .

A small sample of these error messages in Help is shown in the figure.

## 13.10 Handling messages for stand-alone scripts and tools

Python scripts can be run as stand-alone scripts or as tools.

Messaging works a bit differently for each one.

However, a script can be designed to handle both scenarios.

For a stand-alone script, there is no way to view messages, and they have to be printed to the interactive interpreter.

For a tool, functions such as AddError are used instead of a print statement to ensure messages appear in the geoprocessing environment including the Resu1ts window.

Standard practice is to write a message-handling routine that writes messages to both the interactive interpreter and the geoprocessing environment, using a print statement for the former and ArcPy functions such as AddError, AddWarning, and AddMessage for the latter.

## 13.11 Customizing tool progress information

When a tool runs, information on its progress can take several forms.

By default, the geoprocessing framework in ArcGIS uses background processing.

This means you can continue to use an application whi1e the geoprocessing operations run in the background.

During background processing, a progress bar appears at the bottom of the document on the ArcGIS for Desktop application status bar, as shown in the figure.

When the geoprocessing operation is completed, a pop-up notification appears in the notification area, at the far right of the taskbar.

Background processing can be enabled or disabled on the Geoprocessing Options dialog box.

The slider under Background Processing, as shown in the figure, can be moved to control how long the pop-Up window appears at the end of background processing.

When background processing is disabled, foreground processing is enabled.

During foreground processing, a progress dialog box appears.

A progress dialog box includes a progressor, which consists of a horizontal bar indicating the progress of the tool, and a message area, which consists of a complete list of geoprocessing messages, as shown in the figure. This is the same list of messages that appears in the Results window.

Whether a tool runs as background or foreground processing can be controlled at the application level using the Geoprocessing Options dialog box.

For a script tool, background or foreground processing can also be controlled as part of the script tool properties, as described in section 13.3.

The appearance of the progress dialog box, which appears during foreground processing, can be controlled using the ArcPy progressor functions.

These functions also have an effect on the Results window.

The ArcPy progressor functions include the following:

SetProgressor - sets the type of progressor

SetProgressorLabel -changes the label of the progressor

SetProgressorPosition-moves the step progressor by an increment

ResetProgressor-resets the progressor

There are two types of progressors: default and step.

1n the default type, the progressor moves back and forth continuously but doesn't provide a clear indication of how much progress is being made.

The lab el above the progressor provides information on the current geoprocessing operation.

In the step progressor, the percentage completed is shown.

This can be useful when processing large datasets.

The type of progressor is set using the SetProgressor function.

This function establishes a progressor object, which allows progress information to be passed to a progress dialog box.

The appearance of the progress dialog box can be controlled using either the default progressor or the step progressor.

The syntax of this function is

The progressor type is either default or step.

The message is the progressor label that appears at the beginning of the tool's execution.

The three remaining parameters are for step progressors only and indicate the start value, the end value, and the step interval.

In a typical step progressor, the start value would be set to 0, the end value to however many steps are completed in the geoprocessing operations, and the step interval to 1.

The SetProgressorLabel function is used to update the label of the progressor, which is typically a unique string specific to each step.

The SetProgressorPosition function is used to move the step progressor by an increment based on the percentage of features completed.

These functions are commonly used in combination so that the label is updated at every increment.

Once tool execution is completed, the progressor can be reset to its original position using the ResetProgressor function.

The following Copy Feature Classes to Geodatabase script uses a custom progress dialog box.

A step progressor is used, and the number of steps is derived from the number of feature classes in the list.

In the for loop, the label is changed to the name of the shapefile being copied, and after the shapefile is copied, the step progressor is m oved by an increment.

The script is as follows:

Running the script brings up a progress dialog box with a progressor that shows the percentage completed.

This percentage is calculated from the step progressor parameters-that is, the steps are automatically converted to a percentage as they are completed.

Another consideration is the number of steps being used in a step progressor.

In many script, it is not known in advance how many feature classes, field, or records will need to be processed.

A script that uses a search cursor, for example, may iterate over millions of records.

If each iteration were one step, the progress dialog box would need to be updated millions of times, which could severely reduce performance.

It may therefore be necessary to include a section in the script that determines the number of iterations (feature classes, rows, or whatever the case may be) and then determines an appropriate number of steps based on the number of iterations.

## 13.12 Running a script in process

Python scripts can be run in process or out of process.

Running a script in process means that a script can be run as is without ArcGIS having to start another process or program.

Running a script out of process means that ArcGIS has to start another process for the script to run.

When another process is started, it takes time for both programs to run, which reduces performance.

Other performance issues also arise from message community cation between the two processes.

In general, therefore, it is recommended that a Python script be run in process so that it will run faster.

Specifying that a tool should run in process or out of process can be done on the Source tab on the tool properties dialog box.

By default, this option is selected - that is, scripts are run in process.

It should be noted that this option applies only to scripts written in Python.

Although running tools in process is recommended to improve performance, there are certain cases when running tools in process can cause problems.

For example, some nonstandard modules from third parties may not have the necessary logic to run in process.

If you are using third-party modules and are experiencing problems, running the tool out of process may be the solution.

Standard Python libraries have modu1es that have the necessary logic and can be run in process without difficulty.

## Points to remember

Although Python scripts can be run as stand-alone scripts outside of ArcGIS for Desktop applications, there are many benefits to running scripts as tools.

Tools allow a closer integration of scripts in the ArcGIS geoprocessing framework.

For example, tools that reference a script can be used in ModelBuilder the same way as any other tool.

A tool can be created in any custom toolbox and reference a single Python file (.py) that is called when the tool is run.

For tools to be usable and effective, tool parameters need to be created.

This includes setting parameters in the tool properties, as well as including code in the script to receive the parameter values.

Tool parameters define what the tool dialog box looks like.

Effective tools have carefully designed parameters.

Each parameter has a data type, such as feature class, table, value, field, or other.

The parameter properties provide detailed control of the allowable inputs for each parameter.

This ensures that the parameters passed from the tool dialog box to the script are as expected.

All tools should have outputs so that the tool can be used in ModelBuilder.

Sometimes the only way to accomplish this is to use derived parameters, which do not appear on the tool dialog box.

Tool behavior can be further customized using a ToolValidator class.

Various message functions can be used to write what will appear on the progress dialog box and in the Results window.

The appearance of the progress dialog box can be controlled using a number of different functions to change the progressor.

Running scripts in process is recommended to improve performance.

# Chapter 14 Sharing tools

## 14.1 Introduction

The ArcGIS geoprocessing framework is designed to facilitate the sharing of tools.

Custom toolboxes can be added to ArcToolbox and integrated into regular workflows.

Toolboxes can contain any number of tools, consisting of both model tools and script tools.

Tools can therefore be shared by distributing a toolbox file (.tbx) that contains the accompanying Python scripts (.py).

However, there are a number of obstacles to sharing script tools.

One of the principal obstacles is that the resources available to the creator of the script will likely be different from those available to the user of the shared script tools.

This includes map documents, toolboxes, scripts, layer files, and any other files used by the tools.

Another obstacle is the organization of these resources on a local computer or network.

Paths present a fairly persistent problem when sharing tools.

This chapter provides guidelines on how to distribute script tools, including how to structure toolboxes, scripts, documentation, and other files that are commonly distributed with shared tools.

To help overcome some of these obstacles, ArcGIS 10.1 has introduced geoprocessing packages as a convenient way to distribute shared tools

## 14.2 Choosing a method for distributing tools

Tools that are developed to share with others can vary from the simple to the complex.

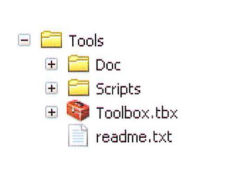
The simplest case is a single toolbox file with one or more tools inside the toolbox and no additional files.

In a more typical example, a shared tool could consist of a toolbox file, one or more scripts that are used in script tools, and some documentation.

A more complex example could contain a toolbox file, several scripts, documentation, compiled Help files, and sample data.

A recommended folder structure for these files is presented later in this chapter.

A relatively typical folder structure might look like the example in the figure.



One of the most common ways to share tools is simply to make all the files available in their original folder structure.

This typically involves the use of a file compression utility to create a single ZIP file of the folders and their contents.

This ZIP file can then be posted online or emailed.

The recipient can download the file and extract the contents to access the individual folders and files.

The toolbox IS then added to ArcToolbox to access the tools.

There are two other ways to share tools.

If users have access to the same local area network, the folder containing the tools can be copied to a folder that is accessible to all users.

A toolbox can be added directly from the network, and no files need to be copied to the user's computer.

A second alternative is to publish the toolbox as a geoprocessing service using ArcGIS for Server, which can then be accessed by anyone with an internet connection.

The method to use depends 1argely on the relationship between the creator of the tool and the intended users, as well as the software and the skills of the user.

For example, if tools are developed primarily for use by others within the same organization, making tools availab1e on a loca1 area network may be the most efficient.

To make tools avai1able to a broad community of users, the use of a ZIP file is likely the most convenient.

A number of other considerations will influence how to share tools, including w here the input and output data is located and what products and extensions the tools require.

In the ZIP fi1e method, for example, any tool data also has to be packaged with the tool because a typical user will not have access to any of the data on the network.

## 14.3 Handling licensing issues

Tools distributed using the ZIP file method will run on a user's computer, which may not have the necessary products or licenses to run the tools.

Scripts shou1d therefore include 10gic to check for the necessary product 1eve1s (ArcGIS for Desktop Basic, ArcGIS for Desktop Standard, or ArcG1S for Desktop Advanced) and extension 1icenses (ArcGIS 3D Ana1yst, ArcG1S Spatial Ana1yst, and more).

Even if a user has the necessary extensions installed, a license may not have been obtained for the current session.

In this scenario, the tool will stop with an error message.

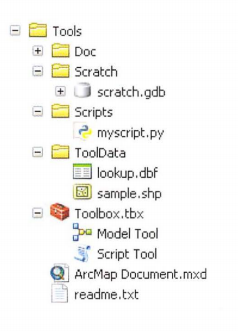
To facilitate the use of shared tools, the necessary product level and extensions need to be described in the tool’s documentation.

Working with licenses is covered in chapter 5.

## 14.4 Using a standard folder structure for sharing tools

A standard folder structure, like the example in the figure, is recommended by Esri for easy sharing of geoprocessing tools.

There is no requirement to use this specific structure, but it provides a good starting point.



|  |
| --- |
| >>> TIP  Python scripts, by default, are not shown in ArcCatalog, but they can be added as a file type (.py) by going to the menu bar and clicking Customize > ArcCatalog Options > File Types. |

The Tools folder contains one or more toolboxes (.tbx files), which contain the tools, including model tools and script tools.

Toolboxes can also be placed inside a geodatabase, but a .tbx file directly under the Tools folder is easier to find.

Tools should have the "Store relative path names" check box selected more on paths later in this chapter.

Tool documentation should clearly state what product level and extensions are required for the tools to run.

A README file (readme.txt) is often included that explains how the tool works and contains special instructions on how the tool needs to be installed, contact information for the tool's creators, and the like.

Distributing an ArcMap document (.mxd) with the tools is optional but may be helpful if example datasets are part of the shared tool.

The ToolData folder may contain sample datasets that a user can work with to learn about the functionality of a tool before trying it out on the use 1" s own data.

The tools may also require certain data as part of tool execution, such as lookup tables, also included in this folder.

The Scripts folder contains the Python scripts used in the script tools.

Other related files may include script libraries, dynamic-link libraries (DLL), and executable files, such as .exe and .bat (batch) files.

Scripts can also be embedded directly into a toolbox, in which case there are no separate script files.

This is not very common, however, since in many cases the purpose of sharing the tools is for users to use and learn from the scripts and contribute to their continued improvement.

Many model tools and script tools use a workspace, and a default file geodatabase for scratch data (scratch.gdb) can be provided in the Scratch folder.

Tool documentation is provided in the Doc folder.

Documentation can consist of a Microsoft Word document (.doc or .docx) or PDF file (.pdf) that provides instructions, external Help compiled HTML files (.chm) that are referenced by tools or toolboxes, and XML style sheets that replace the default tool dialog boxes and Help dialog boxes.

Experienced Python coders are likely to open the actual scripts and learn from both the comments and the code in the scripts.

Many other users, however, may never look at the scripts and instead use only the tool dialog boxes.

Good documentation will ensure that users get the most out of a tool and understand what it will accomplish, as well as its limitations, without having to open the actual scripts.

## 14.5 Working with paths

Paths are an integral part of working with data and tools.

When tools are shared, paths become particularly important, because without proper documentation of where files are located, the tools will not run.

If you have worked with ArcGIS to create map documents or tools, you are probably familiar with absolute and relative paths.

Absolute paths are also referred to as full paths.

They start with a drive letter, followed by a colon (:), and then the folder and file name-for example, C:\Data\streams.shp.

Relative paths refer to a location that is relative to a current folder.

Relative paths make use of special symbols-a single dot (.) and a double dot (..).

A single dot represents the folder you are working in, and a double dot represents the parent folder.

Although technically correct, this convention for navigating folders is not very practical because you cannot type relative paths in ArcGIS or Python scripts.

Still, it is important to understand the concept of relative paths and what it means in respect to manipulating data in ArcGIS.

Consider the following example with two shape files located in the C:\alldata\shape files\final folder: boundary.shp and locations.shp.

Relative to each other, there is no need to know the path other than the base names-that is, the file names.

Now consider an example where you want to run a tool that uses the shape files locations.shp and flood zone.shp.

These files are located in two different folders, and therefore their relative paths are final\locations.shp and project\floodzone.shp.

The higher-level folders-that is, all data\shape files-are not needed to locate one file relative to the other.

You have likely worked with relative paths when saving map documents (.mxd files).

To avoid lost data connections when folders are moved or renamed, the data source options in map document properties can be set to relative paths.

When map documents are saved using relative paths, ArcMap converts absolute paths to relative paths based on the location where the .mxd file is stored.

For example, consider a map document that is saved with relative paths enabled and that is stored as follows:

If the shape file locations.shp is added to this map document, the absolute path is converted to the following relative path:

Relative to the location of the map document (Map.mxd), locations.shp is located in the parent folder of the map document-that is, C:\alldata (hence, the single dot followed by the double dot)-and under the subfolder shape files\final.

The name and location of the parent folder itself is not needed to access the shapefile and is therefore not part of the relative path.

|  |
| --- |
| >>> TIP  Don't worry too much about the notation for relative paths since you can't type this notation in ArcGIS or Python anyway |

The use of relative paths makes it possible to move or rename folders.

For example, if the alldata folder were renamed "data," all paths in the ArcMap document would remain intact.

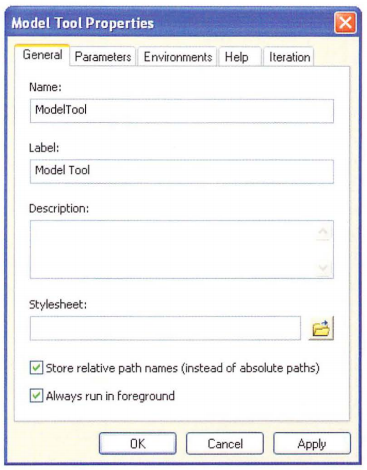
Similarly, if the drive letter were modified from C to E, all paths would also remain intact.

One limitation of relative paths is that they cannot span multiple disk drives.

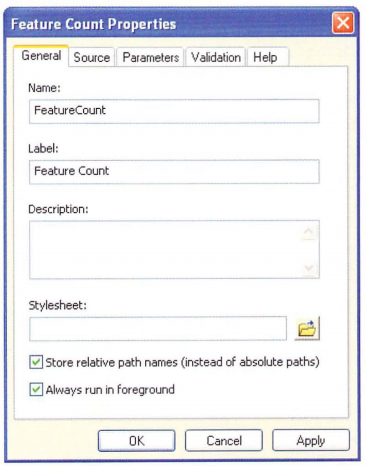
If some files are located on the C drive and some on the E drive, only absolute paths will preserve the correct locations of all files.

Similar to working with map documents, absolute paths and relative paths can be used in model tools.

Relative paths for models can be enabled on the model properties dialog box.



In script tools, relative paths are enabled in the Add Script wizard when creating a script tool, or on the script tool properties dialog box for existing tools.



Relative paths are relative to the current folder where the toolbox file is located.

When relative paths are enabled, it applies to the script files, datasets used for the default value properties, files referenced in the tool documentation, layer files used for the symbology properties, compiled Help files, and style sheets.

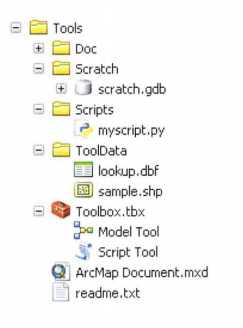
It is important to recognize that paths within the script are not converted because ArcGIS has no reliable way to examine and modify the script code.

Therefore, if a script uses absolute paths, they are not converted to relative paths when relative paths are enabled for the script tool.

|  |
| --- |
| >>> TIP  In general, Python code needs to be written so that files can be found relative to a known location, which is typically the location of the script itself |

After this review of working with paths, it is worthwhile to revisit relative paths in the context of sharing tools.

For the purpose of this discussion, the same example folder structure discussed earlier in this chapter, as shown in the figure, will be used.



To share tools, relative paths have to be enabled in the script tool properties.

1n this example, the script tool will reference a script in the Scripts folder.

It may also reference tool documentation in the Doc folder.

The script itself may reference data in the ToolData folder.

These references will continue to be valid when the script tool is shared with another user, as long as the standard folder structure is maintained.

If the toolbox file (Toolbox.tbx) containing the script tool were moved to a different location separate from the other folders and files, the script files called by the script tool would not be found and the script would not work.

The tool dialog box would open, but upon tool execution, the following error message would appear: “Script associated with this tool does not exist.”

Therefore, for a script tool to work correctly, the folder structure must be maintained.

## 14.6 Finding data and workspaces

In general, it is best to avoid hard-coding paths into your script if it is going to be shared with others as a script tool.

Instead, the paths are derived from the parameters on the tool dialog box and these paths are passed to the script.

The script reads these parameters using GetParameterAsText and related functions.

Sometimes, however, it is necessary to hard-code paths to a particular location.

For example, an existing layer file may be necessary to set the symbology for an output parameter.

Or a p articular tool may require the use of a lookup table.

Depending on the nature of the information, it may already be incorporated into the script (for example, a lookup table can be coded as a Python dictionary), but this may not always be possible.

Some data files may therefore be necessary for the tool to run, even though they are not provided as parameters for the user.

Instead, these data files are provided by the author of the script and distributed as part of the shared tool.

Following the suggested folder structure presented earlier in this chapter, these files would be placed in the ToolData folder, making it possible for the data files to be found relative to the location of the script.

The path of the script can be found using the following code :

Or:

Running this code results in a string with the comp1ete path of the script, but without the name of the script itself.

If the data files necessary for the script to run are located in the ToolData folder, per the suggested folder structure, the Python modu1e os.path can be used to create a path to the data.

The folder structure used thus far can serve as an example.

The Tools folder contains the shared tools, including the toolbox, the script, and the data files.

Relative paths are enab1ed for the script tool, so the Tools folder can be moved, or even renamed, and the script will still work.

The script referenced by the script tool is located in the Scripts folder.

The script needs a file called lookup.dbf, located in the ToolData folder, to run.

The file name can be hard-coded into the script because the author of the script is a1so the author of the lookup.dbf file and the creator of the ToolData folder.

However, the absolute path shou1d not be hard-coded into the script but the relative path used instead: ToolData\lookup.dbf.

This will make it possible for the Tools folder to be moved to any location without the user of the script tool being limited to the absolute path originally used by the author of the script.

The code that references the lookup.dbf file in the script is as follows:

Notice that two e1ements are hard-coded into the script: the actual file name of the tool data (lookup.dbf) and the folder where the tooldata is located (ToolData).

These are both created by the author of the to01and can therefore be hard-coded into the script because they do not depend on user input.

A similar approach can be used to reference the location of a scratch workspace.

Scratch workspaces are very common in models, and they can also be used in scripts.

Using the same example folder structure as before, the script to set the scratch workspace is as follows:

When a scratch workspace is set, a few other considerations should be kept in mind, as follows:

Write permission to the workspace is required

A scratch workspace can be set as part of the environment settings.

If this is the case, a script should use this workspace because it most likely has been intentionally set by the user.

So the preceding example code would typically be preceded by an if statement that evaluates whether a scratch workspace has been specified in the environment settings that are being passed to the script

Using the current workspace as a scratch workspace is possible, but it can cause problems.

First, the current workspace can become cluttered if the script generates a lot of output.

Second, cleaning up the results afterward can be cumbersome because it may be difficult to separate intermediate data, which can be deleted, from the final results.

Third, if the current workspace is a remote database on a server, it can cause performance issues.

Finally, instead of the name of an (empty) scratch geodatabase being hard coded into a script, the CreateScratchName function can be used to create a unique dataset in the scratch workspace.

## 14.7 Creating a geoprocessing package

The approach for distributing shared tools as described so far is quite robust but also somewhat cumbersome.

It typically requires that you manually consolidate data, tools, and supporting files into a single folder.

As an alternative, ArcGIS 10.1 has introduced geoprocessing packages, which are a more convenient way to distribute all the tools and files related to geoprocessing workflows.

This section describes what a geoprocessing package is and how to create it.

A geoprocessing package is a single compressed file with a .gpk extenion.

This single file contains all the fi les necessary to run a particular geoprocessing workflow, including custom tools, input datasets, and other supporting files.

This file can be posted online, e-mailed, or shared through a local area network.

Although this sounds similar to the use of a ZIP file, as described earlier in this chapter, geoprocessing packages are created very differently and have additional functionality.

A geoprocessing package is created from one or more results in the Results window, which have been created by successfully running one or more geoprocessing tools.

A basic workflow to create and share a geoprocessing package is as follows:

1. Add data (and custom tools if needed) to an ArcGIS for Desktop application.

2. Create a geoprocessing workflow by running one or more tools.

3. In the Results window, select one or more results, right-click the selection, and click Share As > Geoprocessing Package.

4. Complete the entries on the Geoprocessing Package dialog box, which includes options for sharing, for adding additional results or files, and for sharing datasets or only the schema

5. Share the resulting .gpk file.

A recipient of the geoprocessing package can open the contents in an ArcG1S for Desktop application to examine the datasets and workflows used.

A single .gpk file contains all the resources needed to rerun the geoprocessing workflow, including tools, layers, and other files.

Tools can include system tools as well as custom tools.

So, if a geoprocessing result was created using a script tool, this script tool and the underlying .py files necessary for the tool to run would all be included in the geoprocessing package.

The single greatest benefit of using geoprocessing packages is that all the necessary resources are automatically combined in a single file, no matter where they are located.

There is no need to manually consolidate all the resources into a single folder as required by the more traditiona1approach using ZIP files.

Geoprocessing packages are described in great detail in ArcGIS Desktop Help on the Contents tab, under Geoprocessing > Sharing geoprocessing workflows

## 14.8 Embedding scripts and password protecting tools

The most common way to share script tools is to reference the Python script file in the script too1 properties and to provide the script file separately, typically in the Scripts folder

This allows users to clearly see which scripts are being used, and the scripts can be opened to view the code.

Scripts can also be embedded in a toolbox.

The code is then contained within the toolbox, and a separate script file is no longer needed.

This approach can make it easier to manage and share tools.

To import a script, right-click the script tool and click import Script.

Once a script is imported into a tool, the toolbox can be shared without including the script fi1es.

1n other words, just sharing the .tbx file is sufficient, and no separate .py files need to be provided for the script tool to run.

When a script is imported, however, the original script file is not delete it is simply copied and embedded in the toolbox.

Embedding scripts does not mean they can no longer be viewed or edited.

Say, for example, you have imported a script and shared a toolbox with another user.

The recipient can right-click the script tool and click Export to obtain a copy of the original script.

Once exported, the script can be viewed and edited the same way as any other script in a shared tool.

Although embedding scripts is a useful way to reduce the number of files to manage and share, it can lead to some confusion.

For example, some script tools use multiple scripts-for example, a script that is referenced by the script tool and additional scripts that are called by the first script.

Embed ding multiple scripts can be confusing to users because it becomes less transparent how the scripts work.

1n addition, embedding scripts was introduced in ArcG1S 10, so many users are probably more familiar with seeing a .tbx file and one or more .py files than having them embedded.

One very good reason to embed your scripts is to create password protection.

Regular script files cannot be password protected.

If you share your tools including individual .py files, any user can open these scripts with a Python editor or a text editor.

Users can modify the code or copy it for use in their own scripts.

This is, in fact, one of the reasons why working with Python is so appealing.

If, for some reason, you need password protection, you can right-click the script tool and click Set Password-this works only if you have previously imported a script.

Setting a password does not affect execution of the script tool, but any attempt to view the script or export the script will prompt the use of a password.

## 14.9 Documenting tools

Good documentation is important when sharing tools.

Documentation includes background information on how the tool was developed as well as specifics on how the tool works.

Documentation can also be used to explain specif1 c concepts that m ay be new to other users

There are a number of ways to provide documentation for a tool within ArcGIS, as follows:

A brief, text-only description can be provided on the tool properties dialog box.

This description will appear as the default text in the Help panel on the tool dialog box.

A more detailed description can be created by editing the Description page of a tool.

This description is used in multiple locations, including the tool reference page, the Help panel on the tool dialog box, and the Help panel in the Python window.

A style sheet can be used, providing additional control of the look and feel of the tool dialog box.

A compiled Help file can be created and referenced to appear as the tool reference page.

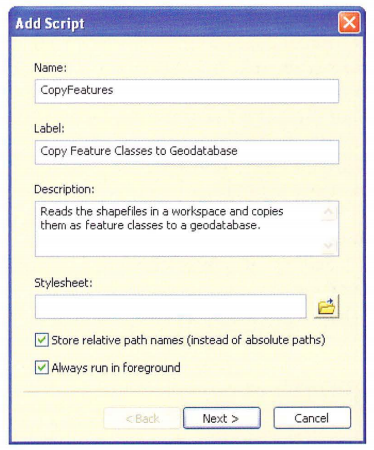
Following is a more detailed explanation of some of the ways to provide documentation for a tool.

### Providing a brief description

The simplest form of tool documentation in ArcGIS is accomplished by providing a brief description in the Description box on the tool properties dialog box.

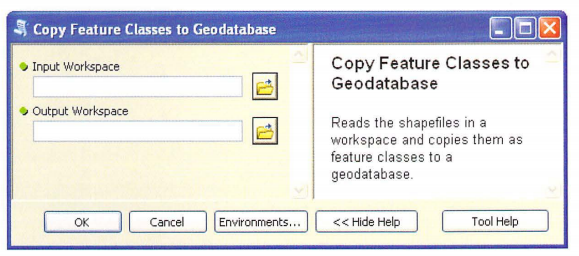
This description can be created in the Add Script wizard or by accessing the script tool properties.

By default, this description will appear in the Help panel on the tool dialog box.



This type of tool documentation is a bit limited because it allows for text only, and it provides a description of the tool only, and not individual parameters.

However, it is a good place to start.



This type of tool documentation is a bit limited because it allows for text only, and it provides a description of the tool only, and not individual parameters.

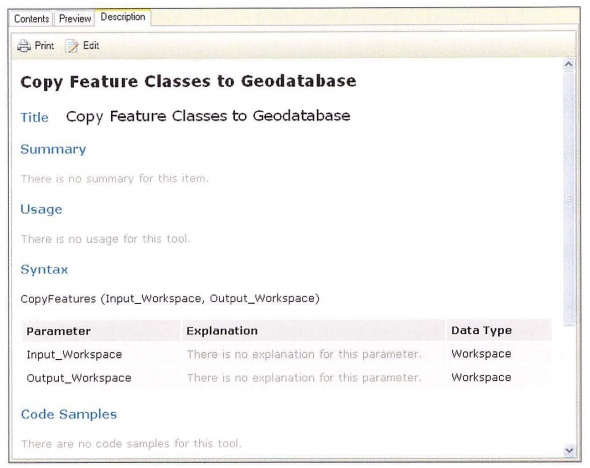
However, it is a good place to start.

### Editing the Description page

By defau1t, when a script tool is created, a Description page is a1so created, which is populated with the to01's basic syntax.

This page can be viewed in ArcCatalog on the Description tab, where you would normally review metadata.

For example, the default Description page of the Copy Feature Classes to Geodatabase script tool created in chapter 13 is shown in the figure.



The default description is rather rudimentary and provides 1itt1e in addition to what the tool dialog box does.

However, the contents of the Description page can be modified in ArcCatalog by clicking the Edit button under the Description tab, in the same way that metadata can be edited.



From here, you can edit the title and summary, provide examples of the usage, enter detai1s of the syntax in a tab1e, and add code samples.

You can a1so 10ad a thumbnai1 graphic that illustrates how the tool works.

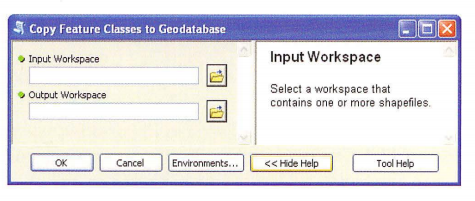
Tags are used to identify the subject or content of your tool.

The documentation you provide for Item Description is used by the Search window to find your tool.

Notice that there are many entries in the description that are not re1evant to tool usage because they are mostly intended for documenting the metadata for datasets.

Once you modify the Description page, this information is used on the tool dialog box.

For example, if you enter a description for individua1 parameters, it will appear on the tool dialog box He1p pane1 for that parameter.



The Description page also becomes the default page that loads when the Help option for a particular tool is clicked, unless a different file has been specified.

### Using style sheets

Style sheets are used to control the properties of the tool dialog box.

A style sheet provides style and layout information, including fonts, alignment, and margins.

The default style sheet is typically sufficient and is found in the C:\Program Files\ArcGIS\Desktop10.1\ArcToolbox\Stylesheets folder.

The default style sheet is applied automatically for any new tool dialog box, but you can create your own if desired.

## Using a compiled Help file

A script tool can reference a compiled Help file (.chm).

This file is used when viewing the tool Help page.

Compiled Help files are similar to HTML files but are compiled to create a single, self-contained package of documentation.

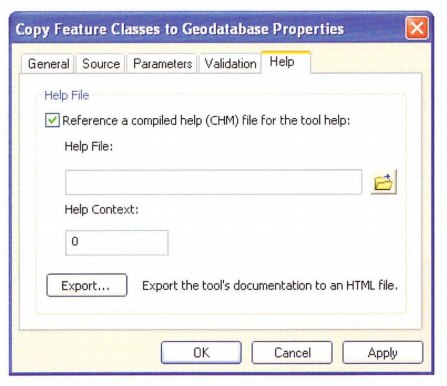
Compiled Help files are a proprietary format for Help files developed by Microsoft for use in the Windows operating system.

Creating a compiled Help file requires the use of the Microsoft Help Software Development Kit (SDK).

If you have a compiled Help file, you can reference it on the Help tab on the tool properties dialog box.

Optionally, you can provide the Help context by specifying an HTML topic ID.

The Help context controls which topic in the .chm file will be displayed.



If you have already created detailed documentation on the tool's Description page, you can export this documentation to an HTML file.

This is an option on the Help tab on the tool properties dialog box.

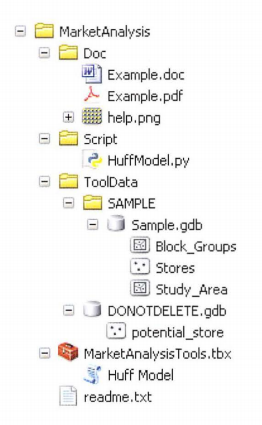
You can then use this HTML file when creating a compiled Help file.

## 14.10 Example tool: Market analysis

This section looks at an example tool to review the organization of the files that are part of the tool , as well as the tool's documentation.

The example tool is a market analysis tool based on the Huff Model, which is used to estimate sales potential for store locations based on demographic data.

The tool was created by Drew Flater as an example of a more advanced script tool and is posted on the ArcGIS for Desktop Resource Center.



The tool’s organization closely follows the suggested folder structure.

There is a single .tbx file with a single script tool.

The actual script file, HuffModel.py, is located in the Script folder.

The ToolData folder contains a sample dataset as well as some data that the tool needs for execution.

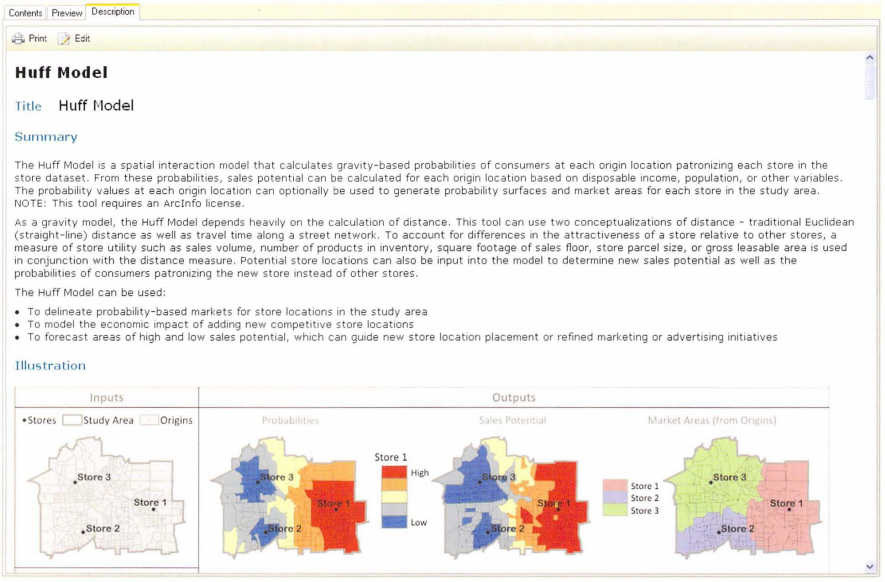
The Doc folder contains a .doc or .docx file and a .pdf file that describe the sample data and includes a brief tutorial on using the tool.

Detailed tool documentation can be found on the Description tab in ArcCatalog.

The same documentation can be accessed by viewing the tool's Help file, although the formatting will be slightly different depending on the HTML browser.

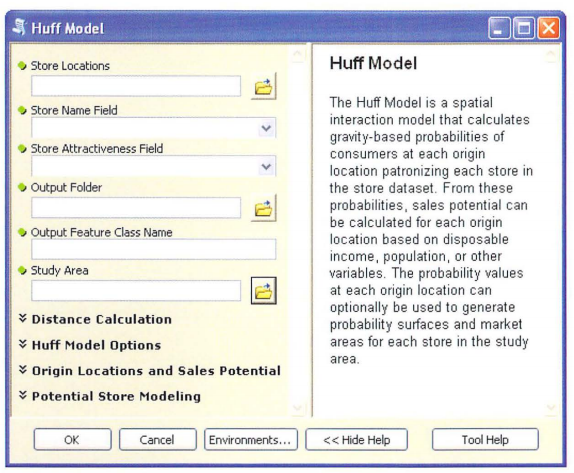
The text and formatting of this documentation is store d in the toolbox.

The image in the documentation is referenced in the tool's description, but the actual image file, help.png, is located in the Doc folder.



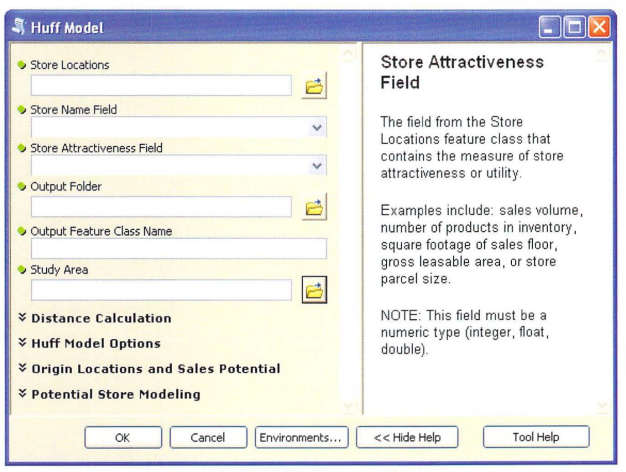
The tool dialog box itself contains a large number of required and optional parameters.

When the tool dialog box is open, a brief description of the tool appears in the Help panel

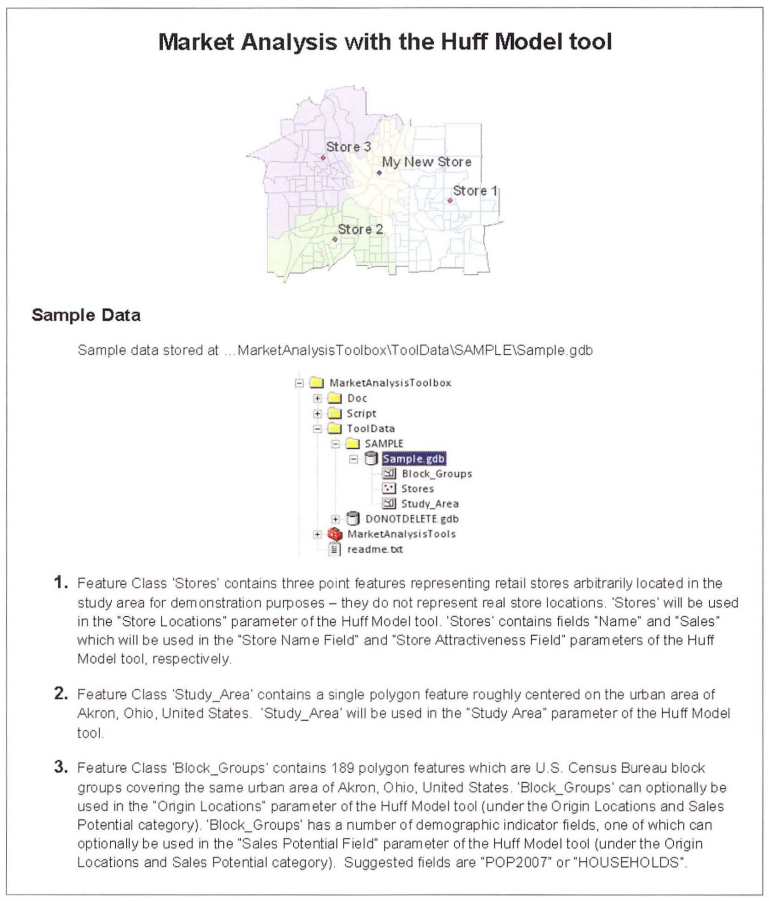


The content of the Help panel changes when the cursor is placed inside one of the parameter boxes-in this case, the Store Attractiveness field.

This information is derived from the tool’s description.



Separate from ArcGIS, tool documentation is provided in the form of a tutorial fil e on disk that describes the sample data and explains how to run the tool.



Finally, the script its e lf contains documentation in the form of comments.

This example script is relatively advanced-it has 748 lines of code, and the tool dialog box has 18 parameters.

But the tool’s standard organization and documentation make it relatively easy to use.

## Points to remember

The ArcGIS geoprocessing framework is designed to facilitate the sharing of tools. Custom toolboxes can be added to ArcToolbox and integrated into regular workflows. Toolboxes can contain any number of tools, consisting of both model tools and script tools. Tools can therefore be shared by distributing a toolbox file (.tbx) that contains the accompanying Python scripts (.py) and any other resources needed to run the tools

To ensure custom tools work properly, the resources needed to run the tools should be made available in a standard folder structure. This includes folders for scripts, data, and documentation.

Absolute paths work only when f1 1es are not moved and folders are not renamed. To share tools, relative paths should be enabled for each script too1. Relative paths are relative to the current folder, which for scripts is where the toolbox is located. Relative paths cannot span multiple drives.

Geoprocessing packages provide an alternative way to distribute script tools. A geoprocessing package is a single, compressed f1 le with a .gpk extension that contains all the f1 1es necessary to run a particular geoprocessing workflow, including custom tools, input datasets, and other supporting f1 les

Shared tools can be documented in various ways , including editing the Description page in ArcCatalog, using style sheets, and referencing compiled Help files.