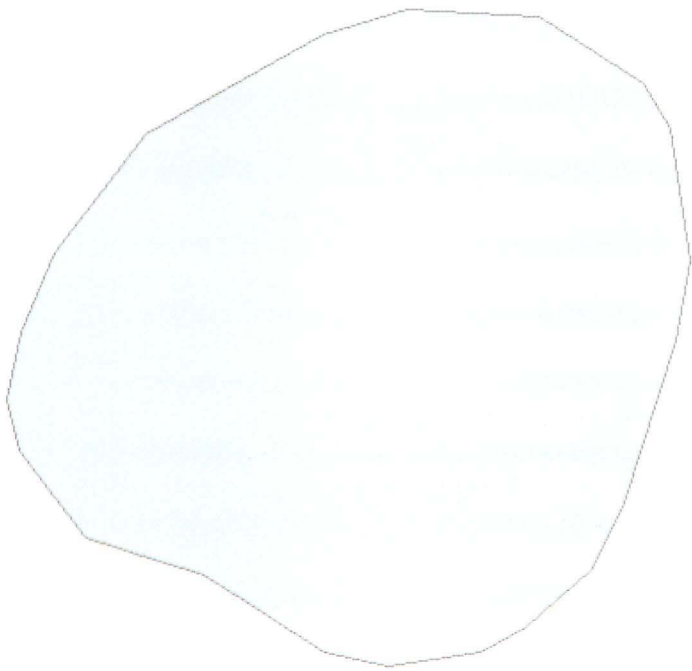
Finally, the script needs to iterate over the lines of the input text file and create a point object for every line. The result is a single array with 21 point objects. The completed script is as follows:

```
import arcpy, fileinput, os
from arcpy import env
env.workspace = "C:/Data"
infile = "C:/Data/points.txt"
fc = "newpoly.shp"
arcpy.CreateFeatureclass_management("C:/Data", fc, "Polygon")
cursor = arcpy.da.InsertCursor(fc, ["SHAPE@"])
array = arcpy.Array()
point = arcpy.Point()
for line in fileinput.input(infile):
    point.ID, point.X, point.Y = line.split()
    line_array.add(point)
polygon = arcpy.Polygon(array)
cursor.insertRow([polygon])
fileinput.close()
del cur
```

The result of the script is a new shapefile called `newpoly.shp` with a single polygon feature, as shown in the figure.



The example script is still relatively simple because it created only a single polygon with no other attributes. However, it illustrates the concept of using the `Point` and `Array` classes to create new geometry objects.

8.7 Using cursors to set the spatial reference

The spatial reference for a feature class describes the coordinate system, the spatial domain, and the precision. The spatial reference is typically set when the feature class is created. However, since specifying a spatial reference is not required, it results in an unknown coordinate system when none is specified. In this case, the Define Projection tool can be used to record the coordinate system information for the feature class.

A spatial reference applies to all the features in a feature class. By default, the spatial reference of the geometry of an object returned from a cursor is, therefore, the same as the spatial reference of the feature class opened by the cursor. In certain circumstances, however, you may be working with geometries that have a different spatial reference from the feature class—for example, if you have a feature class in a state plane coordinate system and you want to insert new features using a text file that has universal transverse Mercator (UTM) coordinates. In this case, you could set the spatial reference on the update or insert cursor to ensure proper conversions. You would open an insert cursor on the feature class and set the spatial reference of the cursor to UTM, thus declaring that the geometries to be inserted need to be converted from UTM to state plane.

You can also set the spatial reference of a search cursor. Specifying a spatial reference that is different from the spatial reference of the feature class results in geometries that are converted into the spatial reference of the cursor.

Consider the example of using a point feature class in state plane coordinates and writing a script that exports the x,y coordinate pairs of the point objects in decimal degrees. The `SearchCursor` function is used to establish a read-only cursor on the state plane coordinates of the feature class, but the spatial reference of this cursor is set to the desired geographic coordinate system, in decimal degrees. This is accomplished using the following code:

```
import arcpy
fc = "C:/Data/hospitals.shp"
prjfile = "C:/projections/GCS_NAD_1983.prj"
spatialref = arcpy.SpatialReference(prjfile)
cursor = arcpy.da.SearchCursor(fc, ["SHAPE@"], "", spatialref)
```

Next, an output file is created, using the `open` function. This opens the file in writing mode (`"w"`) so that new lines of text can be written to it, as follows:

```
output = open("result.txt", "w")
```

The next step is to iterate over the rows, create a geometry object for each row, and write the x,y coordinates to the output file using the `write` method. This part of the code is as follows:

```
for row in cursor:
    point = row[0]
    output.write(str(point.X) + " " + str(point.Y) + "\n")
```

The coordinates are written as decimal degrees in a string, with a space (`" "`) separating the coordinates of each pair, and with a line break (`"\n"`) for each point object. The very last step is to close the output file using the `close` method. The complete code is as follows:

```
import arcpy
from arcpy import env
env.workspace = "C:/Data"
fc = "hospitals.shp"
prjfile = "C:/Projections/GCS_NAD_1983.prj"
spatialref = arcpy.SpatialReference(prjfile)
cursor = arcpy.da.SearchCursor(fc, ["SHAPE@"], "", spatialref)
output = open("result.txt", "w")
for row in cursor:
    point = row[0]
    output.write(str(point.X) + " " + str(point.Y) + "\n")
output.close()
```

In this example, the spatial reference is set by using an existing projection file (.prj), but it can also be obtained from an existing feature class.

> ***Note:*** *Although setting the spatial reference on the search cursor can be used to convert geometries from one coordinate system to another, it is not very robust. In particular, setting any necessary datum transformation is not automatically taken into account and would need to be set in the script as part of the environment settings.*

8.8 Using geometry objects to work with geoprocessing tools

Inputs for geoprocessing tools often consist of feature classes. Sometimes, however, these feature classes do not yet exist and need to be created from geometry information. In this case, you can create a new feature class, populate the feature class using cursors, and then use the feature class in geoprocessing tools. This can become cumbersome, however. As an alternative, geometry objects can be used instead of both input and output feature classes to make geoprocessing simpler.

For example, the following code creates a list of geometry objects from a list of coordinates, and then uses the geometry objects as input to the Buffer tool, as follows:

```
import arcpy
from arcpy import env
env.workspace = "C:/Data"
coordlist = [[17.0, 20.0], [125.0, 32.0], [4.0, 87.0]]
pointlist = []
for x, y in coordlist:
    point = arcpy.Point(x,y)
    pointgeometry = arcpy.PointGeometry(point)
    pointlist.append(pointgeometry)
arcpy.Buffer_analysis(pointlist, "buffer.shp", "10 METERS")
```

In the example code, the geometry objects are created as a list of point objects. First, an empty list is created using `pointlist = []`. In the `for` loop, the list of coordinate pairs is used to create point objects using the `Point` class. These point objects are then used by the `PointGeometry` class to create geometry objects, which are appended to the list. The list becomes the input for the Buffer tool. An alternative would be to first create a feature class based on the list of coordinates, but if this feature class is not necessary for anything else, the use of geometry objects will result in more efficient code.

Geometry objects can also be created directly as the output of geoprocessing tools. For example, the following code uses an empty geometry

object as the output of the Copy Features tool, and the result is a list of geometry objects, as follows:

```
import arcpy
fc = "C:/Data/roads.shp"
geolist = arcpy.CopyFeatures_management(fc, arcpy.Geometry())
length = 0
for geometry in geolist:
    length += geometry.length
print "Total length: " + length
```

The use of geometry objects can improve the efficiency of your code because it allows you to avoid the steps of having to create temporary feature classes and use a cursor to read through all the features.

Points to remember

- The geometry object provides access to a number of properties, including length and area. Geometry tokens can be used as shortcuts to specific geometry properties.
- Individual vertices of geometry objects are stored as an array of point objects (or an array containing multiple arrays of point objects in the case of multipart features).
- New features can be created or updated using the insert and update cursors. A script can define a feature by creating point objects, populating their properties, and placing the point objects in an array. This new array can then be used to set the geometry of a feature.
- The spatial reference can be set on cursors to work with geometries in a coordinate system that is different from that of the feature class.
- Geometry objects can be used instead of feature classes as inputs and outputs for geoprocessing tools to make scripting easier.

Chapter 9
Working with rasters

9.1 Introduction

Rasters present a unique type of spatial data, and a number of geoprocessing tools are designed specifically to take advantage of the raster data structure. This chapter illustrates how to use regular ArcPy functions to list and describe rasters. ArcPy also includes a Spatial Analyst module referred to as `arcpy.sa`, which fully integrates map algebra into the Python environment, making scripting much more efficient. Map algebra operators are described, as well as functions and classes of the `arcpy.sa` module.

9.2 Listing rasters

The `ListRasters` function returns a Python list of rasters in a workspace. The syntax of the function is

```
ListRasters({wild_card}, {raster_type})
```

An optional `wild_card` parameter can be used to limit the list based on the name of the rasters. The optional `raster_type` parameter can be used to limit the list based on the type of raster—for example, JPEG or TIFF.

The following code illustrates the use of the `ListRasters` function to print a list of the rasters in a workspace:

```
import arcpy
from arcpy import env
env.workspace = "C:/raster"
rasterlist = arcpy.ListRasters()
for raster in rasterlist:
    print raster
```

The output would look something like the following:

```
elevation
landcover.tif
tm.img
```

The name of each raster is printed to the Interactive Window in PythonWin or the next line in the Python window, along with an optional file extension. For example, it is .img for the ERDAS IMAGINE format, .tif for the TIFF format, .jpg for the JPEG format, and so on. No file extensions are added for the Esri GRID (global resource information database) format or for rasters stored inside a geodatabase. Therefore, when no file extension is present, be sure to determine whether you are working with a GRID or with a raster dataset inside a geodatabase.

The parameters of the `ListRasters` function can be used to filter the results. For example, the following code prints a list of the rasters in the workspace that are in the ERDAS IMAGINE format:

```
import arcpy
from arcpy import env
env.workspace = "C:/raster"
rasterlist = arcpy.ListRasters("*", "IMG")
for raster in rasterlist:
    print raster
```

Once the names of the rasters are obtained, other functions can be used, including functions to describe the data as discussed in the next section.

9.3 Describing raster properties

Rasters can be described using the generic `Describe` function as already discussed in chapter 6. The `Describe` function returns the properties for a specified data element. These properties are dynamic, which means the properties that are present depend on the data type being described. For example, when the `Describe` function is used on rasters, a generic set of properties is present in addition to specific properties that are unique to the specific raster element.

Three different raster data elements can be distinguished:

1. Raster dataset—a raster spatial data model that is stored on disk or in a geodatabase. Raster datasets can be stored in many formats, including TIFF, JPEG, IMAGINE, Esri GRID, and MrSID. Raster datasets can be single band or multiband.

2. Raster band—one layer in a raster dataset that represents data values for a specific range in the electromagnetic spectrum or other values derived by manipulating the original image bands. Many types of satellite images, for example, contain multiple bands.

3. Raster catalog—a collection of raster datasets defined in a table of any format, in which the records define the individual raster datasets that are included in the catalog. Raster catalogs can be used to display adjacent or overlapping raster datasets without having to combine them into a mosaic in one large file.

Properties for each of these elements vary. For example, the format (TIFF, JPEG, and others) is a property of the raster dataset and the cell size is a property of the raster band. The general `dataType` property can be used to determine the type of data element. All properties, however, are accessed using the same `Describe` function.

The following code illustrates the use of the `Describe` function, which returns an object with properties that can be accessed, in this case for printing:

```
import arcpy
from arcpy import env
env.workspace = "C:/raster"
raster = "landcover.tif"
desc = arcpy.Describe(raster)
print desc.dataType
```

For this example of a raster in TIFF format, the `dataType` property returns the type `RasterDataset`. Properties that are specific to raster datasets only include the following:

- `bandCount`—the number of bands in the raster dataset
- `compressionType`—the compression type (LZ77, JPEG, JPEG2000, or None)
- `format`—the raster format (GRID, IMAGINE, TIFF, and more)
- `permanent`—indicates the permanent state of the raster: `False` if the raster is temporary, `True` if the raster is permanent
- `sensorType`—the sensor type used to capture the image

Once it has been determined that an element is a raster dataset, these properties can be accessed. For example, the following code includes additional properties used to describe the TIFF file:

```
import arcpy
from arcpy import env
env.workspace = "C:/raster"
raster = "landcover.tif"
desc = arcpy.Describe(raster)
print desc.dataType
print desc.bandCount
print desc.compressionType
```

This particular .tif file is a single-band uncompressed TIFF, and therefore the property `bandCount` returns a value of `1` and `compressionType` returns a value of `None`.

Many other properties that are commonly associated with rasters can be accessed for individual raster bands only. For example, the cell resolution is a very important raster property, but individual bands within one raster dataset can have different resolutions. A number of properties are specific to raster bands, including the following:

- `height`—the number of rows
- `isInteger`—indicates whether the raster band is an integer type
- `meanCellHeight`—the cell size in y direction
- `meanCellWidth`—the cell size in x direction
- `noDataValue`—the NoData value of the raster band
- `pixelType`—the pixel type, such as 8-bit integer, 16-bit integer, single precision floating point, and others
- `primaryField`—the index of the field
- `tableType`—the class name of the table
- `width`—the number of columns

For single-band raster datasets, the band itself does not have to be specified (there is only one, after all) and the properties can be accessed directly by describing the raster dataset. For example, the following code determines the cell size and pixel type of a raster:

```
import arcpy
from arcpy import env
env.workspace = "C:/raster"
rasterband = "landcover.tif"
desc = arcpy.Describe(raster)
print desc.meanCellHeight
print desc.meanCellWidth
print desc.pixelType
```

For this particular example, the code returns values of 30.0 by 30.0 and U8—this means the cell size is 30 by 30 meters and the pixel type is an unsigned 8-bit integer. These properties do not report the unit type, which has to be obtained from the Spatial Reference property. For example, the following code determines the name of the spatial reference and the unit:

```
spatialref = desc.spatialReference
print spatialref.name
print spatialref.linearUnitName
```

For multiband rasters, however, the specific band needs to be specified. Without a particular band being specified, properties such as cell size, height, width, and pixel type cannot be accessed. Specific bands are referenced using Band_1, Band_2, and so on. The following code illustrates how the properties for a band in a multiband raster dataset are accessed:

```
import arcpy
from arcpy import env
env.workspace = "C:/raster"
rasterband = "img.tif/Band_1"
desc = arcpy.Describe(rasterband)
print desc.meanCellHeight
print desc.meanCellWidth
print desc.pixelType
```

Note: *Individual bands in a raster are sometimes referenced, for example, as Layer_1 and Layer_2, instead of Band_1 and Band_2.*

9.4 Working with raster objects

ArcPy also contains a Raster class that is used to reference a raster dataset. A raster object can be created in two ways: (1) by referencing an existing raster on disk and (2) by using a map algebra statement. The syntax for the Raster class is

```
Raster(inRaster)
```

The following code illustrates how to create a raster object by referencing a raster on disk:

```
import arcpy
myraster = arcpy.Raster("C:/raster/elevation")
```

When using map algebra statements, the code looks something like the following:

```
import arcpy
outraster = arcpy.sa.Slope("C:/raster/elevation")
```

In both cases, the resulting raster object can be used in Python statements and additional map algebra expressions. Raster objects have many properties, which are largely similar to those already discussed earlier in this chapter, including `bandCount`, `compressionType`, `format`, `height`, `width`, `meanCellHeight`, `meanCellWidth`, `pixelType`, `spatialReference`, and others. Similar to the `Describe` function, these properties are mostly read-only.

Raster objects have only one method: `save`. The raster object (the variable and associated dataset) returned from a map algebra expression is temporary by default. This means the variable and the referenced dataset are deleted when the variable goes out of scope—for example, when ArcGIS is closed or when a stand-alone script is closed. The `save` method can be called to make the raster object permanent. The syntax of the save method is

```
save({name})
```

In the earlier example, the raster object outraster is temporary but can be made permanent using the following code:

```
import arcpy
outraster = arcpy.sa.Slope("C:/raster/elevation")
outraster.save("C:/raster/slope")
```

It may appear somewhat counterintuitive that map algebra expressions result in temporary outputs. Keep in mind that a typical workflow using

rasters can involve numerous steps. If only the final output is actually needed, keeping temporary outputs as intermediate steps results in fewer output files and lower storage requirements.

9.5 Working with the ArcPy Spatial Analyst module

ArcPy includes a Spatial Analyst module, `arcpy.sa`, to carry out map algebra and other operations. The functionality provided by the Spatial Analyst module is largely the same as that of the tools in the Spatial Analyst toolbox. For example, you can run the Slope tool by referencing the Slope tool in the Spatial Analyst toolbox or by importing the `arcpy.sa` module and directly referencing the Slope tool.

The Spatial Analyst module integrates map algebra into the Python environment. This is similar to the use of map algebra in such ArcToolbox geoprocessing tools as Raster Calculator, Single Output Map Algebra, and Multiple Output Map Algebra in earlier versions of ArcGIS. The ArcPy Spatial Analyst module has a series of operators to support map algebra operations.

The Spatial Analyst module provides access to all the raster geoprocessing tools in the Spatial Analyst toolbox. It offers an alternative way to run these tools that can be more efficient than running them using the Spatial Analyst toolbox. Consider the following code that runs the Slope tool:

```
import arcpy
elevraster = arcpy.Raster("C:/raster/elevation")
outraster = arcpy.sa.Slope(elevraster)
```

Notice that the Slope tool is called using `arcpy.sa.Slope`, which appears to follow the regular syntax used for all tools: `arcpy.<toolboxalias>.<toolname>`. However, the alternative `arcpy.<toolname>_<toolboxalias>` syntax does not apply here, and `arcpy.Slope_sa` is not valid. Because `sa` is a module, and not just the alias of a toolbox, the code can be simplified as follows:

```
import arcpy
from arcpy.sa import *
elevraster = arcpy.Raster("C:/raster/elevation")
outraster = Slope(elevraster)
```

The statement `from arcpy.sa import *` imports all the functions from the `arcpy.sa` module, and tools can therefore be called directly—for example, `Slope` versus `arcpy.sa.Slope`. Initially, this may not appear to be much of a saving, but imagine having several dozen raster functions in

a single map algebra expression—omitting `arcpy.sa` several dozen times makes your code shorter and easier to read.

9.6 Using map algebra operators

In addition to providing access to all the Spatial Analyst geoprocessing tools, the `arcpy.sa` module also includes a number of map algebra operators. Most of these operators are available as geoprocessing tools under the Math toolset in the Spatial Analyst toolbox yet are also available as operators in Python. Consider the following example, which converts elevation values from feet to meters using the Times tool:

```
import arcpy
from arcpy.sa import *
elevraster = arcpy.Raster("C:/raster/elevation")
outraster = Times(elevraster, "0.3048")
outraster.save("C:/raster/elev_m")
```

Instead of using the Times tool, the map algebra operator (*) can be used. The second-to-last line of code would look as follows:

```
outraster = elevraster * 0.3048
```

This alternative is a bit shorter, but more importantly, it makes it possible to write elaborate map algebra expressions relatively easily.

Consider the example of a suitability model in which you create three different rasters, each representing a different factor in the suitability model. In the final analysis step, you want to add these three rasters together and determine the average suitability score. Your code could look something like the following:

```
import arcpy
from arcpy.sa import *
f1 = arcpy.Raster("C:/raster/factor1")
f2 = arcpy.Raster("C:/raster/factor2")
f3 = arcpy.Raster("C:/raster/factor3")
temp1raster = Plus(f1, f2)
temp2raster = Plus(temp1raster, f3)
outraster = Divide(temp2raster, "3")
outraster.save("C:/raster/final")
```

The Plus tool has to be used twice to add all three rasters together because the tool can use only two inputs at a time. The Divide tool is used to divide

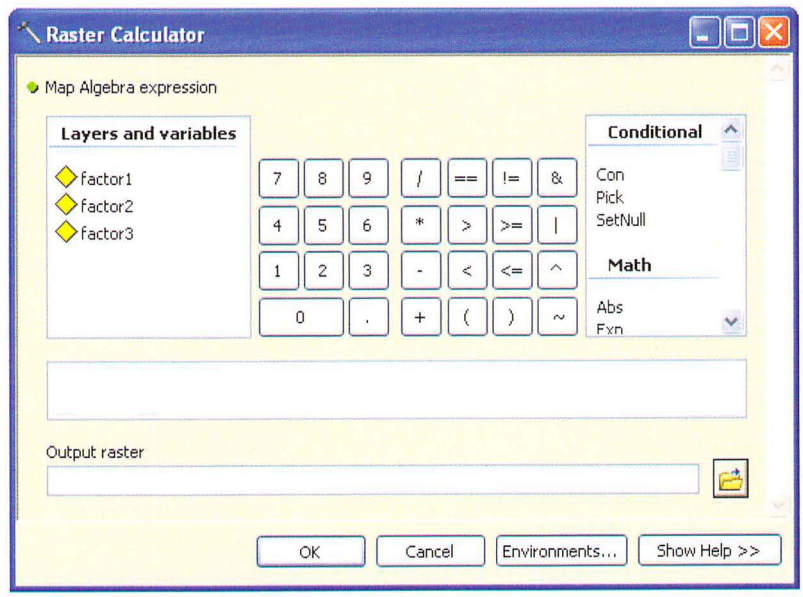
the sum of the three rasters by 3. Using map algebra expressions, this code can be reduced as follows:

```
import arcpy
from arcpy.sa import *
f1raster = arcpy.Raster("C:/raster/factor1")
f2raster = arcpy.Raster("C:/raster/factor2")
f3raster = arcpy.Raster("C:/raster/factor3")
outraster = (f1 + f2 + f3) / 3
outraster.save("C:/raster/final")
```

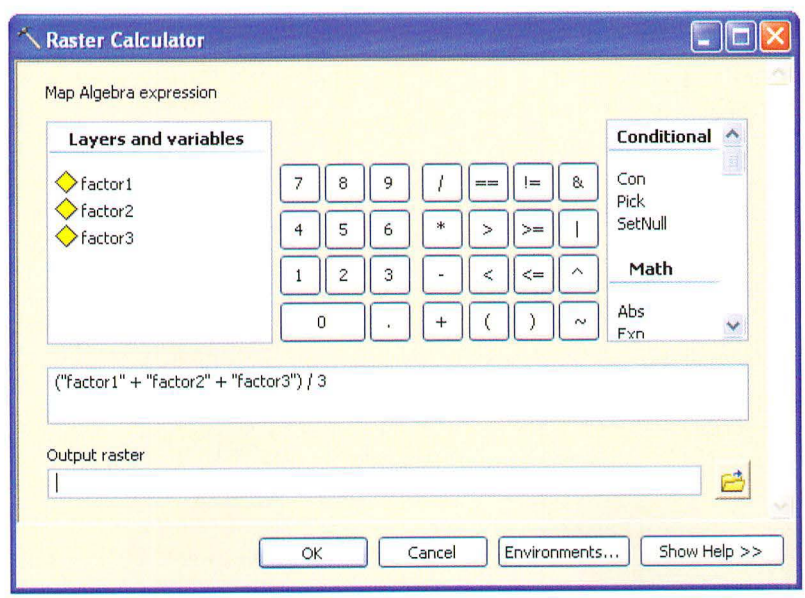
This saves a fair amount of code and also makes it easier to recognize the map algebra expression.

If this kind of statement looks familiar, it is probably because you have used Raster Calculator before. In earlier versions of ArcGIS, it was a tool from the Spatial Analyst toolbar, but in version 10, it was added as a tool in the Spatial Analyst toolbox.

The Raster Calculator tool dialog box looks like the example in the figure. ➔



The dialog box allows you to create an expression by selecting from the layers and the tools. For example, the expression needed for the suitability model would look like the example in the figure. ➔



It looks very much like the Python code used earlier. In effect, the map algebra operators in the `arcpy.sa` module allow you to create Raster Calculator-style expressions directly in Python. You can also call the Raster Calculator tool using the following syntax:

```
RasterCalculator(expression, output_raster)
```

However, this will not make your code much more efficient unless you are using an expression that has map algebra operators that are not available directly in Python.

Table 9.1 shows a list of the map algebra operators that are available from the `arcpy.sa` module. They are grouped into four categories: Arithmetic, Bitwise, Boolean, and Relational.

Table 9.1 Map algebra operators

Category	Operator	Description	Spatial Analyst tool
Arithmetic	-	Subtraction	Minus
	-	Unary Minus	Negate
	%	Modulo	Mod
	*	Multiplication	Times
	/	Division	Divide
	//	Integer Division	N/A
	+	Addition	Plus
	+	Unary Plus	N/A
	**	Power	Power
Bitwise	>>	Bitwise Right Shift	Bitwise Right Shift
	<<	Bitwise Left Shift	Bitwise Left Shift
Boolean	~	Boolean Complement	Boolean Not
	&	Boolean And	Boolean And
	^	Boolean Exclusive Or	Boolean XOr
	|	Boolean Or	Boolean Or
Relational	<	Less Than	Less Than
	<=	Less Than or Equal To	Less Than Equal
	>	Greater Than	Greater Than
	>=	Greater Than or Equal To	Greater Than Equal
	==	Equal To	Equal To
	!=	Not Equal To	Not Equal

A detailed discussion of each operator is beyond the scope of this book, but some observations are in order:

- Most operators have an equivalent tool in the Math toolset, but two do not: // (integer division) and + (unary plus). However, the same tasks can be accomplished using a combination of tools.
- Many of the tools in the Math toolset do not have an equivalent map algebra operator in Python, including commonly used tools such as Abs, Int, Float, Exp10, Log10, and many others.

9.7 Using the ApplyEnvironment function

In addition to the geoprocessing tools in the Spatial Analyst toolbox, there is one more function: the `ApplyEnvironment` function. This function copies an existing raster and applies the current environment settings. The syntax of the function is

```
ApplyEnvironment(in_raster)
```

This function allows you to change things like the extent or the cell size or to apply an analysis mask. The following code illustrates how the `ApplyEnvironment` function is used to set a new cell size of 30 and apply an analysis mask based on an existing shapefile:

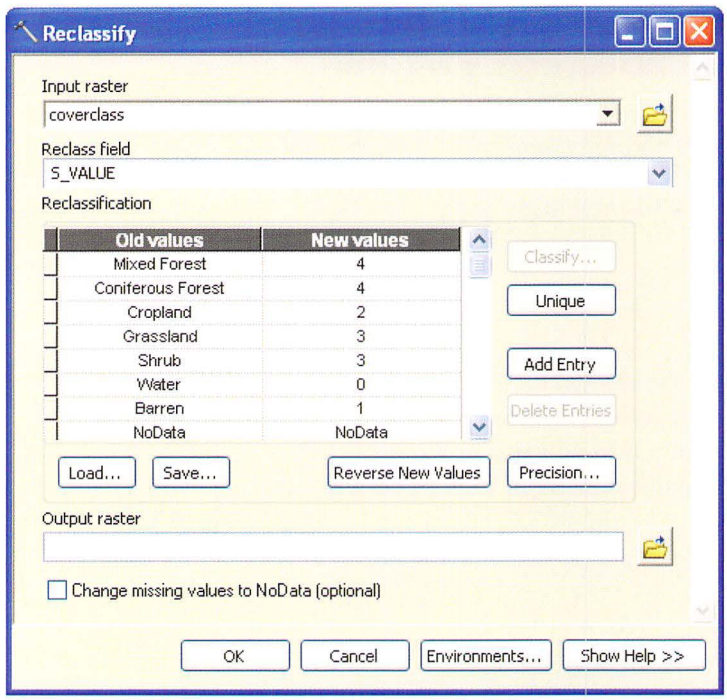
```
import arcpy
from arcpy import env
from arcpy.sa import *
elevfeet = arcpy.Raster("C:/raster/elevation")
elevmeter = elevfeet * 0.3048
env.cellsize = 30
env.mask = "C:/raster/studyarea.shp"
elevrasterclip = ApplyEnvironment(elevmeter)
elevrasterclip.save("C:/raster/dem30_m")
```

Not all environment settings apply to the `ApplyEnvironment` function. They are limited to the following: Cell Size, Current Workspace, Extent, Mask, Output Coordinate System, Scratch Workspace, and Snap Raster. These are the most relevant environment settings when working with rasters.

9.8 Using classes of the arcpy.sa module

The arcpy.sa module also contains a number of classes that are used for defining parameters of raster tools. Typically, these classes are used as shortcuts for tool parameters that would otherwise require a more complicated string value.

Consider the example of the Reclassification tool. With this tool, raster cells are given a new value based on a reclassification table. The tool dialog box in the figure shows an example of a land-use raster being reclassified into a number of values as part of a suitability model.



The syntax of the Reclassify tool is as follows:

```
Reclassify(in_raster, reclass_field, remap, {missing_values})
```

Typing all the values of this table would be rather complicated since this table can have many different entries. Instead, the remap parameter is expressed as a remap object. There are two different Remap classes, depending on the nature of the reclassification:

1. RemapValue—a list of individual input values to be reclassified

2. RemapRange—a list identifying ranges of input values to be reclassified

The syntax of the RemapValue object is

```
RemapValue(remapTable)
```

A remapTable object is defined using a Python list of lists that each contain old and new values, similar to the reclassification table on the tool dialog box. The syntax of a remap table for use in a RemapValue object is

```
[[oldValue1, newValue1], [oldValue2, newValue2], ...]
```

The following code illustrates the use of a remap object to carry out a reclassification of a raster representing land use:

```
import arcpy
from arcpy import env
from arcpy.sa import *
env.workspace = "C:/raster"
myremap = RemapValue([["Barren", 1], ["Mixed Forest", 4], ["Coniferous
Forest", 0], ["Cropland", 2], ["Grassland", 3], ["Shrub", 3], ["Water",
0]])
outreclass = Reclassify("landuse", "S_VALUE", myremap)
outreclass.save("C:/raster/lu_recl")
```

The RemapRange object works in a similar manner but uses value ranges rather than individual values. The syntax of a remap table for use in a RemapRange object is

```
[[startValue, endValue, newValue], ...]
```

The following code illustrates the use of a remap object to carry out a reclassification of a raster of elevation:

```
import arcpy
from arcpy import env
from arcpy.sa import *
env.workspace = ("C:/raster")
myremap = RemapRange([[1, 1000, 0], [1000, 2000, 1], [2000, 3000, 2],
[3000, 4000, 3]])
outreclass = Reclassify("elevation", "TYPE", myremap)
outreclass.save("C:/raster/elev_recl")
```

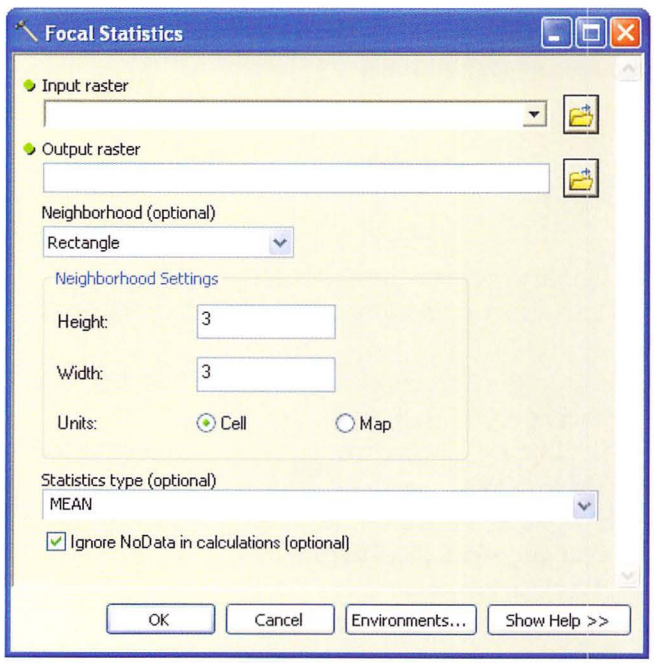
Notice that the end value of the first range is the same as the start value of the second range, and so on. This type of remap table is common when data is continuous, as in the case of a raster of elevation. In addition to the Reclassify tool, remap tables are also used in the Weighted Overlay tool.

There are many other classes in the `arcpy.sa` module. They can be grouped into a number of categories based on logical functionality. Table 9.2 lists these categories.

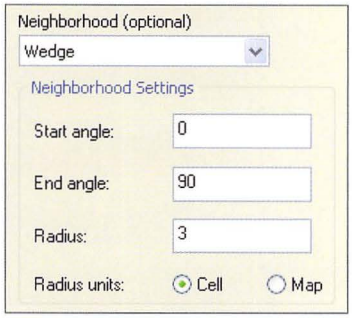
Table 9.2 Categories of classes in the `arcpy.sa` module

Category	Description
Fuzzy Membership	Defines the membership function for fuzzy logic analysis
Horizontal Factor	Identifies the horizontal factor for the Path Distance tool
Kriging Models	Develops the model for creating surfaces with kriging
Neighborhood	Defines the input neighborhood for a series of tools
Overlay	Creates input tables for the Weighted Overlay and Weighted Sum tools
Radius	Identifies a radius for the IDW and Kriging tools
Remap	Defines various remap tables used in reclassification
Time	Identifies the time interval to use in the solar radiation tools
Topo Input	Defines the input to the Topo To Raster tool
Vertical Factor	Identifies the vertical factor for the Path Distance tool

Among the more widely used classes, in addition to the Remap classes already discussed, are the Neighborhood classes, which define neighborhoods of different shapes and sizes. Consider, for example, the Focal Statistics tool. This tool, as well as other tools in the Neighborhood toolbox, requires the definition of a specific neighborhood.



The neighborhood settings vary with the type of neighborhood. For example, for the default rectangular neighborhood, settings include height and width in cell or map units. However, for the wedge neighborhood, the parameters include start angle and end angle and a radius in cell or map units.



Because of the variability of these parameters, neighborhood functions include a neighborhood object. For example, the syntax of the Focal Statistics tool is as follows:

```
FocalStatistics(in_raster, {neighborhood}, {statistics_type}, ➔
➔ {ignore_nodata})
```

There are six different neighborhood objects, each with its own unique set of parameters. They are as follows:

1. `NbrAnnulus`—defines an annulus, or ringlike, neighborhood, which is created by specifying an inner circle's and outer circle's radii in either map units or number of cells.

2. `NbrCircle`—defines a circular neighborhood, which is created by specifying the radius in either map units or number of cells.

3. `NbrIrregular`—defines an irregular neighborhood, which is created by a kernel file.

4. `NbrRectangle`—defines a rectangular neighborhood, which is created by specifying the height and the width in either map units or number of cells.

5. `NbrWedge`—defines a wedge neighborhood, which is created by specifying a radius and two angles in either map units or number of cells.

6. `NbrWeight`—defines a weight neighborhood, which identifies the cell positions to be included within the neighborhood and the weights for multiplying the cell values of the input raster.

For example, the syntax of the `NbrRectangle` object is

```
NbrRectangle({width}, {height}, {units})
```

The following code defines a neighborhood object and uses it in the `FocalStatistics` function:

```
import arcpy
from arcpy import env
from arcpy.sa import *
env.workspace = "C:/raster"
mynbr = NbrRectangle(5, 5, "CELL")
outraster = FocalStatistics("landuse", mynbr, "VARIETY")
outraster.save("C:/raster/lu_var")
```

In this example, the output is a raster of land cover based on a rectangular neighborhood of 5 cells by 5 cells.

9.9 Using raster functions to work with NumPy arrays

Two more raster functions need to be mentioned: `NumPyArrayToRaster` and `RasterToNumPyArray`. These are regular ArcPy functions, not functions of the `arcpy.sa` module. These two functions allow for conversions between rasters and NumPy arrays. A NumPy array is designed to work with very large arrays. NumPy itself is a package used for scientific computing with Python. Among other things, it provides a very powerful *n*-dimensional array object. This type of object makes it possible to move data between databases. For example, the SciPy package contains numerous algorithms that may be useful for a particular application, such as Fourier transforms, maximum entropy models, and multidimensional image processing. Rather than trying to build a tool in ArcGIS that carries out these specialized functions, you could write a script tool that converts a raster to a NumPy array, and then calls specialized functions from the SciPy package. A generic script would look as follows:

```
import arcpy, scipy
from arcpy.sa import *
inRaster = arcpy.Raster("C:/raster/myraster")
my_array = RasterToNumPyArray(inRaster)
outarray = scipy.<module>.<function>(my_array)
outraster = NumPyArrayToRaster(outarray)
outraster.save("C:/raster/result")
```

This is a simplified example and references a generic SciPy function, yet it illustrates how NumPy array functions can be used to export data for processing in another environment and to import the result back into an ArcGIS-compatible format—all within the same Python script. More information on NumPy (Numerical Python) and SciPy (Scientific Library for Python) can be found at http://numpy.scipy.org and http://www.scipy.org, respectively.

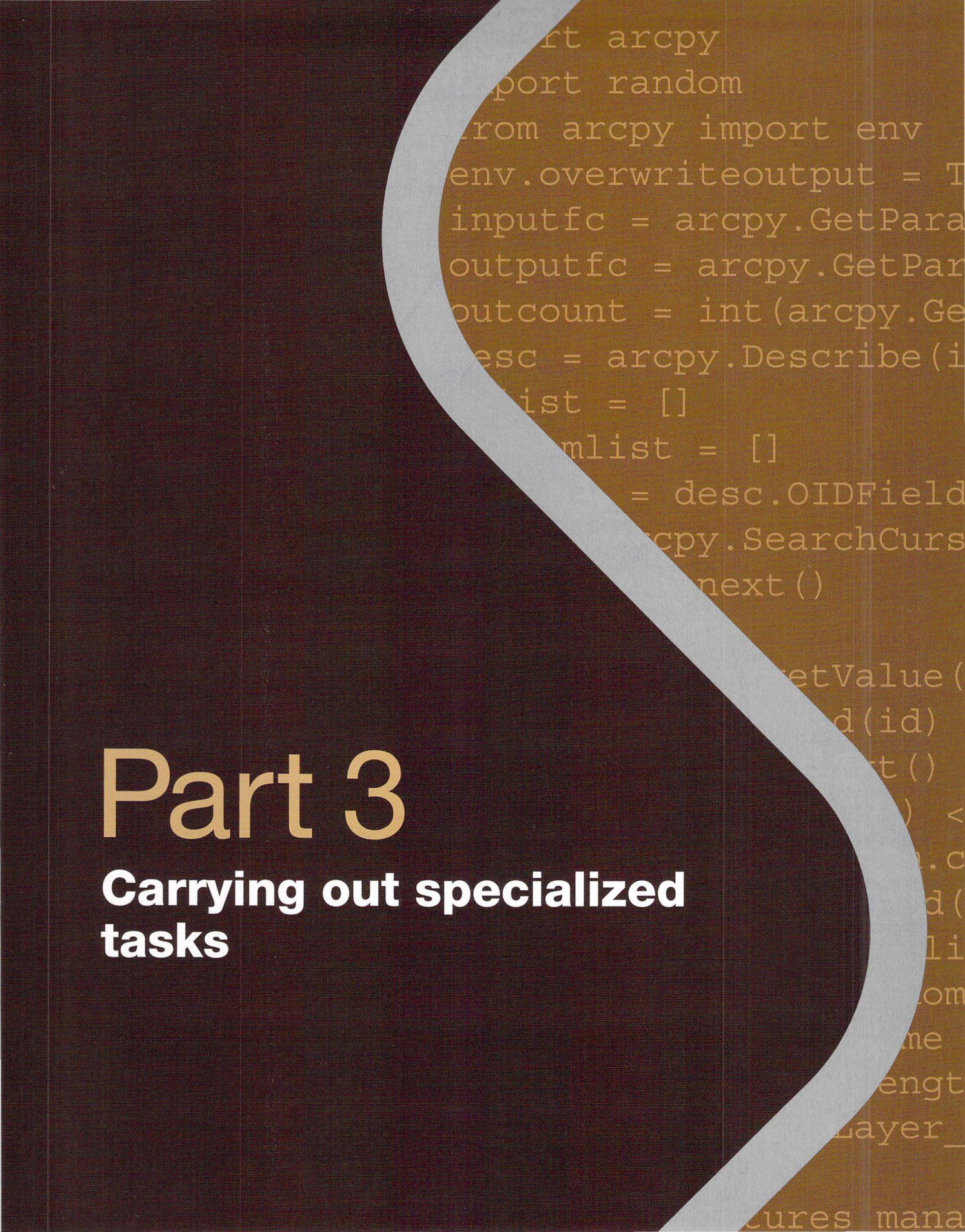
Points to remember

- The `ListRasters` function is used to list rasters in a workspace. The `Describe` function is used to describe raster datasets and raster bands. Properties of objects returned by the `Describe` function are dynamic—that is, they depend on the nature of the data type.

- The `arcpy.sa` module integrates map algebra into the Python environment. In addition to providing access to all the tools in the Spatial Analyst toolbox, the `arcpy.sa` module contains a number of map algebra operators, which make scripting with rasters more efficient.

- The `arcpy.sa` module also contains a number of classes, which are primarily used for defining certain types of parameters of raster tools.

- Conversion functions are available to export a raster to a NumPy array, which makes it possible to use analysis functions from other Python libraries such as SciPy.



Part 3

Carrying out specialized tasks

Chapter 10
Map scripting

10.1 Introduction

This chapter describes the ArcPy mapping module, also referred to as `arcpy.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 mapping 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 would normally carry out there.

Working with the ArcPy mapping module follows the workflow that would be used within an ArcMap session. For example, a typical workflow could 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 module 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 module: (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

```
MapDocument(mxd_path)
```

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

```
mapdoc = arcpy.mapping.MapDocument("C:/Mapping/Study_Areas.mxd")
```

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

```
mapdoc = arcpy.mapping.MapDocument("CURRENT")
```

To use the `CURRENT` keyword, ArcMap must be running because the `MapDocument` object will reference the map document currently loaded into ArcMap. Script tools 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 disabled 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 `MapDocument` 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 `RefreshActiveView` 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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("C:/Mapping/Study_Areas.mxd")
<code that modifies map document properties>
mapdoc.save()
del mapdoc
```

Note: *When a script is finished running, Python automatically removes references to map documents, so the* `del` *statement is not required but still reduces the likelihood of unwanted locks.*

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 ArcGIS 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 (`deleteThumbnail` 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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("CURRENT")
path = mapdoc.filePath
print path
del mapdoc
```

Running this code prints a system path. For example:

```
C:\Maps\final.mxd
```

The `del` statement ensures that the map document lock is removed.

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

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("CURRENT")
mapdoc.title = "Final map of study areas"
mapdoc.save()
del mapdoc
```

As you review these basic examples, remember that they can be used to automate more complex tasks, such as making changes to multiple map documents rather than just the current one.

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:

```
ListDataFrames(map_document, {wild_card})
```

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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("CURRENT")
listdf = arcpy.mapping.ListDataFrames(mapdoc)
for df in listdf:
    print df.name
del mapdoc
```

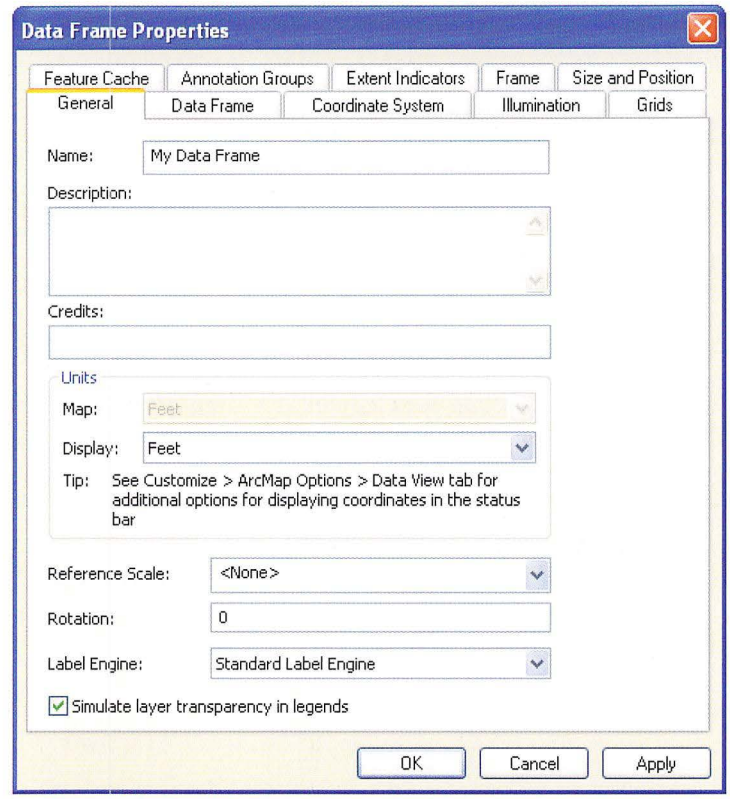
If you want to work with just one of the data frames, you can use its index number, as follows:

```
print listdf[0].name
```

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 Help. 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 document is set to the same spatial reference as that of a specific shapefile, and the scale of all data frames is set to 1:24,000:

```
import arcpy
dataset = "C:/map/boundary.shp"
spatialRef = arcpy.Describe(dataset).spatialReference
mapdoc = arcpy.mapping.MapDocument("C:/map/final.mxd")
for df in arcpy.mapping.ListDataFrames(mapdoc):
    df.spatialReference = spatialRef
    df.scale = 24000
del mapdoc
```

In addition to the properties already discussed, the `DataFrame` object also has two methods: `panToExtent` and `zoomToSelectedFeatures`. The `panToExtent` method maintains the data frame scale 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`:

```
df.panToExtent(lyr.getExtent())
```

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`:

```
df.zoomToSelectedFeatures()
```

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

```
Layer(lyr_file_path)
```

The parameter of the `Layer` function is the full path and file name of an existing .lyr file. For example:

```
Lyr = arcpy.mapping.Layer("C:/Mapping/study.lyr")
```

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 `ListLayers` function is

```
ListLayers(map_document_or_layer, {wild_card}, {data_frame})
```

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:

```
import arcpy
myDoc = arcpy.mapping.MapDocument("CURRENT")
lyrlist = arcpy.mapping.ListLayers(mapdoc)
for lyr in lyrlist:
    print lyr.name
```

To access just the layers in a specific data frame, the `DataFrame` 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 (`""`).

```
import arcpy
myDoc = arcpy.mapping.MapDocument("CURRENT")
dflist = arcpy.mapping.ListDataFrames(mapdoc)
lyrlist = arcpy.mapping.ListLayers(mapdoc, "", dflist[0])
for lyr in lyrlist:
    print lyr.name
```

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

```
import arcpy
lyrfile = arcpy.mapping.Layer("C:/Data/mylayers.lyr")
lyrlist = arcpy.mapping.ListLayers(lyrfile)
for lyr in lyrlist:
    print lyr.name
```

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 would 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.

Note: *Layers can get a bit confusing when working with a group layer. In this case, a single .lyr file can contain more than one layer, which is why the* `ListLayers` *function can have a .lyr file as a parameter. For a .lyr file with only a single layer, which is typical, the* `ListLayers` *function returns a list object with a single value. For the same .lyr file, however, the* `Layer` *function returns a single* `Layer` *object.*

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:

```
import arcpy
myDoc = arcpy.mapping.MapDocument("CURRENT")
dflist = arcpy.mapping.ListDataFrames(mapdoc)
lyrlist = arcpy.mapping.ListLayers(mapdoc, "", dflist[0])
for lyr in lyrlist:
    lyr.showLabels = True
del lyrlist
```

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":

```
import arcpy
myDoc = arcpy.mapping.MapDocument("CURRENT")
lyrlist = arcpy.mapping.ListLayers(mapdoc)
for lyr in lyrlist:
    if lyr.name == "hospitals":
        lyr.showLabels = True
del lyrlist
```

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:

```
if lyr.name.lower() == "hospitals":
```

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

```
supports(layer_property)
```

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:

```
import arcpy
myDoc = arcpy.mapping.MapDocument("CURRENT")
dflist = arcpy.mapping.ListDataFrames(mapdoc)
lyrlist = arcpy.mapping.ListLayers(mapdoc, "", dflist[0])
for lyr in lyrlist:
    if lyr.supports("SHOWLABELS") == True:
        lyr.showLabels = True
del lyrlist
```

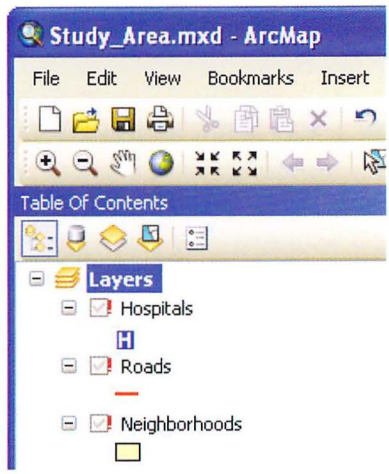
If you are unsure whether a layer supports a particular property, use the `supports` method to test it. Otherwise, you will need to use an error-trapping method, such as a `try-except` statement, which is covered in chapter 11.

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:

- `AddLayer`—makes it possible to add a layer to a data frame within a map document using general placement options.
- `AddLayerToGroup`— makes it possible to add a layer to a group layer within a map document using general placement options.
- `InsertLayer`—makes it possible to add a layer to a data frame or to a group layer within a map document. It provides a more precise way of positioning the layer by using a reference layer.
- `MoveLayer`—makes it possible to move a layer to a specific location within a data frame or group layer within a map document.
- `RemoveLayer`—makes it possible to remove a layer from a map document.
- `UpdateLayer`—makes it possible to update the layer properties or just the symbology of a layer in a map document by extracting the information from a source layer.

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 the 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:

- *Workspace*—a container for data. It can be a folder that contains shapefiles, a coverage, or a geodatabase—for example, mydata.
- *Workspace 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

```
ListBrokenDataSources(map_document_or_layer)
```

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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("CURRENT")
brokenlist = arcpy.mapping.ListBrokenDataSources(mapdoc)
for lyr in brokenlist:
    print lyr.name
del mapdoc
```

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:

```
print lyr.dataSource
```

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 `MapDocument.findAndReplaceWorkspacePaths` is

```
MapDocument.findAndReplaceWorkspacePaths(find_workspace_path, replace_
➥ workspace_path, {validate})
```

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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("CURRENT")
mapdoc.findAndReplaceWorkspacePaths("C:/mydata", "C:/newdata")
mapdoc.save()
del mapdoc
```

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

```
MapDocument.replaceWorkspaces(old_workspace_path, old_workspace_type,
new_workspace_path, new_workspace_type, {validate})
```

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

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("C:/mydata/project.mxd")
mapdoc.replaceWorkspaces("C:/mydata/shapes", "SHAPEFILE_WORKSPACE",
"C:/mydata/database.gdb", "FILEGDB_WORKSPACE")
mapdoc.save()
del mapdoc
```

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.

Valid workspace types are listed as follows:

- `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

```
Layer.findAndReplaceWorkspacePath(find_workspace_path, replace_workspace_path, {validate})
```

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 replaced—in this case, using a different personal geodatabase:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("C:/mydata/project.mxd")
lyrlist = arcpy.mapping.ListLayers(mapdoc):
for lyr in lyrlist:
    if lyr.supports("DATASOURCE"):
        if lyr.dataSource == "C:/mydata/database.gdb/hospitals":
            lyr.findAndReplaceWorkspacePath("database.gdb", "newdata.gdb")
mapdoc.save()
del mapdoc
```

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

```
Layer.replaceDataSource(workspace_path, workspace_type, dataset_name, {validate})
```

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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("C:/mydata/project.mxd")
lyrlist = arcpy.mapping.ListLayers(mapdoc):
for lyr in lyrlist:
    if lyr.supports("DATASOURCE"):
        if lyr.dataSource == "C:/mydata/hospitals.shp":
            lyr.replaceDataSource("C:/mydata/hospitals.shp", "SHAPEFILE_
➥ WORKSPACE", "C:/mydata/newhospitals.shp")
mapdoc.save()
del mapdoc
```

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

```
ListLayoutElements(map_document, {element_type}, {wild_card})
```

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 specific 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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument(r"C:\mydata\project.mxd")
elemlist = arcpy.mapping.ListLayoutElements(mapdoc)
for elem in elemlist:
    print elem.name & " " & elem.type
del mapdoc
```

The printout may look something like the following:

```
Legend LEGEND_ELEMENT
Alternating Scale Bar MAPSURROUND_ELEMENT
North Arrow MAPSURROUND_ELEMENT
Title TEXT_ELEMENT
Study Area DATAFRAME_ELEMENT
```

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:

```
title = arcpy.mapping.ListLayoutElements(mapdoc)[3]
```

Using the `element_type` parameter:

```
title = arcpy.mapping.ListLayoutElements(mapdoc, "TEXT_ELEMENT")[0]
```

Using the `wild_card` parameter:

```
title = arcpy.mapping.ListLayoutElements(mapdoc, "", "Title")[0]
```

In the case of the `element_type` and `wild_card` parameters, the `ListLayoutElements` function returns a list with only a single 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 element 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 `textElement` 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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("C:/mydata/project.mxd")
title = arcpy.mapping.ListLayoutElements(mapdoc, "TEXT_ELEMENT")[0]
title.text = "New Study Area"
mapdoc.save()
del mapdoc
```

A few more examples of code follow 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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("CURRENT")
elemlist = arcpy.mapping.ListLayoutElements(mapdoc, "PICTURE_ELEMENT")
for elem in elemlist:
    if elem.name == "photo1":
        elem.sourceImage = "C:/myphotos/newimage.jpg"
mapdoc.save()
del mapdoc
```

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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("CURRENT")
df = arcpy.mapping.ListDataFrames(mapdoc)[0]
lyr1 = arcpy.mapping.Layer("C:/mydata/Streets.lyr")
lyr2 = arcpy.mapping.Layer("C:/mydata/Ortho.lyr")
legend = arcpy.mapping.ListLayoutElements(mxd, "LEGEND_ELEMENT")[0]
legend.autoAdd = True
arcpy.mapping.AddLayer(df, lyr1, "BOTTOM")
legend.autoAdd = False
arcpy.mapping.AddLayer(df, lyr2, "BOTTOM")
mapdoc.save()
del mapdoc
```

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

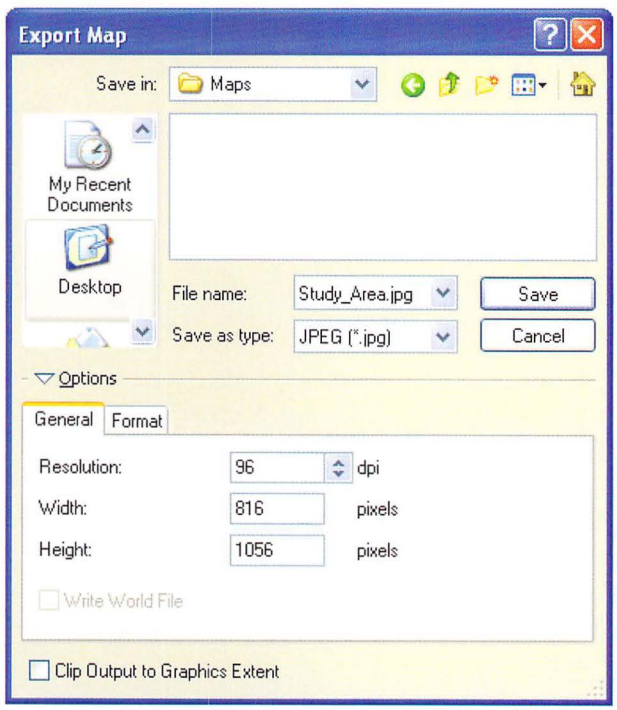
The ArcPy mapping module has a number of exporting functions. They are similar to the ArcMap operation File > Export Map. There is a separate function for each format. The functions are as follows:

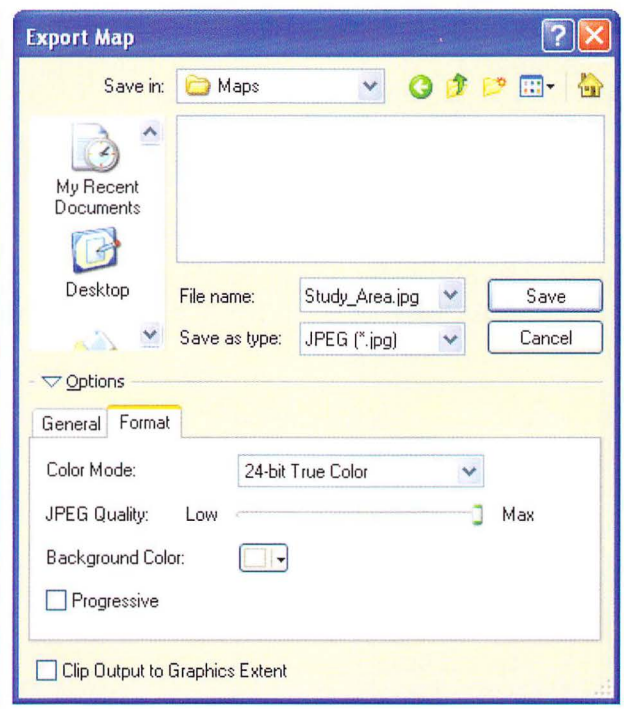
- `ExportToAI`
- `ExportToBMP`
- `ExportToEMF`
- `ExportToEPS`
- `ExportToGIF`
- `ExportToJPEG`
- `ExportToPDF`
- `ExportToPNG`
- `ExportToSVG`
- `ExportToTIFF`

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:

```
ExportToJPEG (map_document, out_jpeg, {data_frame}, {df_export_width},
{df_export_height}, {resolution}, {world_file}, {color_mode}, {jpeg_
quality}, {progressive})
```

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:

```
import arcpy
mapdoc = arcpy.mapping.MapDocument("C:/project/study.mxd")
arcpy.mapping.ExportToJPEG(mapdoc, "C:/project/final.jpg", "", "", "", ➔
➔ 600)
del mapdoc
```

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

```
PrintMap (map_document, {printer_name}, {data_frame}, {out_print_file})
```

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. In 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

```
PDFDocument(pdf_path)
```

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`, `updateDocProperties`, 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.

2. `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:

```
import arcpy
pdfpath = "C:/project/MapBook.pdf"
pdfdoc = arcpy.mapping.PDFDocumentCreate(pdfpath)
pdfdoc.appendPages("C:/project/Cover.pdf")
pdfdoc.appendPages("C:/project/Map1.pdf")
pdfdoc.appendPages("C:/project/Map2.pdf")
pdfdoc.saveAndClose()
del pdfdoc
```

>>> TIP

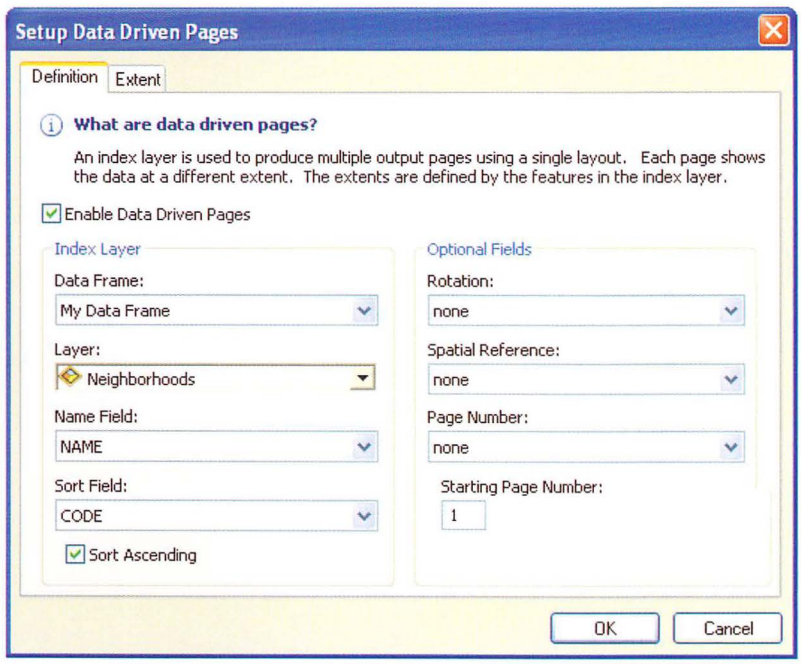
The `PDFDocumentCreate` function does not actually create any blank PDF pages. In the example at left, the actual PDF pages come from existing PDF files. However, these pages could also be created in a script from a map document by using the `ExportToPDF` function.

10.12 Creating map books

ArcGIS has a set of tools 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 PDF format. This is not the same as the `ExportToPDF` function. The syntax of the `exportToPDF` method is as follows:

```
exportToPDF(out_pdf, {page_range_type}, {page_range_string}, {multiple_
files}, {resolution}, {image_quality}, {colorspace}, {compress_vectors},
{image_compression}, {picture_symbol}, {convert_markers}, {embed_fonts},
{layers_attributes}, {georef_info})
```

The following code prints all the pages from a map document that has Data Driven Pages enabled to a PDF file and places an existing cover page in front:

```
import arcpy
pdfpath = "C:/project/MapBook.pdf"
pdfdoc = arcpy.mapping.PDFDocumentCreate(pdfpath)
mapdoc = arcpy.mapping.MapDocument("C:/project/Maps.mxd")
mapdoc.dataDrivenPages.exportToPDF("C:/project/Maps.pdf")
pdfdoc.appendPages("C:/project/Cover.pdf")
pdfdoc.appendPages("C:/project/Maps.pdf")
pdfdoc.saveAndClose()
del mapdoc
```

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, scales, dynamic text, and the like are probably easiest to control on the Setup Data Driven Pages dialog box in ArcMap, although printing the pages and merging different PDF files is easiest to control using scripting.

10.13 Using sample mapping scripts

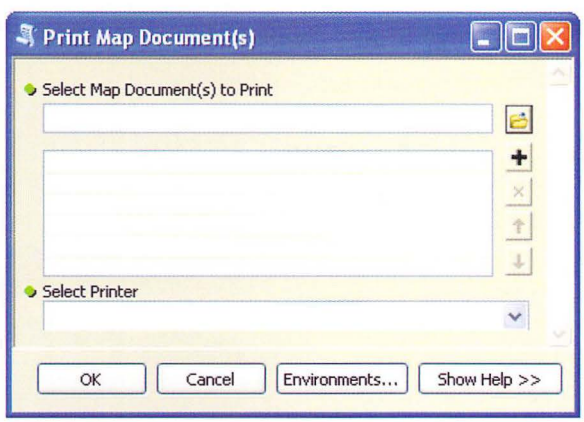
With the release of ArcGIS 10, Esri started making a number of script tools available to illustrate the use of ArcPy. These include a set of mapping script tools created as representative samples of how `arcpy.mapping` can be used to perform a variety of mapping tasks. These tools can be found in the Geoprocessing Model and Script Tool Gallery on the ArcGIS Resource Center.

> ***Note:*** *To obtain the sample tools, go to* 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.

- Cartography Tools.tbx
  - Adjust Layout Text Width (from ArcMap)
  - Find and Replace a Text String
  - Page Layout Element Report
  - Shift Page Layout Elements
  - Update Symbology
- Export and Printing Tools.tbx
  - Append PDF Documents
  - Export Map Documents to PDF
  - Print Data Driven Page(s)
  - Print Map Document(s)
- MXD and LYR Management Tools.tbx
  - Add Layer File into MXD (from ArcMap)
  - Find Broken Data Sources (Report)
  - Find Data Source (Report)
  - Find Layers Projected on the Fly (Report)
  - Multi Layer File Summary (Report)
  - Multi MXD Summary (Report)
  - Replace Layer with Layer File
  - Replace Layer with Layer File (from ArcMap)
  - Update MXD from pGDB to fGDB
  - Update MXD tags

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 is included in the files you get when downloading the tools.

PrintMXDs.py

```
# Author:  ESRI
# Date:    July 5, 2010
# Version: ArcGIS 10.0
# Purpose: This script will print one or more map documents to a local printer.
#          The script is intended to run within a script tool.  There are two
#          parameters:
#               1) Select Map Documents to Print,
#               2) Select Output Printer (auto populated using a validation script)
#
#Notes: The print order of the MXDs is based on how they are entered.  The MXD
#       at the top of the list is first followed by those below it.

import arcpy, string
import arcpy.mapping as MAP

#Read input parameters from script tool
MXDList = string.split(arcpy.GetParameterAsText(0), ";")
printer = arcpy.GetParameterAsText(1)

#Loop through each MXD and print
for MXDPath in MXDList:
    MXD = MAP.MapDocument(MXDPath)
    MAP.PrintMap(MXD, printer)

#Remove variable reference to file
del MXD
```

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.

>>> TIP

When modifying sample scripts, you should first save a copy of the script since there is no Undo button once you save your changes to the script.

Points to remember

- The `arcpy.mapping` module makes it possible to automate mapping tasks. A number of specific mapping classes and functions allow for the manipulation of map documents, data frames, layers, and page layouts.

- The functionality of the `arcpy.mapping` module reflects some of the typical workflows in ArcMap to produce cartographic output. Many procedures, however, are not part of the `arcpy.mapping` module because they lend themselves much more to the highly visual interface of ArcMap. For example, most of the layer symbology properties can be set only on the Layer Properties dialog box in ArcMap. The `arcpy.mapping` module can be used to automate certain repetitive tasks, such as updating the data sources for a large number of layers or replacing text in a large number of map documents.

- Map documents can be opened by referencing .mxd files on disk or by calling the map document currently in use. Map document properties can be accessed, modified, and saved. The `arcpy.mapping` module cannot create new map documents.

- Data frames within a map document can be accessed using the `ListDataFrames` function. Data frame properties can be accessed, modified, and saved.

- Layers within a data frame can be accessed using the `Layer` and `ListLayers` functions. Layer properties can be accessed, modified, and saved.

- Broken data sources in map documents can be identified using the `ListBrokenDataSources` function. Various methods exist to fix broken data sources for map document, layer, and table view objects. These methods can find and replace workspaces, workspace paths, and data sources.

- Individual elements on page layouts can be accessed and modified.

- Maps can be exported to various formats, including PDF, JPEG, and TIFF formats. Maps can also be printed to a local printer or to PDF format. When Data Driven Pages is enabled, scripting can be used to create a map book in PDF format.

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*, *exceptions*, 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:

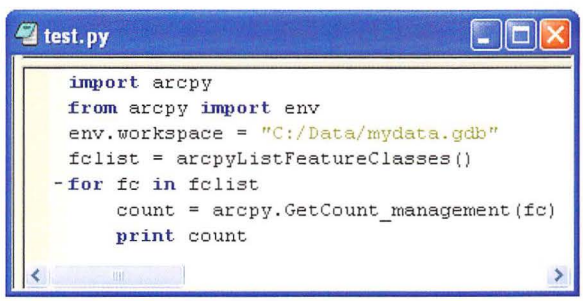
```
import arcpy
from arcpy import env
env.workspace = "C:/Data/mydata.gdb"
fclist = arcpy.ListFeatureClasses()
for fc in fclist
    count = arcpy.GetCount_management(fc)
    print count
```

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:

```
Parsing error SyntaxError: invalid syntax
```

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.



>>> TIP

Remember that you can make the line numbers visible in PythonWin by clicking View > Options > Editor on the menu bar, and then increasing the margin width for line numbers.

When you click the Check button, a message appears on the PythonWin status bar: `Failed to run script - 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 figure.

Changing line 5 to the following code fixes the syntax error:

```
for fc in fclist:
```

Running this code produces the following message: `Python and the TabNanny successfully checked the file 'test.py'`. This means there are no syntax errors and the code will run when you click the Run button.

The Check button runs a syntax checker and TabNanny, which checks for inconsistent indentation and spacing. For example, the following block of code uses inconsistent indentation:

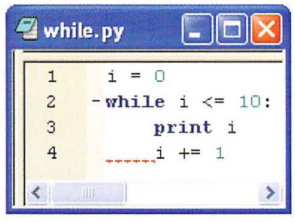
```
for fc in fclist:
    count = arcpy.GetCount_management(fc)
     print count
```

Using this code produces the following message:

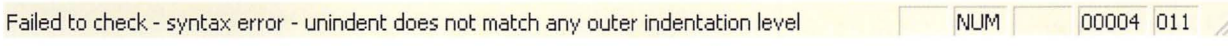
```
Failed to check - syntax error - unexpected indent
```

Similar errors can occur if your indentation uses a combination of spaces and tabs. Especially when you copy from other sources, such as Microsoft Word, Microsoft PowerPoint, or PDF, blocks of code may appear to be indented correctly visually but may actually use a combination of spaces and tabs.

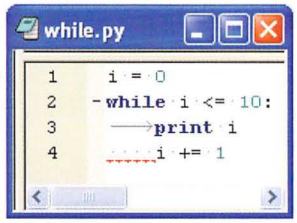
Consider the example code in the figure.



Visually, the block of code appears to be aligned, but TabNanny has underlined some of the whitespace in red. When a check is run, an error message appears, like the example in the figure.



To reveal the nature of the error, on the PythonWin menu bar, click View > Whitespace. The type of characters used for the indentation is displayed in the script window.



The arrow in the script window indicates the use of a tab, and the dots indicate spaces. Indentation should be consistent, so the tab should be replaced by four spaces.

> ***Note:*** *Although Python reports the line where a syntax error occurs, sometimes the actual syntax error occurs on a line that's above the one reported.*

>>> TIP

Copying code from other sources such as Word and .pdf files will likely introduce other types of errors, including quotation marks. In general, therefore, it is not recommended to copy code from these types of files.

>>> TIP

Using the TAB key in PythonWin results in four spaces by default. The spacing can be modified by clicking View > Options > Tabs and Whitespace from the menu bar. An actual tab is normally introduced only when copying from other applications.

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:

```
import arcpy
from arcpy import env
env.workspace = "C:/Data/mydata.gdb"
fclist = arcpy.ListFeatureClasses()
for fc in fclist:
    count = arcpy.GetCount_management(fc)
    print count
```

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 "errors," 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*—the 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 is, on which line of code it failed. Typically, you still have to figure out why the error occurred.

Carefully reviewing the content of error messages

Error messages generated by ArcPy are usually informative. Consider the following example:

```
import arcpy
arcpy.env.workspace = "C:/Data"
infcs = ["streams.shp", "floodzone.shp"]
outfc = "union.shp"
arcpy.Union_analysis(infcs, outfc)
```

This script carries out a union between two input feature classes, which are entered as a list. The result should be a new output feature class in the same workspace. The error message in PythonWin is as follows:

```
ExecuteError: Failed to execute. Parameters are not valid.
ERROR 000366: Invalid geometry type
Failed to execute (Union).
```

This is a specific error message produced by ArcPy, also referred to as an `ExecuteError` exception. The message is useful because it includes the statement: `Invalid geometry type`. Closer inspection of the input feature classes reveals that one of the inputs (streams.shp) is a polyline feature class, and the Union tool works with polygon features only. So the error message does not tell you exactly what is wrong (that is, it did not say that streams.shp is the geometry type polyline and that the Union tool does not accept this geometry type), but it points you in the right direction.

Not all error messages are as useful. Consider the following script:

```
import arcpy
arcpy.env.workspace = "C:/mydata"
infcs = ["streams.shp", "floodzone.shp"]
outfc = "union.shp"
arcpy.Union_analysis(infcs, outfc)
```

>>> TIP

When a specific error code is included in the error message, such as ERROR 000366, you can learn more about it in ArcGIS Desktop Help. In Help, go to Geoprocessing > Tool errors and warnings, and browse to the specific error by number.

This is, in fact, the same script, but it uses a different workspace (C:\mydata), which does not exist. The error message in PythonWin is as follows:

```
Traceback (most recent call last):
  File "C:\Python27\ArcGIS10.1\Lib\site-packages\PythonWin\pywin\
framework\scriptutils.py", line 325, in RunScript
    exec codeObject in __main__.__dict__
  File "C:\data\myunion.py", line 5, in <module>
    arcpy.Union_analysis(infcs, outfc)
  File "C:\Program Files (x86)\ArcGIS\Desktop10.1\arcpy\arcpy\
analysis.py", line 574, in Union
    raise e
ExecuteError: Failed to execute. Parameters are not valid.
ERROR 000366: Invalid geometry type
Failed to execute (Union).
```

Note: *When the same code is run in the Python window in ArcGIS, a different error message results:*

```
Runtime error <class 'ArcGISscripting.ExecuteError'>: ERROR 000732: Input
Features: Dataset streams.shp #;floodzone.shp # does not exist or is not
supported
```

The PythonWin error message is rather misleading. It appears to suggest that the error is on line 5 of the code (where the union is carried out) and that there is an issue with the geometry. The error is, in fact, on line 2 where an invalid workspace is defined, and the invalid geometry message results from the fact that no feature classes could be obtained from the nonexistent workspace. Unfortunately, the error-reporting functions can't always report a more specific error message, such as `Workspace does not exist`.

Carefully examining error messages can be useful since they may, in fact, hold the answer to how to fix a problem. But don't stare yourself blind poring over them because the error may be something quite different and the error messages could prove misleading.

Adding print statements to your script

When you have multiple lines of code that contain geoprocessing tools, it may not always be clear on which line an error occurred. In such cases, it may be useful to add `print` statements after each geoprocessing tool or other important steps to confirm they were run successfully. Consider the following code:

```
import arcpy
from arcpy import env
env.overwriteOutput = True
env.workspace = "C:/Data"
arcpy.Buffer_analysis("roads.shp", "buffer.shp", "1000 METERS")
print "Buffer completed"
arcpy.Erase_analysis("buffer.shp", "zone.shp", "erase.shp")
print "Erase completed"
arcpy.Clip_analysis("erase.shp", "wetlands.shp", "clip.shp")
print "Clip completed"
```

Even if the error message is cryptic and not informative, the `print` statements will illustrate which steps have been completed. The error can most likely be traced to the block of code just prior to the `print` statement that did not execute.

`Print` statements can be effective, but they are most useful when you already have a good idea of what might be causing the error. One of the downsides of using `print` statements is that they need to be cleaned up once the error has been fixed, which can be a substantial amount of work.

Selectively commenting out code

You can selectively *comment out* code to see if removing certain lines eliminates the error. If your script has a typical sequential workflow, you would work from the bottom up. For example, the following code illustrates how the lower lines of code are commented out, using double number signs (##), to isolate the error:

```
import arcpy
from arcpy import env
env.overwriteOutput = True
env.workspace = "C:/Data"
arcpy.Buffer_analysis("roads.shp", "buffer.shp", "1000 METERS")
##arcpy.Erase_analysis("buffer.shp", "streams.shp", "erase.shp")
##arcpy.Clip_analysis("erase.shp", "wetlands.shp", "clip.shp")
```

As with adding print statements, this approach of commenting out lines of code does not identify why an error occurs, but only helps you to isolate where it occurs.

Using a Python debugger

Another, more systematic, approach to debugging code is to use a Python debugger. A debugger is a tool that allows you to step through your code line by line, to place breakpoints in your code to examine the conditions at that point, and to follow how certain variables change throughout your code. Python has a built-in debugger module called `pdb`. It is a bit cumbersome because it lacks a user interface. However, Python editors such as IDLE and PythonWin include a solid debugging environment. In the next example that follows, the PythonWin debugger is used.

In PythonWin, you can turn the Debugger toolbar on and off by clicking View > Toolbars > Debugging from the menu bar.

The tools on the Debugger toolbar are briefly described in table 11.1.

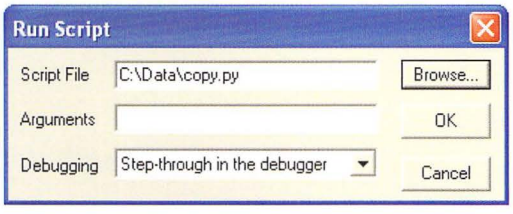
Table 11.1 Tools on the Debugger toolbar

	Watch	Makes the Watch window visible, which allows you to keep track of the values of specifically defined variables in a script
	Stack view	Makes the Stack window visible, which keeps track of all variables in a script
	Breakpoint list	Makes the Breakpoint list window visible, which lists all the current breakpoints in a script
	Toggle Breakpoint	Turns a breakpoint on or off at the cursor location in the current script
	Clear All Breakpoints	Removes all breakpoints from the current script
	Step (or Step Into)	Runs the current line of code and moves to the next line, which can be in a different module, function, or method
	Step Over	Runs the current line of code, and if it includes a Python module, function, or method, it runs it, and then returns to the next line of code in the original script
	Step Out	Runs the current Python module, function, or method, and then returns to the next line of code in the original script
	Go	Runs a script until the next breakpoint or until the last line of code is reached
	Close	Stops the execution of code and exits the debugger to return to the script

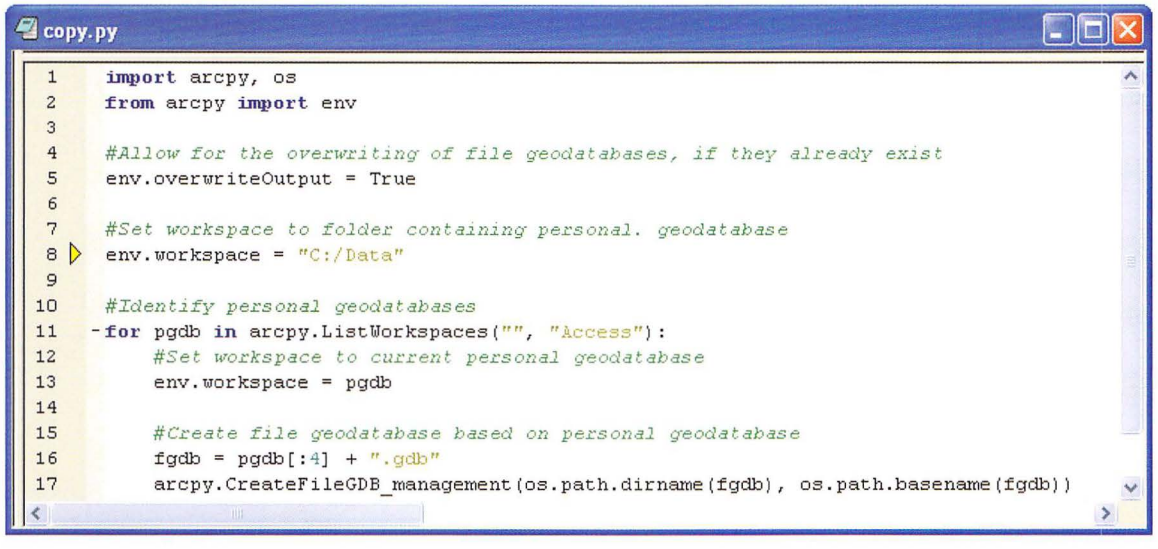
A typical debugging procedure in PythonWin is as follows:

1. Check for any syntax errors and save the script.

2. Run the script, but this time select a Debugging option on the Run Script dialog box—for example, "Step-through in the debugger".

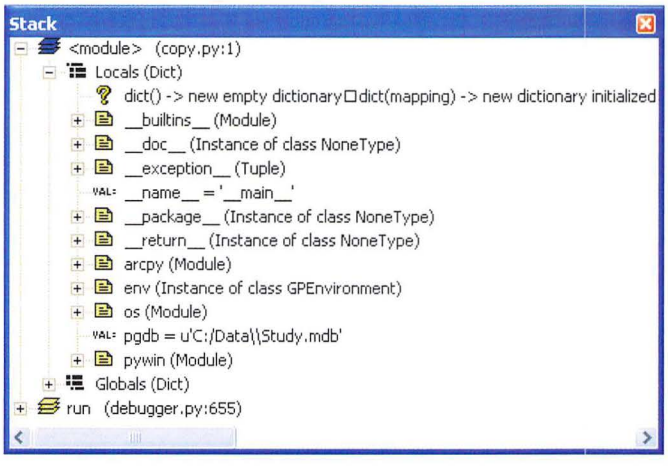


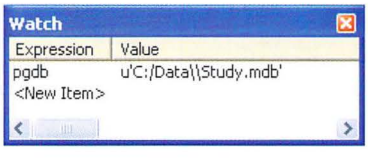
3. Use the Step tool to go through your script line by line. Keep track of any error messages in the Interactive Window. The yellow arrow that appears in the window indicates the current line of code.



4. Use the Step Over and Step Out tools to skip ahead. For example, if the current line of code contains a call to a different module, function, or method, using the Step tool results in stepping into that procedure. Using the Step Over tool will run the line of code without stepping into that procedure—you are stepping over the details of that procedure. Once you step into a procedure, you can use the Step Out tool to step out of the procedure without stepping through the rest of the lines of code. This allows you to fast-forward and return to the next line of code that called the procedure.

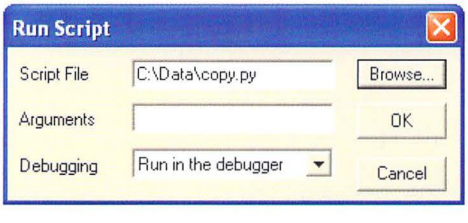
5. While stepping through the script code, open the Watch and Stack windows to keep track of variables. The Stack window keeps track of all the variables and the Watch window keeps track of only the variables you specify manually.





6. When you find an error, use the Close tool to stop the execution of the script. Then fix the error and run the script again.

When scripts get longer, going through a script line by line can be a bit cumbersome. Instead, you can place breakpoints in the script using the Toggle Breakpoint tool. The next time you run the script, you can select the "Run in the debugger" option.



Then the debugger will stop the script at only the predefined breakpoints and run the lines in between breakpoints in one step instead of stopping at every line.

copy.py

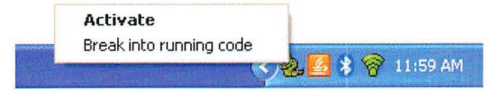
```
1  import arcpy, os
2  from arcpy import env
3
4  #Allow for the overwriting of file geodatabases, if they already exist
5  env.overwriteOutput = True
6
7  #Set workspace to folder containing personal. geodatabase
8  env.workspace = "C:/Data"
9
10 #Identify personal geodatabases
11 for pgdb in arcpy.ListWorkspaces("", "Access"):
12     #Set workspace to current personal geodatabase
13     env.workspace = pgdb
14
15     #Create file geodatabase based on personal geodatabase
16     fgdb = pgdb[:4] + ".gdb"
17     arcpy.CreateFileGDB_management(os.path.dirname(fgdb), os.path.basename(fgdb))
```

The breakpoints can be turned on and off by placing the cursor in the desired line of code and using the Toggle Breakpoint tool. To clear all the breakpoints in the current script, use the Clear All Breakpoints tool.

11.5 Using debugging tips and tricks

Following are some general tips and tricks that will help you to debug your scripts:

- Remember that ArcGIS for Desktop applications often place a lock on a file, which may prevent a script from overwriting the file.
- When working with very large files, first try your code on a small file with similar properties.
- Watch where the values of variables are changing by inserting print statements or breakpoints in the code.
- Place breakpoints inside blocks of code where repetition should be occurring.
- If PythonWin does not stop running while you are debugging, you can interrupt the code by right-clicking the PythonWin icon in the notification area, at the far-right corner of the taskbar, and then clicking "Break into running code". This will result in a `KeyboardInterrupt` exception in the Interactive Window without closing PythonWin. ➔



11.6 Error handling for exceptions

Although debugging procedures can contribute to writing correct code, exception errors are still likely to occur in your scripts. Exceptions refer to errors that are detected as the script is running. One key reason for this is that many scripts rely on user input, and you can't always control the input other users will provide. Well-written scripts, therefore, include error-handling procedures to handle exceptions. Error-handling procedures are written to avoid having a script fail and not provide meaningful feedback.

To handle exceptions, you could use conditional statements to check for certain scenarios, which is analogous to using an `if` statement. You have already encountered some in previous chapters. For example, the existence of a path can be determined in Python using a built-in Python function such as `os.path.exists`. For catalog paths, you can use the `Exists` function to determine whether data exists. For example, the following code determines whether a shapefile exists:

```
import arcpy
from arcpy import env
env.workspace = "C:/Data"
shape_exist = arcpy.Exists("streams.shp")
print shape_exist
```

The `Exists` function can be used for feature classes, tables, datasets, shapefiles, workspaces, layers, and files in the current workspace. The function returns a Boolean value indicating whether the element exists.

Besides determining whether data exists, you can determine whether the data is the right type by using the `Describe` function. For example, if your script requires a feature class, you can use the `datsetType` property to determine whether it is a feature class.

Writing conditional statements for every possible error is tedious. And it is impossible to foresee every error. In the example code earlier in this section, you would have to check the following: (1) whether the workspace is valid, (2) whether there is at least one feature class in the workspace, and (3) whether there is a feature class with at least one feature. This could easily double the code in the script.

There are two strategies to check for errors and report them in a meaningful manner:

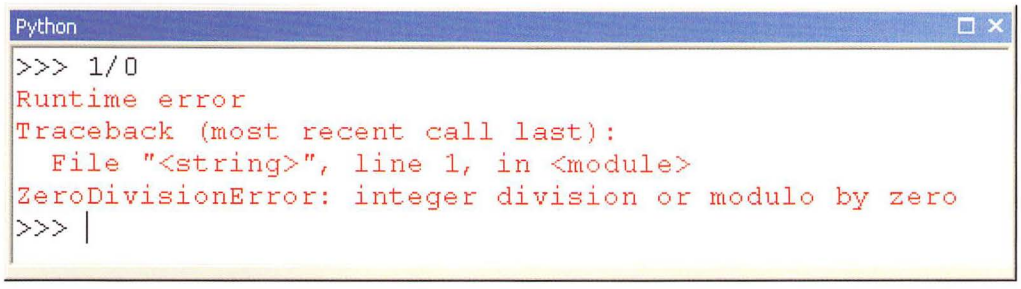
1. Use Python exception objects inside `try-except` statements.

2. Report messages using the ArcPy messaging functions.

A powerful alternative to conditional statements is Python exception objects. When Python encounters an error, it *raises*, or *throws*, an exception. This typically means the script stops running. If such an exception object is not

handled, or *caught*, the script terminates with a runtime error, sometimes also referred to as a *traceback*.

Consider the simple example of trying to divide by zero. In the Python window, it results in a runtime error, as shown in the figure.



The following sections will illustrate how exceptions are raised and how the `try-except` statement can be used to effectively trap errors.

11.7 Raising exceptions

Exceptions are raised automatically when something goes wrong. You can also raise exceptions yourself by using the `raise` statement. You can raise a generic exception using the `raise Exception` statement, as follows:

```
>>> raise Exception
Runtime error
Exception
```

You can also add a specific message, as follows:

```
>>> raise Exception("invalid workspace")
Runtime error
Exception: invalid workspace
```

There are many different types of exceptions. You can view all of them by importing the `exceptions` module and using the `dir` function to list all of them:

```
>>> import exceptions
>>> dir(exceptions)
```

Running this code results in a long printout (not shown in entirety here):

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', ➔
➔ 'BufferError', 'BytesWarning' ...]
```

Each of these exceptions can be used in the `raise` statement. For example:

```
>>> raise ValueError
Runtime error
ValueError
```

This example is a *named* exception—that is, the specific exception is called by name. Named exceptions allow a script to handle specific exceptions in different ways, which can be beneficial. Using the generic `Exception` is referred to as an *unnamed* exception.

A complete description of built-in Python exceptions can be found in the Python documentation. In the documentation, go to Library Reference and browse to section 6, "Built-in Exceptions." It also includes a hierarchy of errors: for example, `ZeroDivisionError` is one of several types of arithmetic errors (`ArithmeticError`).

>>> TIP

The Python documentation is installed in the same program folder as ArcGIS. For a typical installation, you can find the documentation by clicking the Start button on the taskbar, and then, on the Start menu, clicking All Programs > ArcGIS > Python 2.7 > Python Manuals.

Python v2.7 documentation » The Python Standard Library » previous | next | modules | index

6. Built-in Exceptions

Exceptions should be class objects. The exceptions are defined in the module **exceptions**. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the **exceptions** module.

For class exceptions, in a **try** statement with an **except** clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the **raise** statement. If the exception class is derived from the standard root class **BaseException**, the associated value is present as the exception instance's **args** attribute.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be sub-classed to define new exceptions; programmers are encouraged to at least derive new exceptions from the **Exception** class and not **BaseException**. More information on defining exceptions is available in the Python Tutorial under *User-defined Exceptions*.

The following exceptions are only used as base classes for other exceptions.

exception **BaseException**

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that use **Exception**). If **str()** or **unicode()** is called on an instance of this class, the representation of the argument(s) to the instance are returned or the empty string when there were no arguments. All arguments are stored in **args** as a tuple.

11.8 Handling exceptions

Exceptions in a script can be handled using a `try-except` statement. Handling exceptions is often called *trapping*, or *catching*, the exceptions. When an exception is properly handled, the script does not produce a runtime error but instead reports a more meaningful error message to the user. This means the error is trapped, or caught, before it can cause a runtime error.

Consider the following script that divides two user-supplied numbers:

```
x = input("First number: ")
y = input("Second number: ")
print x/y
```

The script will work fine until zero (0) is entered as the second number, resulting in the following error message:

```
First number: 100
Second number: 0
Traceback (most recent call last):
  File " division.py", line 3, in <module>
    print x/y
ZeroDivisionError: integer division or modulo by zero
```

The `try-except` statement can be used to trap this exception and provide additional error handling, as follows:

```
try:
    x = input("First number: ")
    y = input("Second number: ")
    print x/y
except ZeroDivisionError:
    print "The second number cannot be zero."
```

Notice the structure of the `try-except` statement. The first line of code consists of only the `try` statement, followed by a colon (:). Next is a block of indented code with the procedure you want to carry out. Then comes the `except` statement, which includes a specific exception, followed by a colon (:). Next is a block of indented code that will be carried out if the specific exception is raised. The exception `ZeroDivisionError` is a named exception.

In this example, a simple `if` statement might have been more effective to determine whether the value of y is zero (0). However, for more elaborate code, you might need many such `if` statements, and a single `try-except` statement will be sufficient to trap the error.

Multiple `except` statements can be used to catch different named exceptions. For example:

```
try:
    x = input("First number: ")
    y = input("Second number: ")
    print x/y
except ZeroDivisionError:
    print "The second number cannot be zero."
except TypeError:
    print "Only numbers are valid entries."
```

You can also catch multiple exceptions with a single block of code by specifying them as a tuple:

```
except (ZeroDivisionError, TypeError):
    print "Your entries were not valid."
```

In this case, the error handling is not specific to the type of exception and only a single message is printed, no matter what type of error caused the exception. The error message, in this case, is less specific because it describes several exceptions.

The exception object itself can also be called by providing an additional argument:

```
except (ZeroDivisionError, TypeError) as e:
    print e
```

Running this code allows you to catch the exception object itself and you can print it to see what happened rather than printing a custom error message.

It can be difficult sometimes to predict all the types of exceptions that might occur. Especially in a script that relies on user input, you may not be able to foresee all the possible scenarios. So to catch all the exceptions, no matter what type, you can simply omit the exception class from the `except` statement, as follows:

```
try:
    x = input("First number: ")
    y = input("Second number: ")
    print x/y
except Exception as e:
    print e
```

In this example, the exception is unnamed.

The `try-except` statement can also include an `else` statement, similar to a conditional statement. For example:

```
while True:
    try:
        x = input("First number: ")
        y = input("Second number: ")
        print x/y
    except:
        print "Please try again."
    else:
        break
```

In this example, the `try` block of code is repeated in a `while` loop when an exception is raised. The loop is broken by the `break` statement in the `else` statement only when no exception is raised.

One more addition to the `try-except` statement is the `finally` statement. Whatever the result of previous `try`, `except`, or `else` blocks of code, the `finally` block of code will always be executed. This block typically consists of clean-up tasks and could include checking in licenses or deleting references to map documents.

11.9 Handling geoprocessing exceptions

So far, the exceptions raised have been quite general. A Python script can, of course, fail for many reasons that are not specifically related to a geoprocessing tool as the previous examples illustrate. However, because errors related to geoprocessing tools are somewhat unusual in nature, they warrant more attention.

You can think of errors as falling into two categories: geoprocessing errors and everything else. When a geoprocessing tool writes an error message, ArcPy generates a system error. Specifically, when a geoprocessing tool fails to run, it throws an `ExecuteError` exception, which can be used to handle specific geoprocessing errors. It is not one of the built-in Python exception classes, but it is generated by ArcPy and thus the `arcpy.ExecuteError` class has to be used.

Consider this example:

```
import arcpy
arcpy.env.workspace = "C:/Data"
in_features = "streams.shp"
out_features = "streams.shp"
try:
    arcpy.CopyFeatures_management(in_features, out_features)
except arcpy.ExecuteError:
    print arcpy.GetMessages(2)
except:
    print "There has been a nontool error."
```

The Copy Features tool generates an error because the input and output feature classes cannot be the same, as follows:

```
Failed to execute. Parameters are not valid.
ERROR 000725: Output Feature Class: Dataset C:/Data\zip.shp already ➔
➔ exists.
Failed to execute (CopyFeatures).
```

In the example code, the first `except` statement traps any geoprocessing errors, and the second `except` statement traps any nongeoprocessing errors. This example illustrates how both named and unnamed exceptions can be used in the same script. It is important to first check the named exceptions, such as `except arcpy.ExecuteError`, and then the unnamed exceptions. If the unnamed exceptions were checked first, the statement would catch all exceptions, including any `arcpy.ExecuteError` exceptions. This would mean you would never know whether a named exception (that you put in the script) occurred or not.

In larger scripts, it can be difficult to determine the precise location of an error. You can use the Python `traceback` module to isolate the location and cause of an error.

The `traceback` structure is as follows:

```
try:
    import arcpy
    import sys
    import traceback
    <block of code including geoprocessing tools>
except:
    tb = sys.exc_info()[2]
    tbinfo = traceback.format_tb(tb)[0]
    pymsg = "PYTHON ERRORS:\nTraceback info:\n" + tbinfo + "\nError ➔
➔ Info:\n" + str(sys.exc_type) + ":" + str(sys.exc_value) + "\n"
    arcpy.AddError(pymsg)
    msgs = "ArcPy ERRORS:\n" + arcpy.GetMessages(2) + "\n"
    arcpy.AddError(msgs)
    print pymsg + "\n"
    print msgs
```

In this code, two types of errors are trapped: geoprocessing errors and all other types of errors. The geoprocessing errors are obtained using the ArcPy `GetMessages` function. The errors are returned for use in a script tool (`AddError`) and also printed to the standard Python output (`print`). All other types of errors are retrieved using the `traceback` module. Some formatting is applied and the errors are returned for use in a script tool and printed to the standard Python output.

Following is one more example of a `try-except` statement, using the `finally` statement. In this example, a custom exception class is created to handle a license error. A license is checked out in the `try` code block and the license is checked in as part of the `finally` code block. This ensures the license is checked in, no matter the outcome of running the earlier code blocks, as follows:

```
class LicenseError(Exception):
    pass
import arcpy
from arcpy import env
try:
    if arcpy.CheckExtension("3D") == "Available":
        arcpy.CheckOutExtension("3D")
    else:
        raise LicenseError
    env.workspace = "C:/raster"
    arcpy.Slope_3d("elevation", "slope")
except LicenseError:
    print "3D license is unavailable"
except:
    print arcpy.GetMessages(2)
finally:
    arcpy.CheckInExtension("Spatial")
```

Using the `try-except` statement for error trapping is very common. The `ExecuteError` exception class is useful, but in practice, most scripts rely on the simple but effective `try-except` statement without using specific exception classes.

Sometimes, you will see an entire script wrapped in a `try-except` statement. It would look something like the following structure in which the `try` code block could contain hundreds of lines of code:

```
try:
    import arcpy
    import traceback
    ##multiple lines of code here

except:
    tb = sys.exc_info()[2]
    tbinfo = traceback.format_tb(tb)[0]
    pymsg = "PYTHON ERRORS:\nTraceback info:\n" + tbinfo + "\nError ➔
➔ Info:\n" + str(sys.exc_info()[1])
    msgs = "ArcPy ERRORS:\n" + arcpy.GetMessages(2) + "\n"
    arcpy.AddError(pymsg)
    arcpy.AddError(msgs)
```

11.10 Using other error-handling methods

In addition to a `try-except` statement for trapping errors in scripts, several other error-handling methods can be used. Some of them are covered in earlier chapters but warrant further mention here:

- Validating table and field names using the `ValidateTableName` and `ValidateFieldName` functions, respectively (chapter 7).
- Checking for licenses for products using the `CheckProduct` function and for extensions using the `CheckExtension` function (chapter 5).
- Checking for schema locks—many geoprocessing tools will not run properly if schema locks exist on the input.

11.11 Watching for common errors

Following are a number of common errors to look out for when you are scanning your scripts and examining your data.

Common Python code errors

- Simple spelling mistakes
- Forgetting to import modules, such as `arcpy`, `os`, or `sys`
- Case sensitivity—for example, `mylist` versus `myList`
- Paths—for example, using a single backslash (\), such as C:\Data\streams.shp
- Forgetting colons (:) after statements (`for`, `while`, `else`, `try`, `except`)
- Incorrect or inconsistent indentation
- Conditional (==) versus assignment (=) statements

Common geoprocessing-related errors

- Forgetting to determine whether data exists. A small typo in the name of a workspace or feature class will cause a tool to fail. Always double-check that the inputs to a script exist.

- Forgetting to check for overwriting output. The default setting is not to overwrite outputs, so unless this option is specifically cleared, a tool that attempts to overwrite output will not run. A very common scenario is to run a script and it works, but when you run it a second time, it fails—fixing this could be as simple as setting the `overwriteOutput` property of the `env` class to `True`.

- Data is being used in another application. You may be trying to run a script, but it will not run because you are also using the data in ArcMap or ArcCatalog—this is very common because often you are exploring the data that is going to be used in the script. Closing these applications and trying the script again may resolve a script error.

- Not checking the properties of parameters and objects returned by tools. For example, it may sound logical that the Get Count tool produces a count—that is, a number. It actually returns a result object that is printed to the Results window, so you have to use the `getOutput` method to obtain this count. Similarly, distinctions between feature classes and feature layers may seem somewhat trivial, but they may be just the difference between proper tool execution and failure. Carefully examine tool syntax and determine the exact nature of the inputs and outputs.

It is worth noting that many geoprocessing-related errors can be prevented when using script tools. Building a script tool includes validation for preventing invalid parameters. This is covered in chapter 13.

Some of these suggestions may appear rather rudimentary, but the solutions can often be simple if you only knew where to look for them. The syntax of a good Python geoprocessing script is often relatively simple, which is part of the beauty of using Python.

>>> TIP

Remember that geoprocessing scripts don't have to follow Python coding logic alone. They must also obey the rules of the ArcGIS geoprocessing framework.

Points to remember

- Errors in geoprocessing scripts are bound to happen. Although syntax errors are relatively easy to catch, your script may contain other types of errors that prevent proper script execution. Scripts can be made more robust by incorporating error-handling procedures.

- Various debugging methods exist. Relatively simple approaches include carefully examining error messages, adding print statements to the code to review intermediate results, and selectively commenting out code. If these methods are not sufficient to identify and fix errors, a Python debugger can be used such as the PythonWin Debugger. A debugger allows you to carefully step through the code line by line to review error messages and examine the state of variables. Breakpoints can be added to step through larger blocks of code.

- Any debugging procedure will typically identify where the error occurs but not exactly why it occurs. It is therefore good practice to always be aware of common errors, including Python coding errors and ArcGIS geoprocessing errors.

- Basic error-handling procedures include checking whether data exists, determining whether data inputs are the right type, checking for licenses and extensions, and validating table and field names. Typically, an `if` statement is used for this type of error handling.

- It is nearly impossible to anticipate every possible type of errors, and code that checks for such errors would become too cumbersome to write. Whenever something goes wrong in a script, an exception is automatically raised. These exceptions can be trapped using a `try-except` statement. This type of statement makes it possible to identify the type of error or else specific errors. Customized error-handling procedures can be implemented based on the nature of the error. Additional statements, including `else` and `finally`, can be added to the `try-except` statement to ensure efficient error trapping.

- Error messages can be very useful for identifying the nature of the error and how to fix the script. These include both general Python messages and error messages resulting from the ArcPy `ExecuteError` exception class.

Chapter 12
Creating Python functions and classes

12.1 Introduction

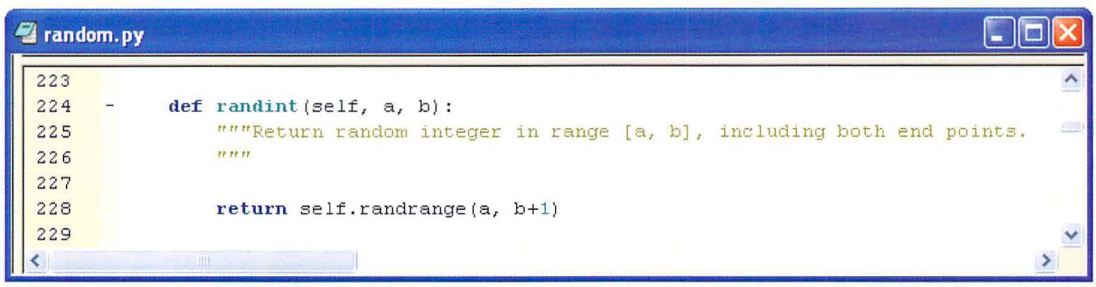
This chapter describes how to create custom functions in Python that can be called from elsewhere in the script or from another script. Custom functions make it easy to reduce the code you have written to carry out procedures. Functions are organized into modules, and modules can be organized into a 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 integer between 1 and 100:

```
import random
x = random.randint(1,100)
print x
```

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. In a typical installation of Python 2.7 as part of the ArcGIS 10.1 installation, the path is: C:\Python27\ArcGIS10.1\Lib\random.py. You can open this script in a Python editor like PythonWin and examine the code. Inside 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 defined 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

```
def <functionname>(<arguments>):
```

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:

```
def printmessage():
    print "Hello world"
```

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:

```
printmessage()
```

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:

```
import arcpy
arcpy.env.workspace = "C:/Data"
fields = arcpy.ListFields("streams.shp")
namelist = []
for field in fields:
    namelist.append(field.name)
```

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:

```
def listfieldnames():
```

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 is, 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:

```
def listfieldnames(table):
```

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:

```
def listfieldnames(table):
    fields = arcpy.ListFields(table)
    namelist = []
    for field in fields:
        namelist.append(field.name)
```

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:

```
def listfieldnames(table):
    fields = arcpy.ListFields(table)
    namelist = []
    for field in fields:
        namelist.append(field.name)
    return namelist
```

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

```
fieldnames = listfieldnames("C:/Data/hospitals.shp")
```

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 called 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 field 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:

```
def printmessage():
    print "Hello world"
```

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:

```
import sys
import os
import helloworld
helloworld.printmessage()
```

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:

```
import sys
print sys.path[0]
```

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

```
import os
print os.getcwd()
```

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:

```
import sys
print sys.path
```

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:

```
C:\Python27\ArcGIS10.1
C:\Python27\ArcGIS10.1\Lib
C:\Python27\ArcGIS10.1\Lib\site-packages
C:\Program Files\ArcGIS\Desktop10.1\bin
C:\Program Files\ArcGIS\Desktop10.1\arcpy
C:\Program Files\ArcGIS\Desktop10.1\ArcToolbox\Scripts
```

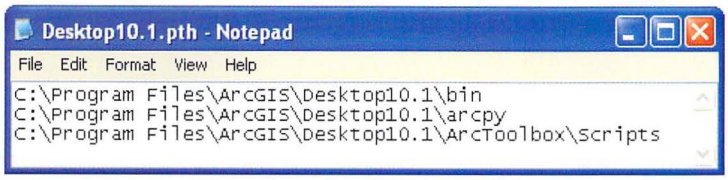
Note: *The list of paths will vary based on how ArcGIS and Python are installed and the versions of the software.*

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:

1. Use a path configuration file (.pth).

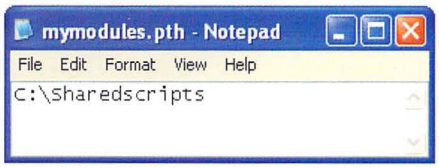
   The most convenient way to access a module in a different folder is to add a path configuration file to a folder that is already part of `sys.path`. It is common to use the site-packages folder—for example, C:\Python27\ArcGIS10.1\lib\site-packages. A path configuration file

has a .pth extension and contains the path(s) that will be appended to `sys.path`. This file can be created using a basic text editor, and each line must contain a single path. When ArcPy is installed as part of the ArcGIS installation, a path configuration file called `Desktop10.1.pth` is placed in the site-packages folder of Python. The file itself looks like the example in the figure.



The path configuration file makes all the modules located in the specific folders available to Python.

You can create a .pth file yourself if you commonly work with scripts that are located in different folders. For example, if the modules you want to import are in the folder C:\Sharedscripts, you would create a .pth file and place it in the Python site-packages folder. The file itself would look like the example in the figure.



Note: *The paths in the path configuration file use a backslash (\) and are not case sensitive.*

2. Append the path using code.

You can also temporarily add a path to your script. For example, if the scripts you want to call are in the folder C:\Sharedscripts, you can use the following code prior to calling the function:

```
sys.path.append("C:/Sharedscripts")
```

Note: *Because this is Python code, you need to use a forward slash (/) for the path.*

The `sys.path.append` statement is a temporary solution meant just so a particular script can call a function in another script.

Note: *A third alternative is to modify the PYTHONPATH variable directly from within the operating system. However, this is somewhat cumbersome and error-prone, and therefore not recommended.*

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:

```
import random
x = random.randint(1,100)
print x
```

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:

```
def printmessage():
    print "Hello world"
print message()
```

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:

```
>>> import hello
"Hello world"
```

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:

```
def printmessage():
    print 'Hello world'
if __name__ == '__main__':
    printmessage()
```

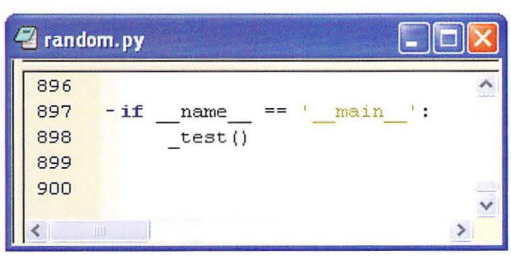
In this case, the test of the module will be run only if the script is run by itself. If you import the script, no code will be run until you call the function.

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:

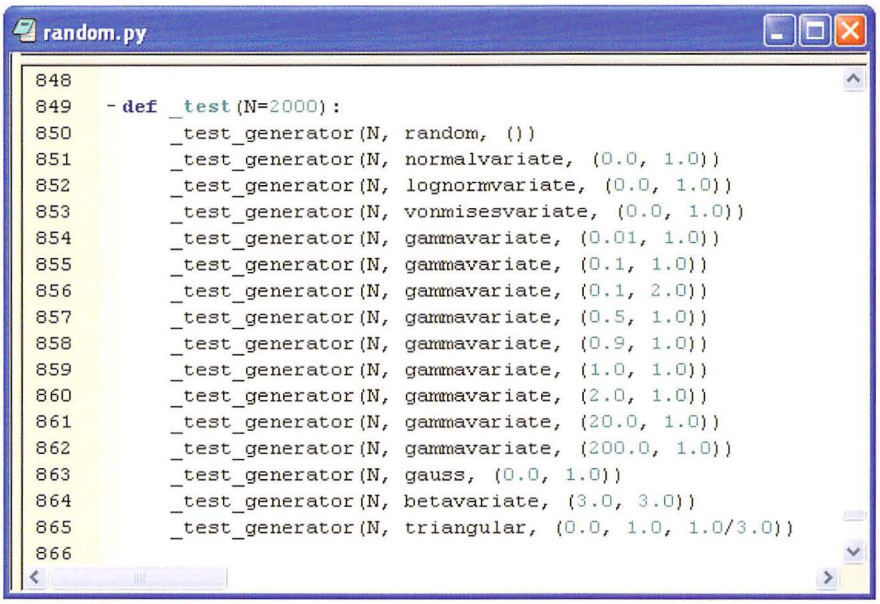
```
import arcpy
import os
def mycooltool(<arguments>):
    <line of code>
    <line of code>
    ...
if __name__ == '__main__':
    mycooltool(<arguments>)
```

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:

```
2000 times random
0.0 sec, avg 0.490386, stddev 0.290092, min 0.000360523, max 0.999743
2000 times normalvariate
0.015 sec, avg -0.0379325, stddev 1.01517, min -3.31413, max 3.54333
2000 times lognormvariate
0.0 sec, avg 1.55066, stddev 1.96947, min 0.0308862, max 24.7307
```

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 interact 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:

```
class Person(object):
    def setname(self, name):
        self.name = name
    def greeting(self):
        print "My name is (0).".format(self.name)
```

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 something 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.

```
me = Person()
```

>>> TIP

The Style Guide for Python Code recommends using the CapitalizedWords, or CapWords, convention for class names—for example, `MyClass`. By contrast, the recommended style for variables, functions, and scripts is all lowercase.

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:

```
me.setname("Abraham Lincoln")
me.greeting()
```

Running this code prints the following:

```
My name is Abraham Lincoln.
```

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:

```
class Parcel(object):
    def __init__(self, landuse, value):
        self.landuse = landuse
        self.value = value

    def assessment(self):
        if self.landuse == "SFR":
            rate = 0.05
        elif self.landuse == "MFR":
            rate = 0.04
        else:
            rate = 0.02
        assessment = self.value * rate
        return assessment
```

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 `value`. 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 where the actual calculation occurs.

Next, take a look at how to use this class. The following code creates an instance of the parcel object:

```
myparcel = Parcel("SFR", 200000)
```

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

```
print "Land use: ", myparcel.landuse
mytax = myparcel.assessment()
print mytax
```

Running this code prints:

```
Land use: SFR
10000.0
```

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 may 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 parcelclass.py and is as follows:

```
class Parcel(object):
    def __init__(self, landuse, value):
        self.landuse = landuse
        self.value = value

    def assessment(self):
        if self.landuse == "SFR":
            rate = 0.05
        elif self.landuse == "MFR":
            rate = 0.04
        else:
            rate = 0.02
        assessment = self.value * rate
        return assessment
```

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

```
import parcelclass
myparcel = parcelclass.parcel("SFR", 200000)
print "Land use: ", myparcel.landuse
mytax = myparcel.assessment()
print mytax
```

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 module is stored as a .py file, but a package is a folder (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 possible 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 file called "__init__.py."

For example, if you had a package you wanted to call "mytools," you would need to have a folder called "mytools" and inside this folder would need to be a file called "__init__.py". The structure of a package called mytools with two modules (`analysis` and `model`) would look as follows:

`~/Python`—a directory in PYTHONPATH

`~/Python/mytools`—a directory for the mytools package

`~/Python/mytools/__init__.py`—package code

`~/Python/mytools/analysis.py`—analysis module

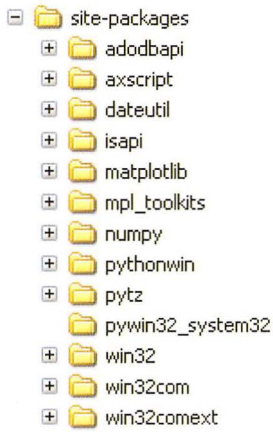
`~/Python/mytools/model.py`—model module

To use the package, your code would look as follows:

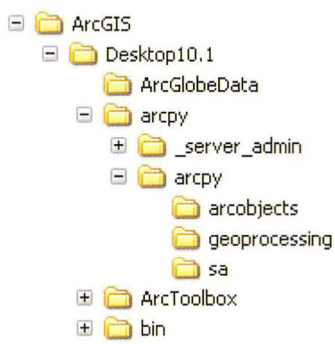
```
import mytools
output = mytools.analysis.<function>(<arguments>)
```

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.

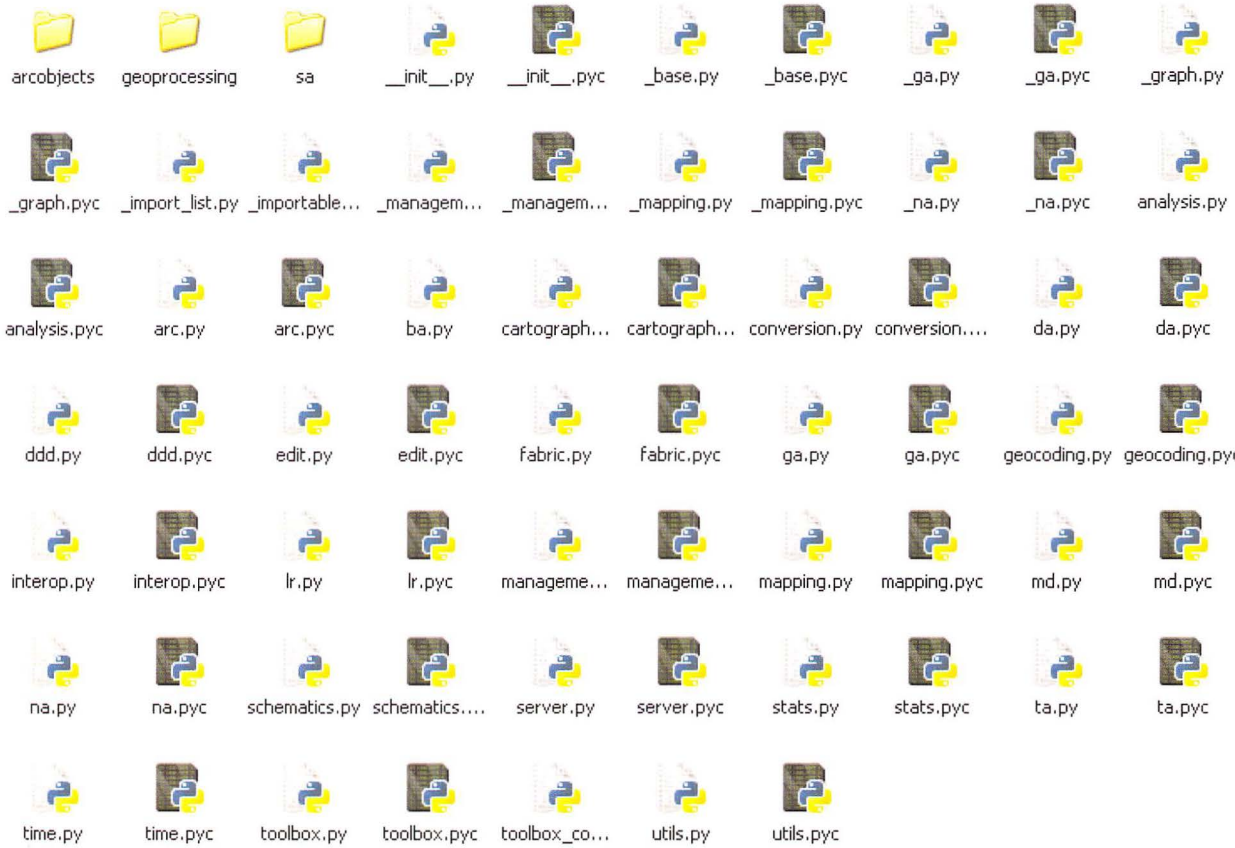


ArcPy is referred to as a site package because a typical ArcGIS installation includes both ArcPy and Python, and the folder where ArcPy is located is automatically recognized by Python through the Desktop10.1.pth file located in the Lib\site-packages folder. Where exactly is ArcPy installed? Typically, 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 arcpy 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, geocoding.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\Desktop10.1\arcpy is added to the PYTHONPATH environment variable in Windows and you can start using ArcPy immediately.

Points to remember

- Custom functions can be created using the `def` statement. The block of code that follows the `def` statement defines what the function actually does. Custom functions can contain arguments, although they are not required.

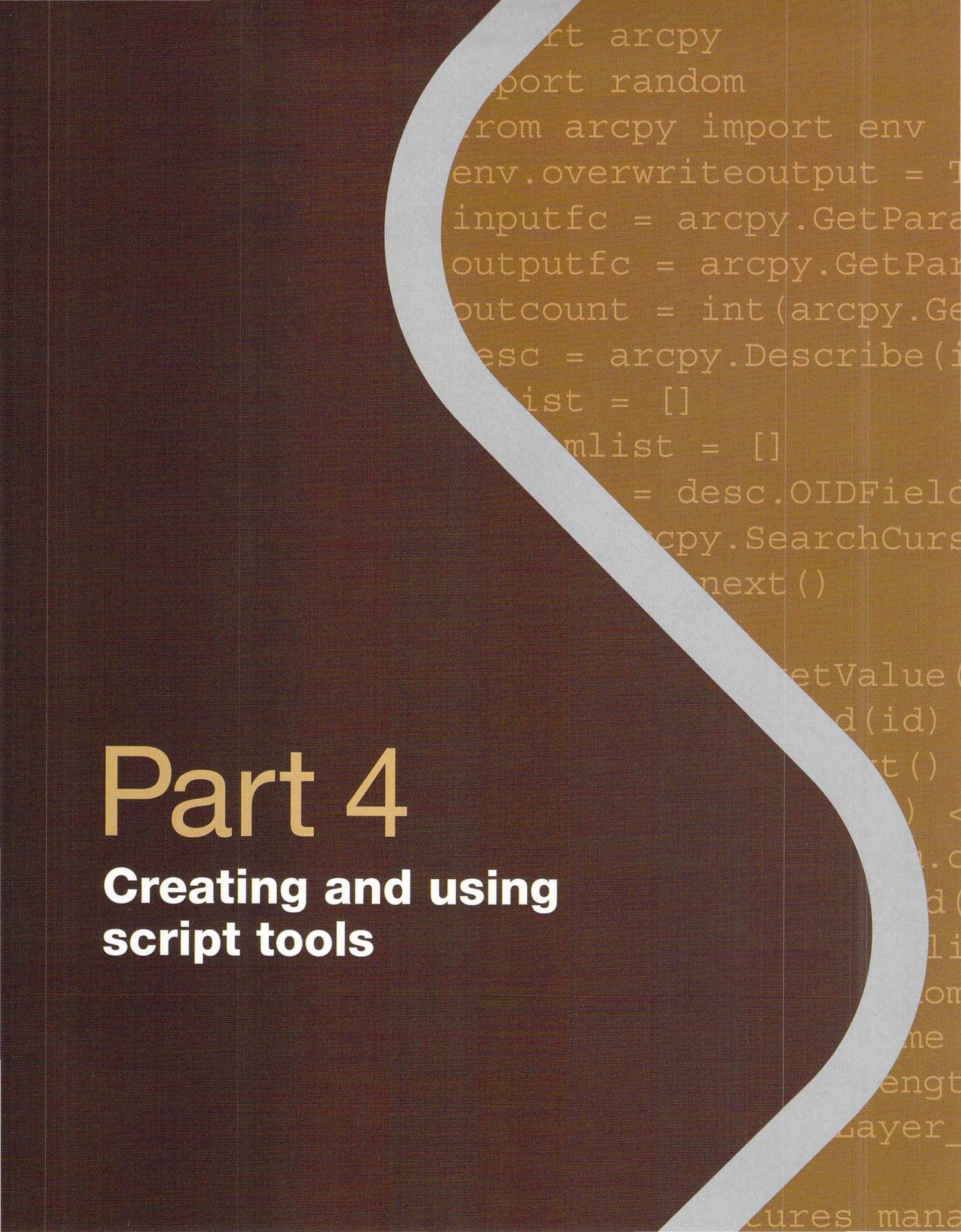
- Custom functions can be called from within the same script or from another script. When calling a function from another script, you import the script that contains the function as a module. A custom module is therefore a regular .py file that contains at least one custom function.

- To distinguish between running a script by itself and importing it as a module into another script, you can use the `if __name__ == 'main__':` statement.

- When importing modules, you cannot use paths, and modules (scripts) need to be located in the same folder as the script importing the module(s) or in the folders included in the PYTHONPATH environment variable. As needed, you can permanently add a path by using a .pth file in the site-packages directory or temporarily add a path in your script using the `sys.path.append` statement.

- Custom classes can be created to make it easier to group together functions and variables related to a particular item. Classes can be called from within the same script or from another script.

- As your collection of custom functions and classes grows, you can consider making it a package, similar to the ArcPy site package.



Part 4

Creating and using script tools

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 a stand-alone 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.

2. *As a 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.

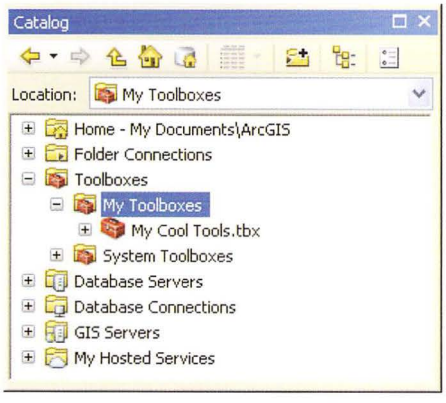
13.3 Steps to creating a tool

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 ArcGIS. 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 toolboxes, 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-click inside ArcToolbox and click Add Toolbox, which allows you to browse for a toolbox 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 logical location to organize custom toolboxes, but they can also be located in any folder where 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 geodatabase 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.

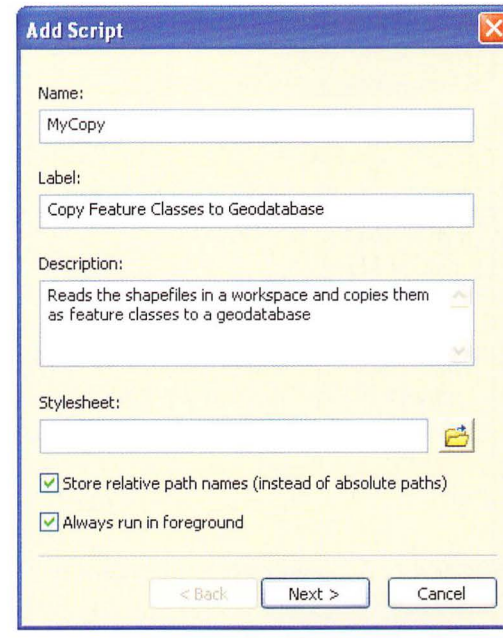
The Add Script wizard has three panels. The first panel is used to specify the script name, label, and description. The second panel is used to specify the actual script file (.py), including its path. The third panel is used to specify the tool's parameters. Each of these panels is reviewed here in detail.

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:

```
# Python script: copyfeatures.py
# This script copies all feature classes from a workspace into
# a file geodatabase.

# Import the ArcPy package.
import arcpy
import os

# Set the current workspace.
from arcpy import env
env.workspace = "C:/Data"

# Create a list of feature classes in the current workspace.
fclist = arcpy.ListFeatureClasses()

# Copy each feature class to a file geodatabase - keep the same
# name but use the basename property to remove any file
# extensions, including .shp.
for fc in fclist:
    fcdesc = arcpy.Describe(fc)
    arcpy.CopyFeatures_management(fc, os.path.join("C:/Data/study.gdb/", ➔
➔ fcdesc.basename))
```

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 panel of the wizard, as shown in the figure. ➔

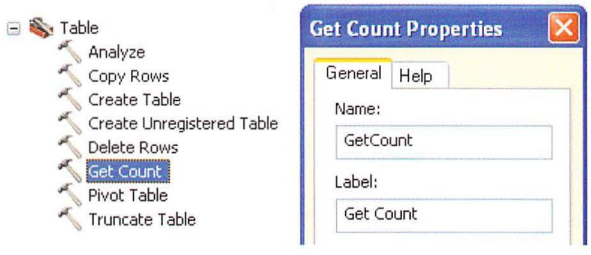


The first panel of the wizard is used to specify the tool name, label, description, and style sheet as follows:

- 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 field to provide a customized description. The text is automatically used to provide the contents of the Help panel on a tool dialog box.

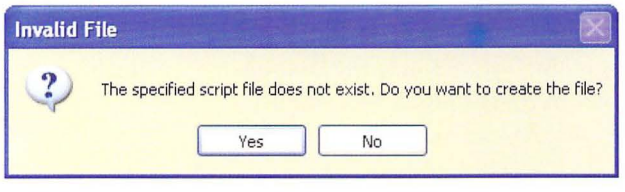
- An optional style sheet can be selected. If none is selected, the default style sheet is used. Style sheets are used to control the properties of items on a tool dialog box. A style sheet provides style and layout information. All the system tools in ArcToolbox use the default style sheet. Typically, 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 ArcGIS for Desktop application while the tool is running.) Some scripts require foreground processing—for example, mapping scripts, which use the `CURRENT` keyword to obtain the active map document in ArcMap. For other scripts, selecting foreground or background processing is a matter of preference.

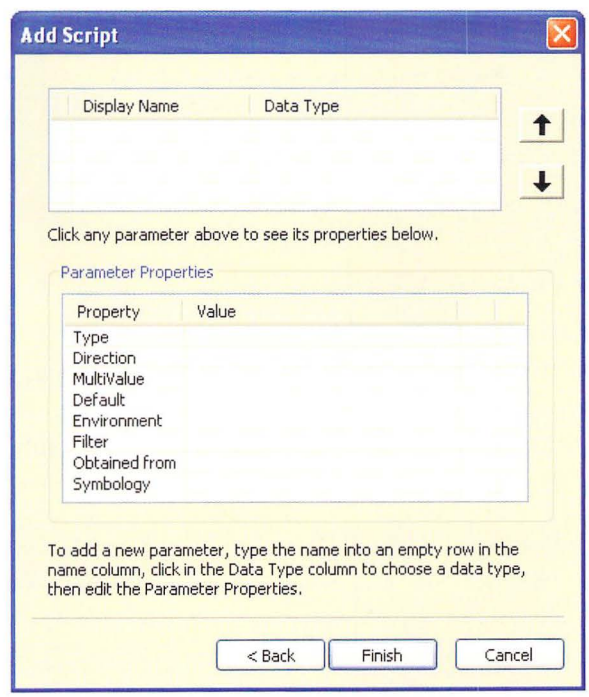
In the second panel of the Add Script wizard, you can set the following:

- 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.



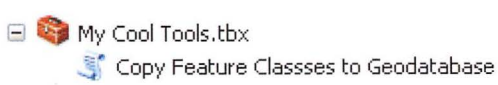
- The "Show command window when executing script" check box, which is cleared by default. When the box is selected, an additional window appears during tool execution to show messages that are not part of the regular geoprocessing messages but are written to the standard output for Python. For example, the Python `print` statement writes 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.

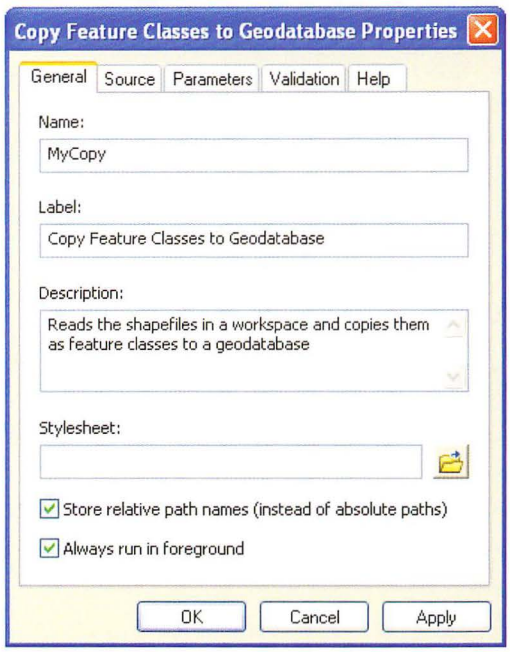


The third panel of the Add Script wizard is used to specify the tool parameters. By default, no parameters are listed, but most tools need at least one input parameter and one output parameter. The top half of the panel allows you to create parameters and the bottom half allows you to set the properties for each of these parameters. Setting parameters is covered in detail later in this chapter, including parameter properties. ➔

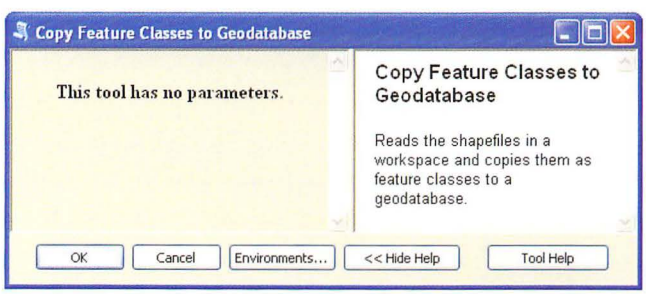
To complete the Add Script wizard, click the Finish button. Completing this wizard adds a tool to your custom toolbox.



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 on the dialog box are Validation and Help. These tabs are revisited later in this chapter.



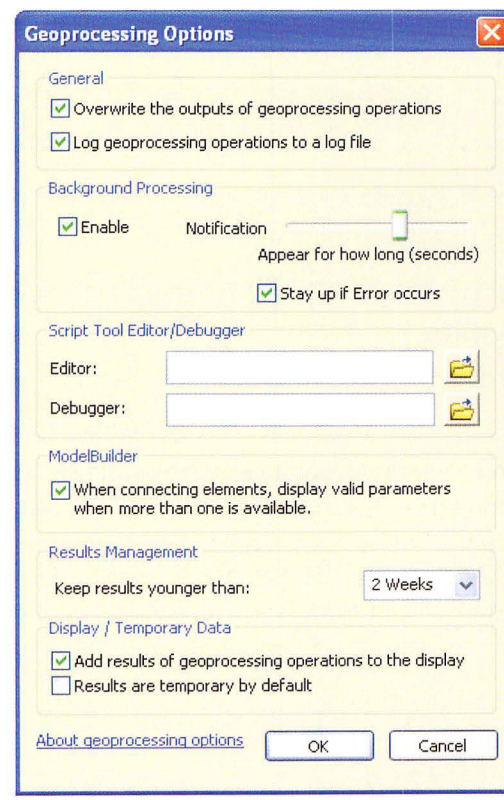
Your new tool can be accessed just like a regular tool. Right-click the tool and click Open or simply double-click the tool in its toolbox. The tool dialog box seems empty at this point because no parameters were set in the third panel of the Add Script wizard. The Help panel on the right shows the tool's 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 critical steps in creating tools 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 leave the Python editor open while you do it. You can open a script from within the Python editor, but there is also a shortcut in ArcGIS. Right-click the tool in the toolbox 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 application, click Geoprocessing > Geoprocessing Options. ➔



To select PythonWin as the editor, browse to the location of the PythonWin application. Typically, this application is located at C:\Python27\ArcGIS10.1\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

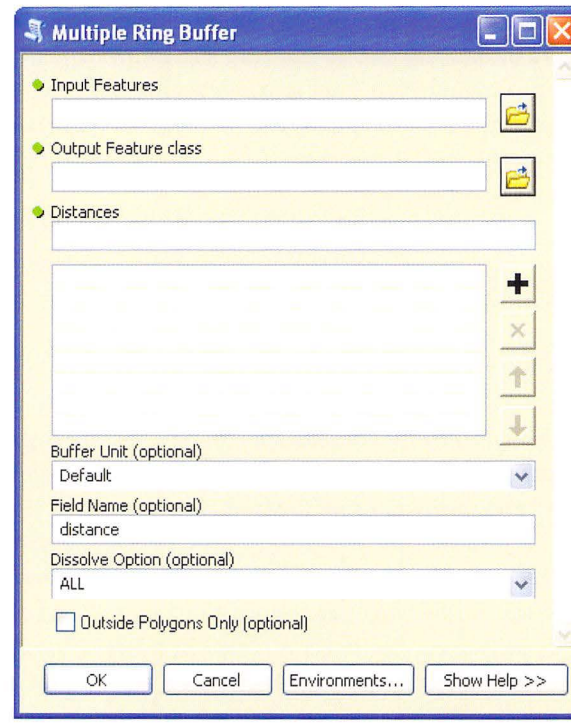
As you have seen in several earlier chapters, all geoprocessing tools have parameters. Parameter values are set on the tool dialog box. In a stand-alone script, parameters are typically set within the script unless user input is expected. For tools, parameters can be set using the tool dialog box. When 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:

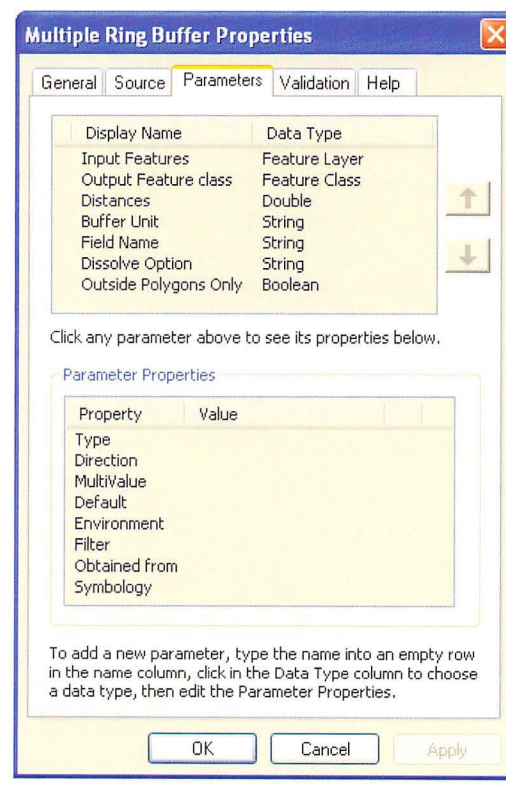
- Including code in the script to receive the parameter values
- Setting up the parameters in the tool's properties

Next, you will try this out by using one of the built-in tools, the Multiple Ring Buffer 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 Multiple Ring Buffer 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 its properties.*





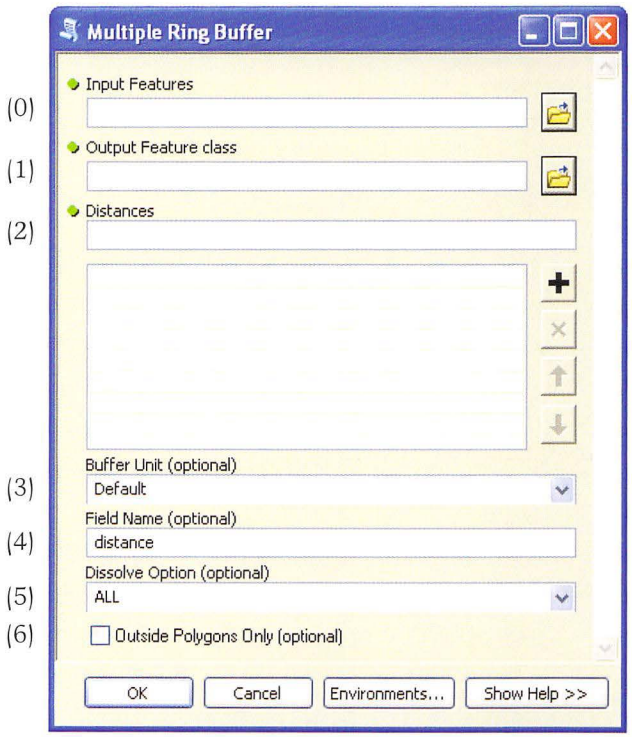
Once a user specifies the parameter values on the Multiple Ring Buffer tool dialog box, the tool can be run. Once the tool is run, the user-specified parameter values will be passed to the script. Take a look at the script's code to see how these parameter values are received by the script. The example 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.

```
import arcgisscripting
import os
import sys
import types
import locale

gp = arcgisscripting.create(9.3)

#Define message constants so they may be translated easily
msgBuffRings  = gp.GetIDMessage(86149) #"Buffering distance "
msgMergeRings = gp.GetIDMessage(86150) #"Merging rings..."
msgDissolve   = gp.GetIDMessage(86151) #"Dissolving overlapping boundaries..."

def initiateMultiBuffer():


    # Get the input argument values
    # Input FC
    input           = gp.GetParameterAsText(0)
    # Output FC
    output          = gp.GetParameterAsText(1)
    # Distances
    distances       = gp.GetParameter(2)
    # Unit
    unit            = gp.GetParameterAsText(3)
    if unit.lower() == "default":
        unit = ""
    # If no field name is specified, use the name "distance" by default
    fieldName       = checkFieldName(gp, gp.GetParameterAsText(4), os.path.dirname(output))
    #Dissolve option
    dissolveOption  = gp.GetParameterAsText(5)
    # Outside Polygons
    outsidePolygons = gp.GetParameterAsText(6)
    if outsidePolygons.lower() == "true":
        sideType = "OUTSIDE_ONLY"
    else:
        sideType = ""

    createMultiBuffers(gp, input, output, distances, unit, fieldName, dissolveOption, sideType)
```

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

```
<variable> = arcpy.GetParameterAsText(<index>)
```

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).



```
def initiateMultiBuffer():

    # Get the input argument values
    # Input FC
    input           = gp.GetParameterAsText(0)
    # Output FC
    output          = gp.GetParameterAsText(1)
    # Distances
    distances       = gp.GetParameter(2)
    # Unit
    unit            = gp.GetParameterAsText(3)
    if unit.lower() == "default":
        unit = ""
    # If no field name is specified, use the name "distance" by default
    fieldName       = checkFieldName(gp, gp.GetParameterAsText(4), os.path.dirname(output))
    #Dissolve option
    dissolveOption  = gp.GetParameterAsText(5)
    # Outside Polygons
    outsidePolygons = gp.GetParameterAsText(6)
    if outsidePolygons.lower() == "true":
        sideType = "OUTSIDE_ONLY"
    else:
        sideType = ""

    createMultiBuffers(gp, input, output, distances, unit, fieldName, dissolveOption, sideType)
```

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:

```
outsidePolygons = gp.GetParameterAsText(6)
if outsidePolygons.lower() == "true":
    sideType = "OUTSIDE_ONLY"
else:
    sideType = ""
```

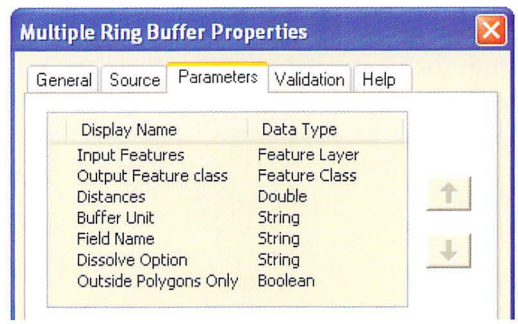
The Outside Polygons Only parameter on the tool dialog box is a Boolean value of True or False. These values are converted to strings, and as a result, the conditional statement uses the string value `"true"` instead of the Boolean value `True`.

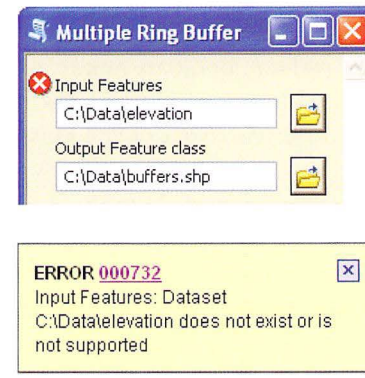
The Distances parameter on the tool dialog box is received by the script using the `GetParameter` function. This is because the parameter consists of a list of values (doubles) instead of a single value. The `GetParameter` function reads this list as an object.

> ***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 tool, 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 box 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) and prevent the tool from running. This built-in error-checking mechanism prevents users 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.

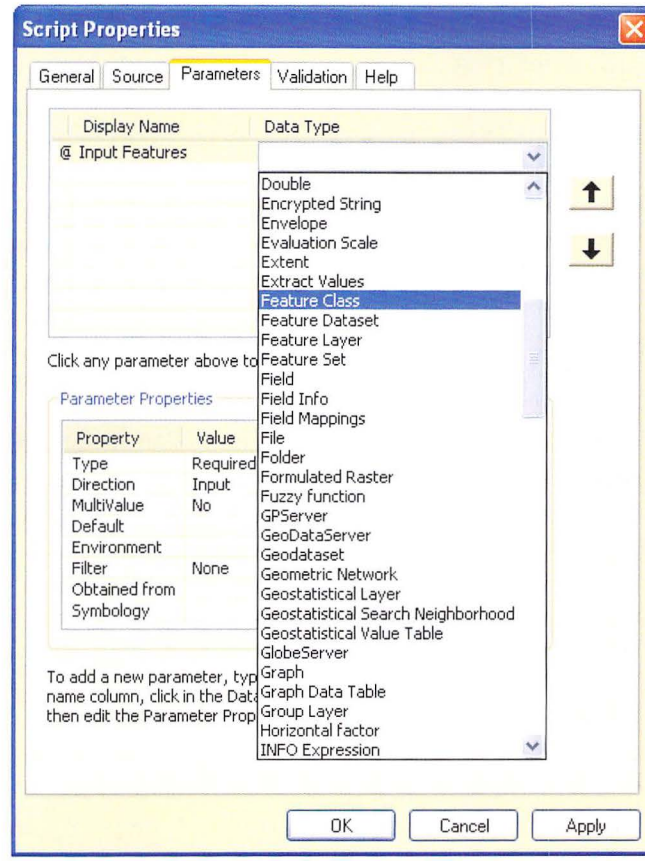
13.6 Setting tool parameters

Tool parameters can be set in the Add Script wizard when creating the tool. They can also be edited after the tool has been created by accessing the tool's properties dialog box. Setting 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

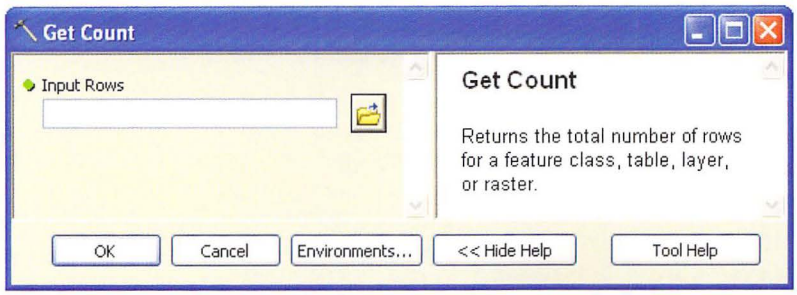
There are three choices for Type: Required, Optional, and Derived. Required means that a parameter value needs to be specified for a tool to run. Optional means that a value is not required for a script to run. Typically, this means that a default value is specified. Derived parameters are used for output parameters only and do not appear on the tool dialog box.

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

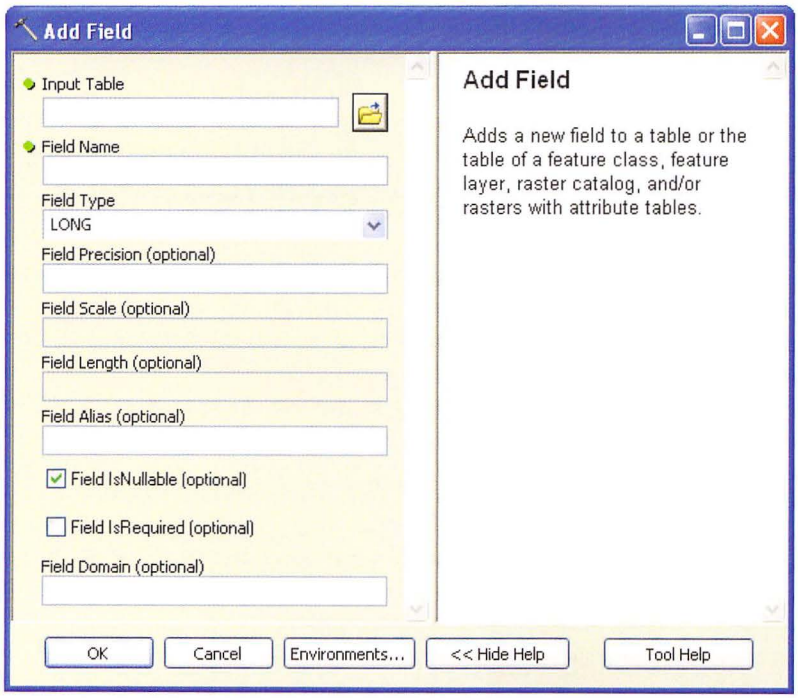
- When a tool outputs a single value instead of a dataset. Such a single value is often referred to as a *scalar*.
- When a tool creates outputs using information from other parameters.
- When a tool modifies an input without creating a new output.

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.

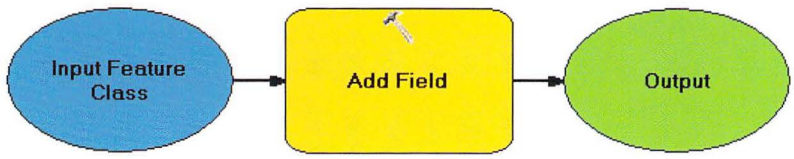


The Add Field tool adds a new field to an input table. The input table is a required parameter, and so is the name of the new field. The rest of the parameters are optional, as shown in the figure.



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).

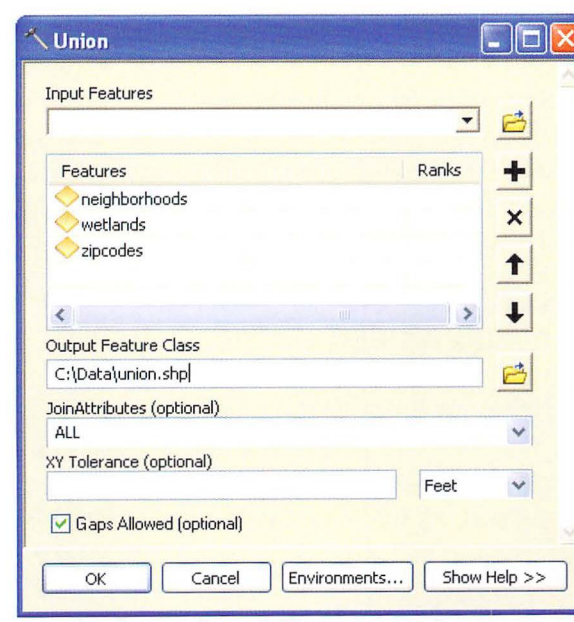
Direction

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

```
import arcpy
input = arcpy.GetParametersAsText(0)
inputList = input.split(";")
```

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. ValueTable 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 framework. If the Environment property is specified, the default property will be ignored—so you need to specify one or the other.

Filter

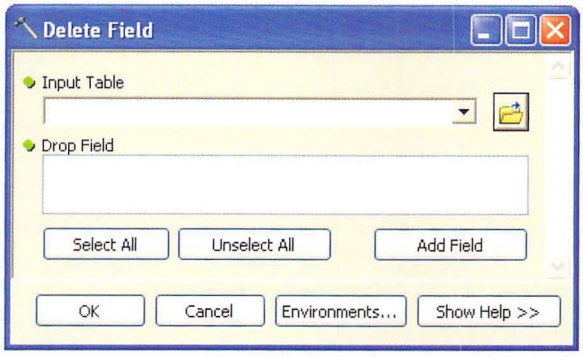
The Filter property allows you to limit the values of dataset types to be entered for a parameter. There are a number of filter types, and the type depends on the data type of the parameter. The different filter types are Value List, Range, Feature Class, File, Field, and Workspace.

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.

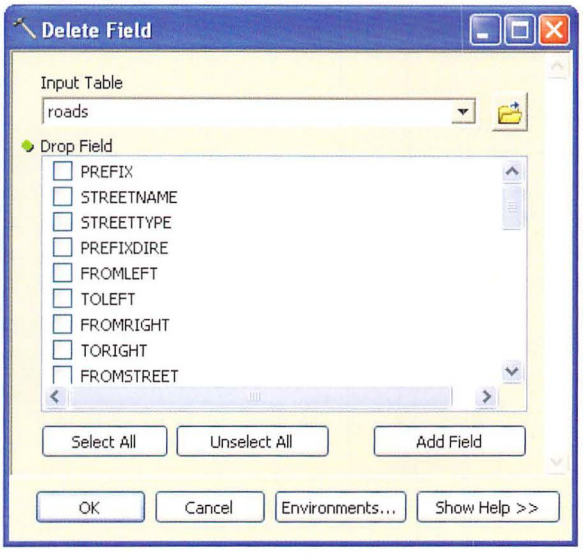
The different filter types provide specific control of which parameters are valid inputs. Carefully setting the filter type will improve the robustness of the tool. The different filter types are discussed in more detail in ArcGIS Desktop Help, under the topic "Setting script tool parameters."

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. In 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 table 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. ➔

Property	Value
Type	Required
Direction	Output
MultiValue	No
Default Environment	
Filter	None
Obtained from	
Symbology	C:\Data\example.lyr

Notice that setting the Symbology property does not control whether 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

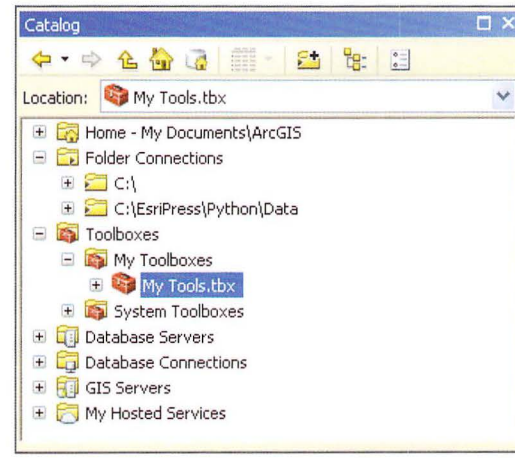
The following example illustrates the steps to convert a stand-alone script to a script tool. The following stand-alone script was introduced at the beginning of the chapter. The script creates a list of all the shapefiles in a workspace and copies them to a geodatabase. For the purpose of this example, the script is located in the C:\Sharedscripts folder. The script is as follows:

```
# Python script: copyfeatures.py
# This script copies all feature classes from a workspace to
# a file geodatabase.

# Import the ArcPy package.
import arcpy
import os

# Set the current workspace.
from arcpy import env
env.workspace = "C:/Data"

# Create a list of feature classes in the current workspace
fclist = arcpy.ListFeatureClasses()

# Copy each feature class to a file geodatabase - keep the same
# name but use the basename property to remove any file
# extensions, including .shp
for fc in fclist:
    fcdesc = arcpy.Describe(fc)
    arcpy.CopyFeatures_management(fc, "C:/Data/study.gdb/" + fcdesc.➔
➔ basename)
```

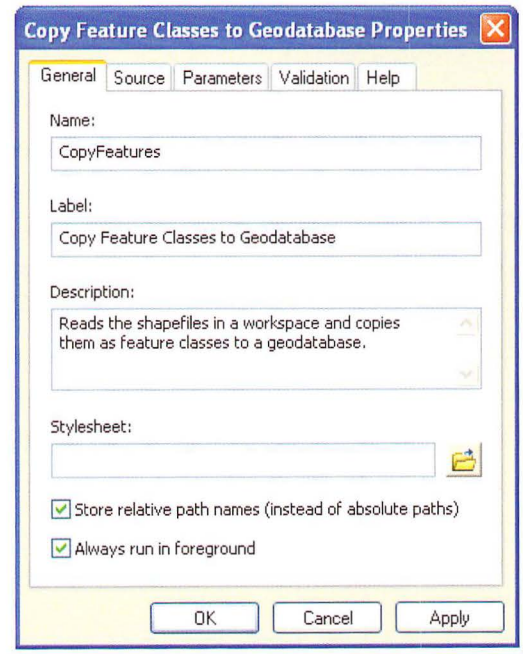
Two workspaces hard-coded into the script have to be modified to parameters using the `GetParameterAsText` function. The revised script (without comments) is as follows:

```
import arcpy
import os
from arcpy import env
env.workspace = GetParameterAsText(0)
outgdb = GetParameterAsText(1)
fclist = arcpy.ListFeatureClasses()
for fc in fclist:
    fcdesc = arcpy.Describe(fc)
    arcpy.CopyFeatures_management(fc, os.path.join(outgdb, fcdesc.➔
➔ basename))
```

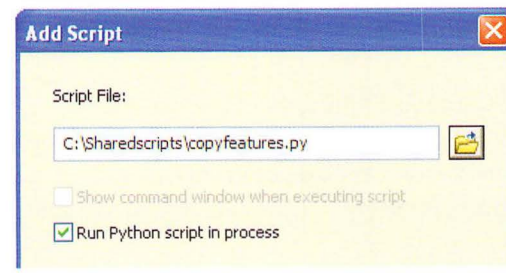
A custom toolbox is created in the My Toolboxes folder. This empty toolbox can be dragged to ArcToolbox. ➔



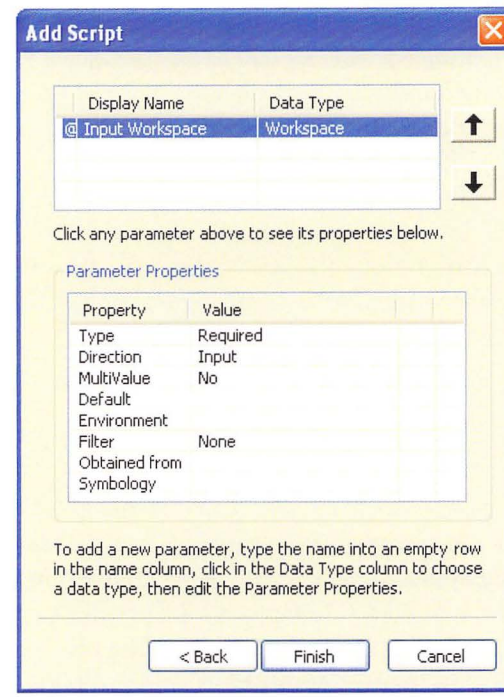
To create a new script tool, right-click the custom toolbox and click Add > Script. In the first panel on the Add Script dialog box, specify the name, label, and description of the script tool. Select the check boxes for storing relative path names and foreground processing. ➔



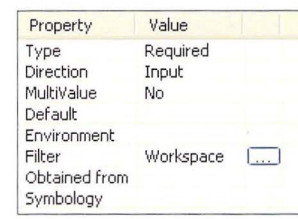
In the second panel on the Add Script wizard, browse to the location of the script file—in this case, C:\Shared-scripts\copyfeatures.py. Leave the other settings to their defaults. ➔

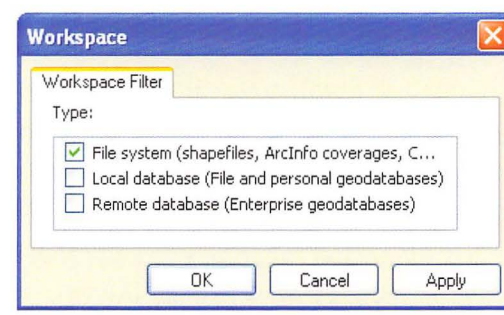


In the third panel on the Add Script wizard, you can specify the parameters. For the first parameter, enter the name **Input Workspace** and select Workspace for Data Type. ➔

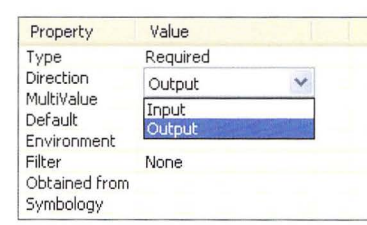


Under Parameter Properties, click the Filter property. Under Value, select Workspace. This brings up the Workspace filter. Under Type, clear the check boxes for "Local database" and "Remote database." ➔

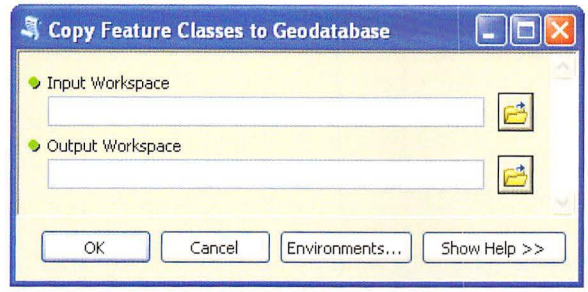




For the second parameter, enter the name **Output Workspace** and select Workspace for Data Type. Under Parameter Properties, set Direction to Output. ➔



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 shapefiles 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.

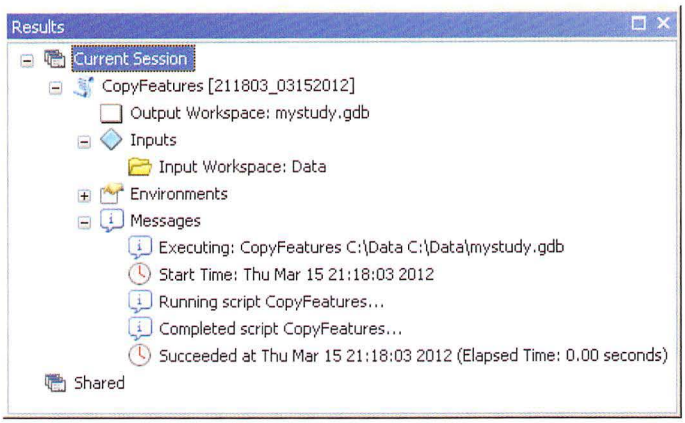
Tool behaviors can be set on the Validation tab on the Script Properties dialog box. In the Validation panel, you can use Python code that uses a Python class called `ToolValidator`. The `ToolValidator` class controls how the tool dialog box is changed based on user input. It is also used to describe the output data the tool produces, which is important for using tools in ModelBuilder.

The `ToolValidator` class was introduced in ArcGIS 9.3, providing more possibilities for creating robust tools. A detailed description of customizing tool behavior is not provided here. Details on the `ToolValidator` class can be found in ArcGIS 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:

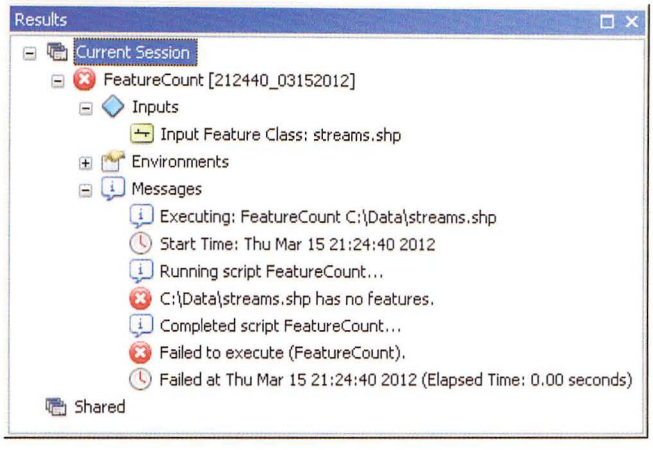
- `AddMessage`—for general information messages (severity = 0)
- `AddWarning`—for warning messages (severity = 1)
- `AddError`—for error messages (severity = 2)
- `AddIDMessage`—for both warning and error messages
- `AddReturnMessage`—for all messages, independent of severity

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:

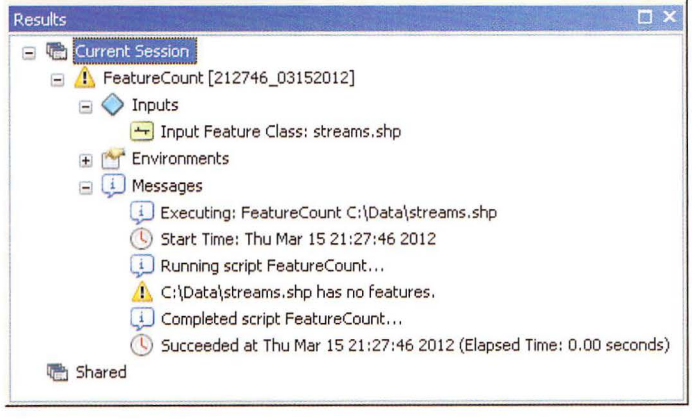
```
import arcpy
fc = arcpy.GetParameterAsText(0)
result = arcpy.GetCount_management(fc)
fcount = int(result.getOutput(0))
if fcount == 0:
    arcpy.AddError(fc + " has no features.")
else:
    arcpy.AddMessage(fc + " has " + str(fcount) + " features.")
```

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 also 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

```
AddIDMessage(message_type, message_ID, {add_argument1}, {add_argument2})
```

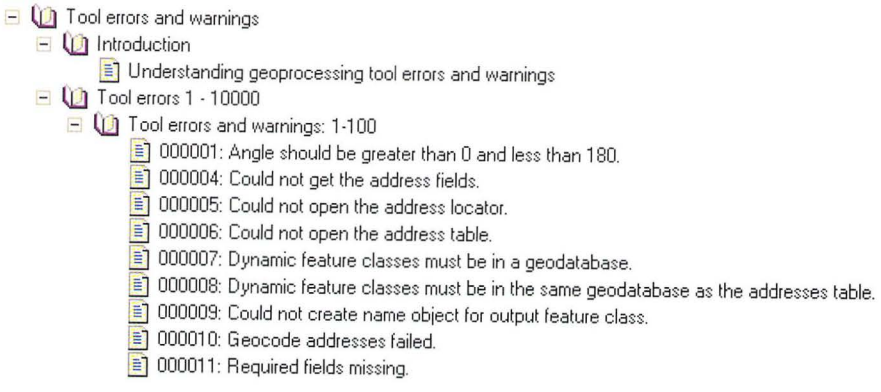
The message type can be set to `Error`, `Informative`, or `Warning`. The message ID number indicates the specific Esri system message. Depending on the message, additional arguments may be necessary. In the following example code, an error message, with the message ID number 12, is produced if the output feature class already exists:

```
import arcpy
infc = arcpy.GetParameterAsText(0)
outfc = arcpy.GetParameterAsText(1)
if arcpy.Exists(outfc):
    arcpy.AddIDMessage("Error", 12, outfc)
else:
    arcpy.CopyFeatures_management(infc, outfc)
```

The syntax of error message 12 is

```
000012 : <value> already exists
```

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 Results window.

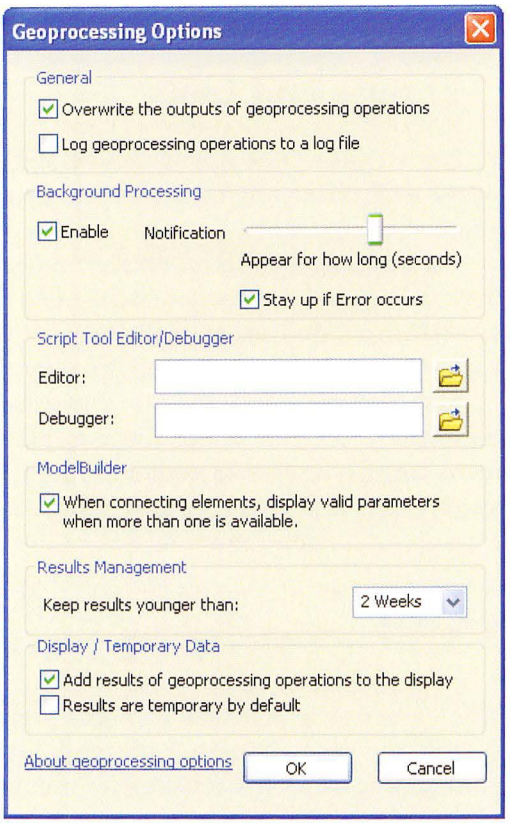
Standard practice is to write a message-handling routine that writes messages to both the interactive interpreter and the geoprocessing environment, using a `print` statement for the former and ArcPy functions such as `AddError`, `AddWarning`, and `AddMessage` for the latter.

13.11 Customizing tool progress information

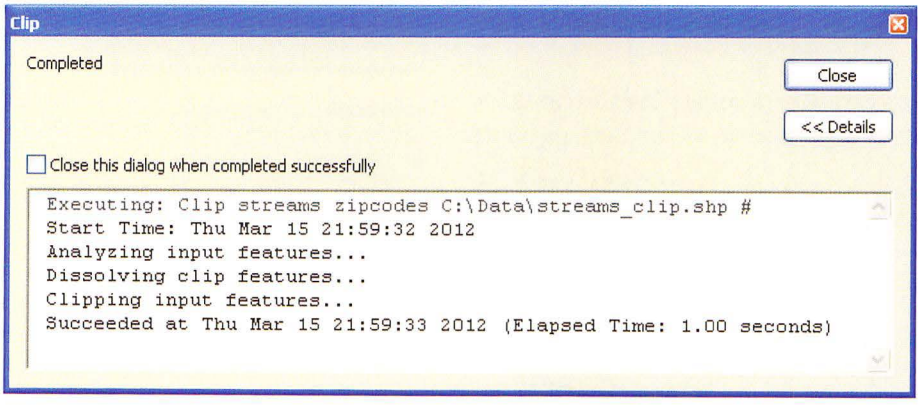
When a tool runs, information on its progress can take several forms. By default, the geoprocessing framework in ArcGIS uses background processing. This means you can continue to use an application while the geoprocessing operations run in the background. During background processing, a progress bar 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.

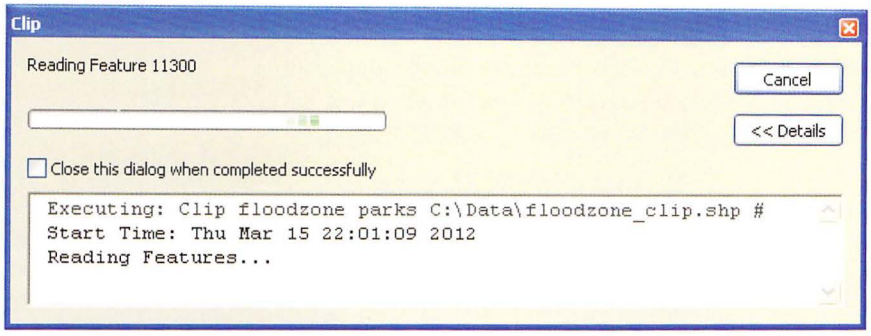


Whether a tool runs as background or foreground processing can be controlled at the application level using the Geoprocessing Options dialog box. For a script tool, background or foreground processing can also be controlled as part of the script tool properties, as described in section 13.3. The appearance of the progress dialog box, which appears during foreground processing, can be controlled using the ArcPy progressor functions. These

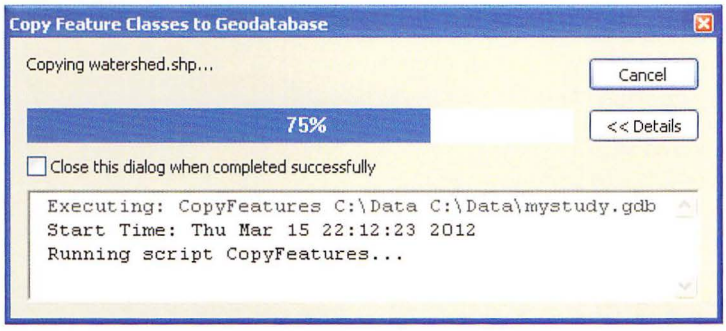
functions also have an effect on the Results window. The ArcPy progressor functions include the following:

- `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. In 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 label 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

```
SetProgressor(type, {message}, {min_range}, {max_range}, {step_value})
```

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 moved by an increment. The script is as follows:

```
import arcpy
import os
from arcpy import env
env.workspace = arcpy.GetParameterAsText(0)
outworkspace = arcpy.GetParameterAsText(1)
fclist = arcpy.ListFeatureClasses()
fcount = len(fclist)
arcpy.SetProgressor("step", "Copying shapefiles to geodatabase...", 0, ➔
➔ fcount, 1)
for fc in fclist:
    arcpy.SetProgressorLabel("Copying " + fc + "...")
    fcdesc = arcpy.Describe(fc)
    outfc = os.path.join(outworkspace, fcdesc.baseName)
    arcpy.CopyFeatures_management(fc, outfc)
    arcpy.SetProgressorPosition()
arcpy.ResetProgressor()
```

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 scripts, it is not known in advance how many feature classes, fields, 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 communication 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 modules 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.

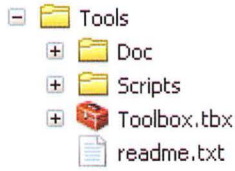
Chapter 14
Sharing tools

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 e-mailed. 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 largely 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 available on a local area network may be the most efficient. To make tools available to a broad community of users, the use of a ZIP file is likely the most convenient.

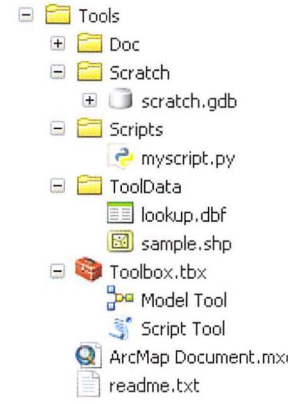
A number of other considerations will influence how to share tools, including where the input and output data is located and what products and extensions the tools require. In the ZIP file method, for example, any tool data also has to be packaged with the tool because a typical user will not have access 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 should therefore include logic to check for the necessary product levels (ArcGIS for Desktop Basic, ArcGIS for Desktop Standard, or ArcGIS for Desktop Advanced) and extension licenses (ArcGIS 3D Analyst, ArcGIS Spatial Analyst, 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 user'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.

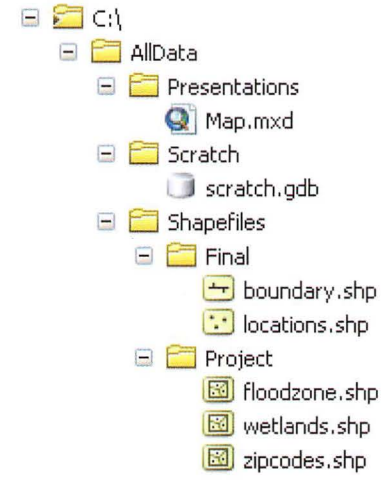
Tool documentation is provided in the Doc folder. Documentation can consist of a Microsoft Word document (.doc or .docx) or PDF file (.pdf) that provides instructions, external Help compiled HTML files (.chm) that are referenced by tools or toolboxes, and XML style sheets that replace the default tool 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

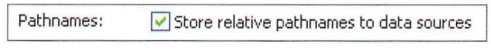
Paths are an integral part of working with data and tools. When tools are shared, paths become particularly important, because without proper documentation of where files are located, the tools will not run.

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 shapefiles located in the C:\alldata\shapefiles\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 shapefiles locations.shp and floodzone.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, alldata\shapefiles-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:

```
C:\alldata\presentations\Map.mxd
```

If the shapefile locations.shp is added to this map document, the absolute path is converted to the following relative path:

```
.\..\shapefiles\final\locations.shp
```

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 shapefiles\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. ➔

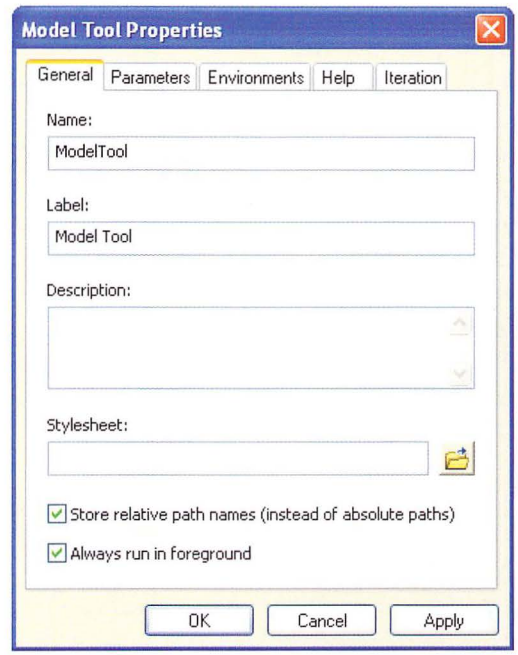
In script tools, relative paths are enabled in the Add Script wizard when creating a script tool, or on the script tool properties dialog box for existing tools. ➔

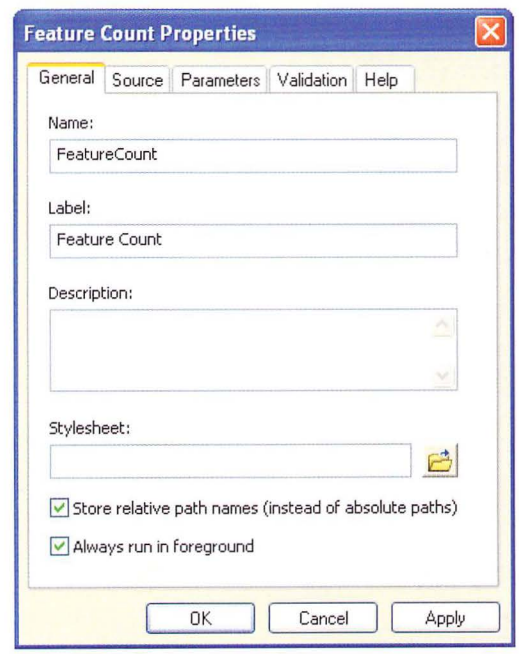
Relative paths are relative to the current folder where the toolbox file is located. When relative paths are enabled, it applies to the script files, datasets used for the default value properties, files referenced in the tool documentation, layer files used for the symbology properties, compiled Help files, and style sheets.

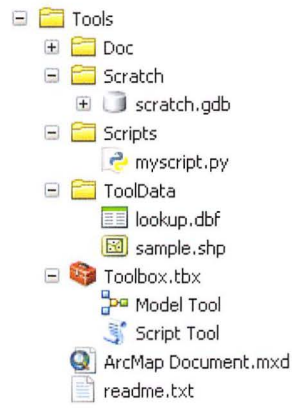
It is important to recognize that paths within the script are not converted because ArcGIS has no reliable way to examine and modify the script code. Therefore, if a script uses absolute paths, they are not converted to relative paths when relative paths are enabled for the script tool.

>>> 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. In 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 particular 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:

```
scriptpath = sys.path[0]
```

Or:

```
scriptpath = os.getcwd()
```

Running this code results in a string with the complete 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 module `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 enabled 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 also the author of the lookup.dbf file and the creator of the ToolData folder. However, the absolute path should 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:

```
import arcpy
import os
import sys
scriptpath = sys.path[0]
toolpath = os.path.dirname(scriptpath)
tooldatapath = os.path.join(toolpath, "ToolData")
datapath = os.path.join(tooldatapath, "lookup.dbf")
```

Notice that two elements are hard-coded into the script: the actual file name of the tool data (lookup.dbf) and the folder where the tool data is located (ToolData). These are both created by the author of the tool and 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:

```
import arcpy
import os
import sys
from arcpy import env
scratchws = env.scratchWorkspace
scriptpath = sys.path[0]
toolpath = os.path.dirname(scriptpath)
if not env.scratchWorkspace:
    scratchws = os.path.join(toolpath, "Scratch/scratch.gdb")
```

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 extension. This single file contains all the files 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 ArcGIS 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 traditional approach 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 tool 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 files. In 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 deleted—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. Embedding multiple scripts can be confusing to users because it becomes less transparent how the scripts work. In addition, embedding scripts was introduced in ArcGIS 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 specific concepts that may 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.

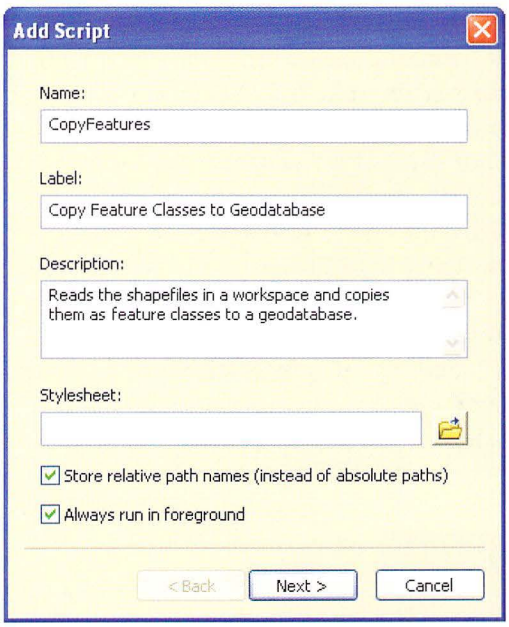
- A style sheet can be used, providing additional control of the look and feel of the tool dialog box.

- A compiled Help file can be created and referenced to appear as the tool reference page.

There are other ways to provide documentation as well, within the script itself or on disk, as follows:

- By commenting code. Good scripts contain detailed comments, which explain how a script works. Not all users of a script tool may look at the script, but for those who do, comments can be very informative. Comments are located inside the actual script files.

- Through separate documentation located on disk—for example, in the Doc folder. Documentation files can be provided as .doc, .docx, .pdf, or other file types. This documentation typically includes a more detailed explanation of the tools and any relevant background concepts.

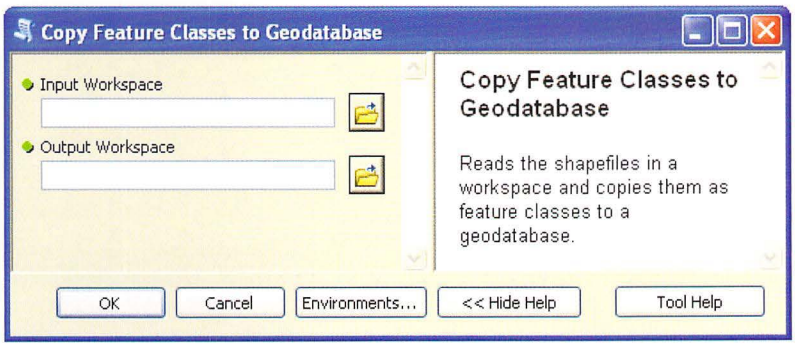


Following is a more detailed explanation of some of the ways to provide documentation for a tool.

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. ➔

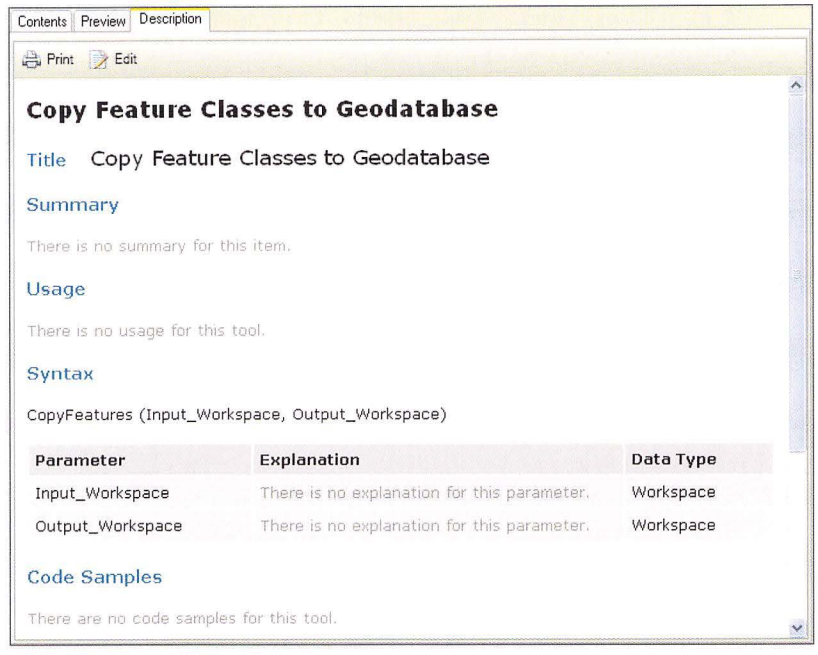
By default, this description will appear in the Help panel on the tool dialog box.



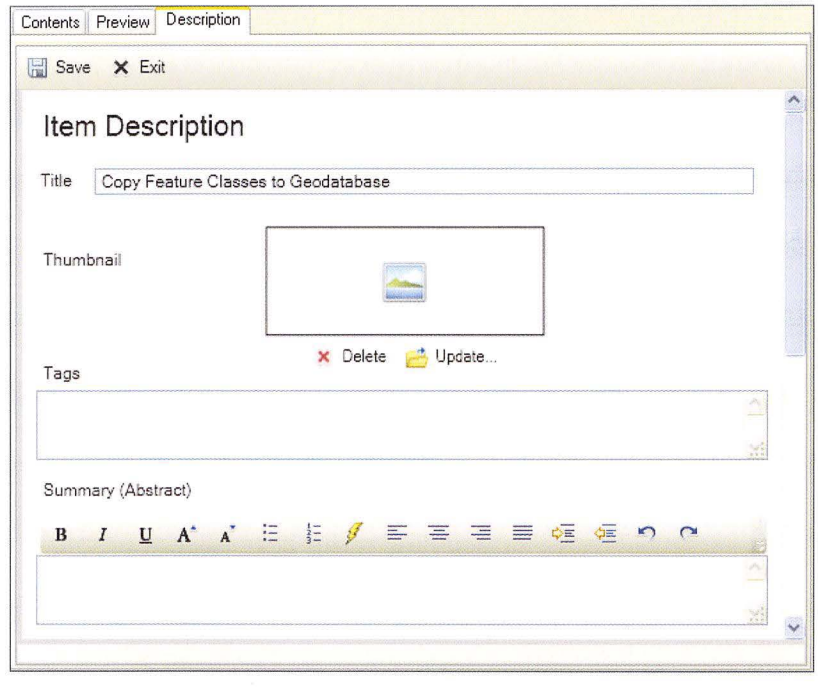
This type of tool documentation is a bit limited because it allows for text only, and it provides a description of the tool only, and not individual parameters. However, it is a good place to start.

Editing the Description page

By default, when a script tool is created, a Description page is also created, which is populated with the tool'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. ➔



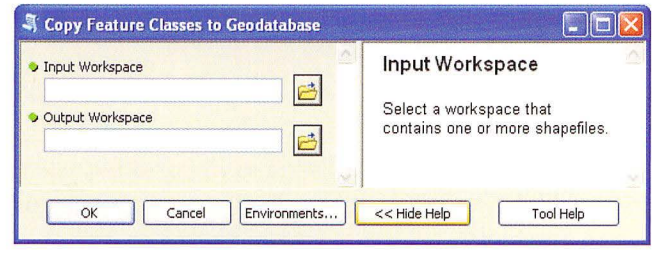
The default description is rather rudimentary and provides little in addition to what the tool dialog box does. However, the contents of the Description page can be modified in ArcCatalog by clicking the Edit button under the Description tab, in the same way that metadata can be edited. ➔



From here, you can edit the title and summary, provide examples of the usage, enter details of the syntax in a table, and add code samples. You can also load a thumbnail 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 relevant 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 individual parameters, it will appear on the tool dialog box Help panel 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.

Note: Creating your own style sheet is not covered in this book.

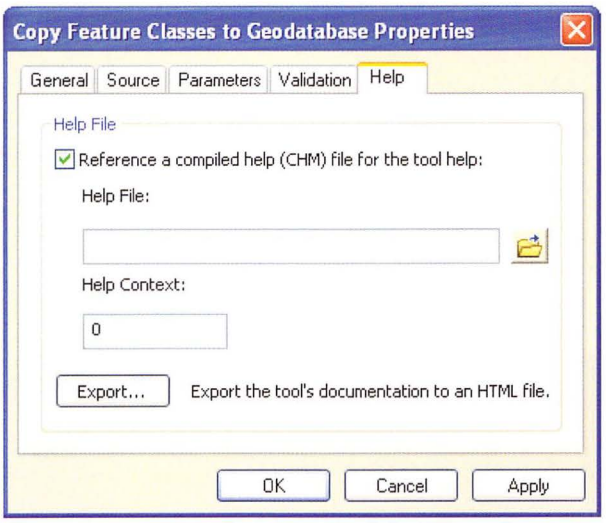
Using a compiled Help file

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).

Note: Creating a compiled Help file is not covered in this book.

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.

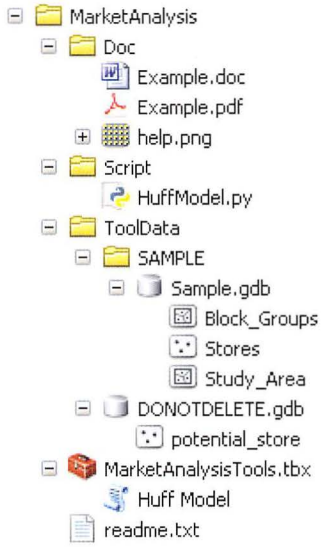


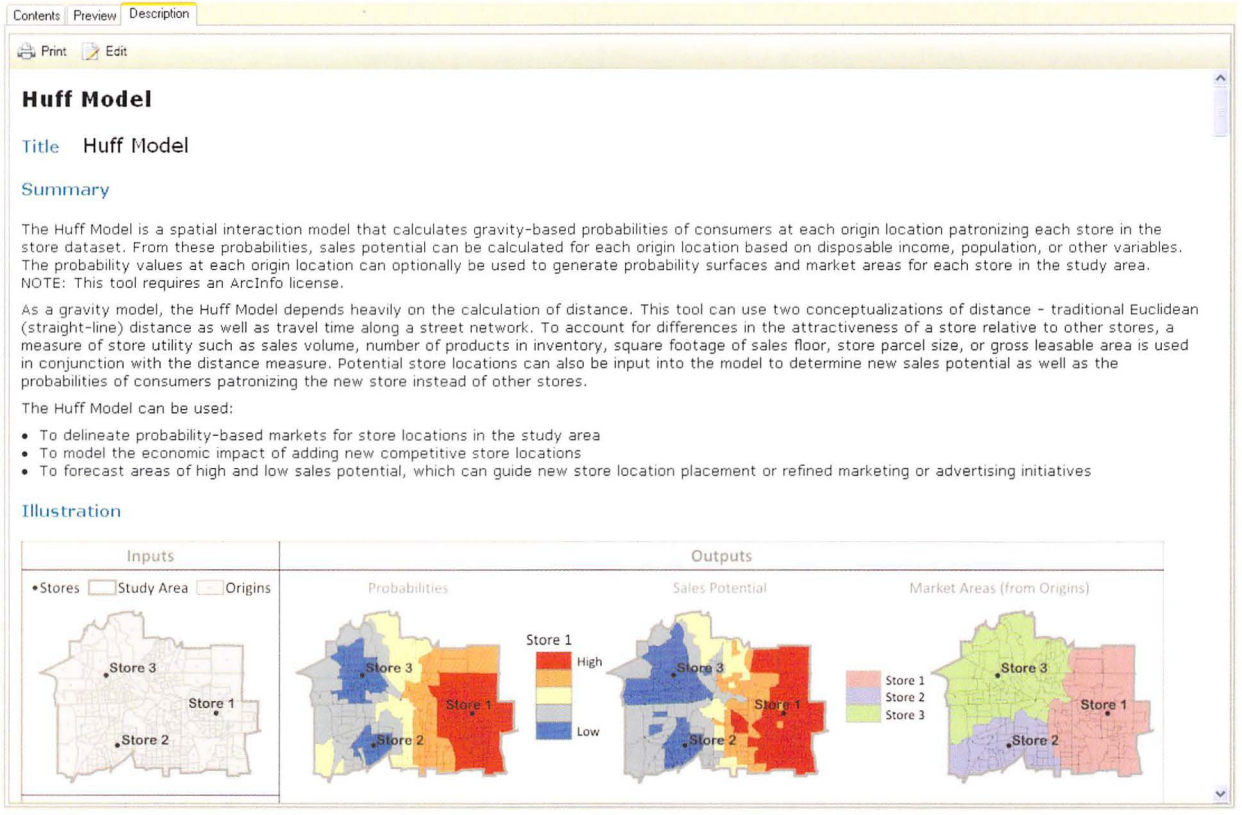
14.10 Example tool: Market analysis

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.

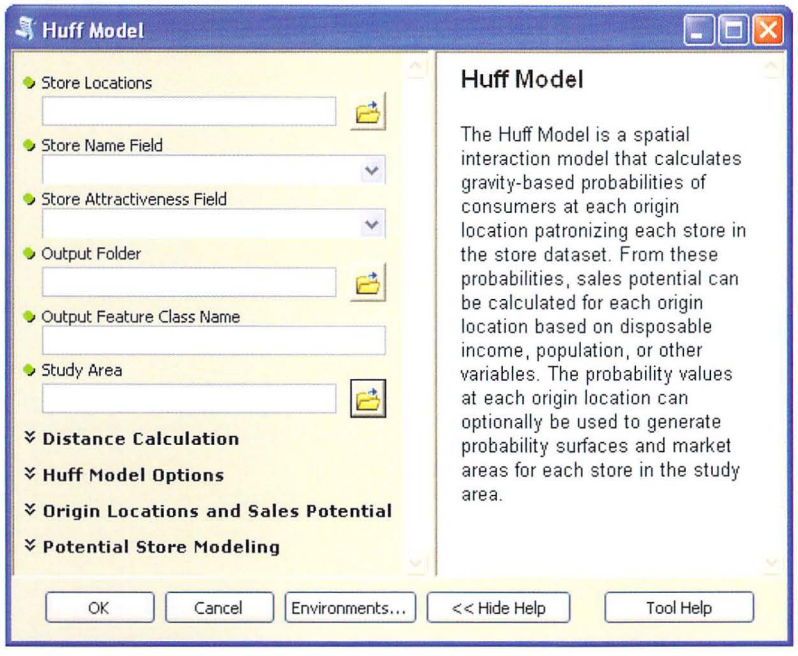
The tool's organization closely follows the suggested folder structure. There is a single .tbx file with a single script tool. The actual script file, HuffModel.py, is located in the Script folder. The ToolData folder contains a sample dataset as well as some data that the tool needs for execution. The 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 stored 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.

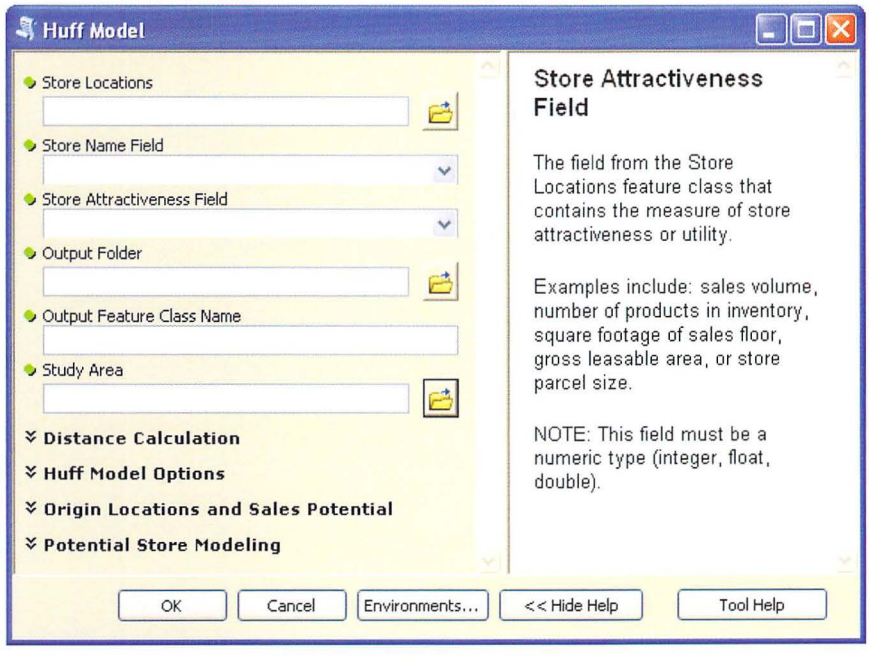




The tool dialog box itself contains a large number of required and optional parameters. When the tool dialog box is open, a brief description of the tool appears in the Help panel.



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.

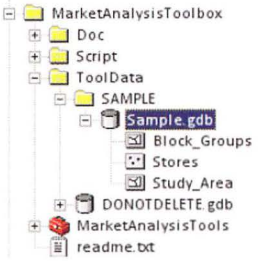


Separate from ArcGIS, tool documentation is provided in the form of a tutorial file on disk that describes the sample data and explains how to run the tool.

Market Analysis with the Huff Model tool

Sample Data

Sample data stored at ...MarketAnalysisToolbox\ToolData\SAMPLE\Sample.gdb



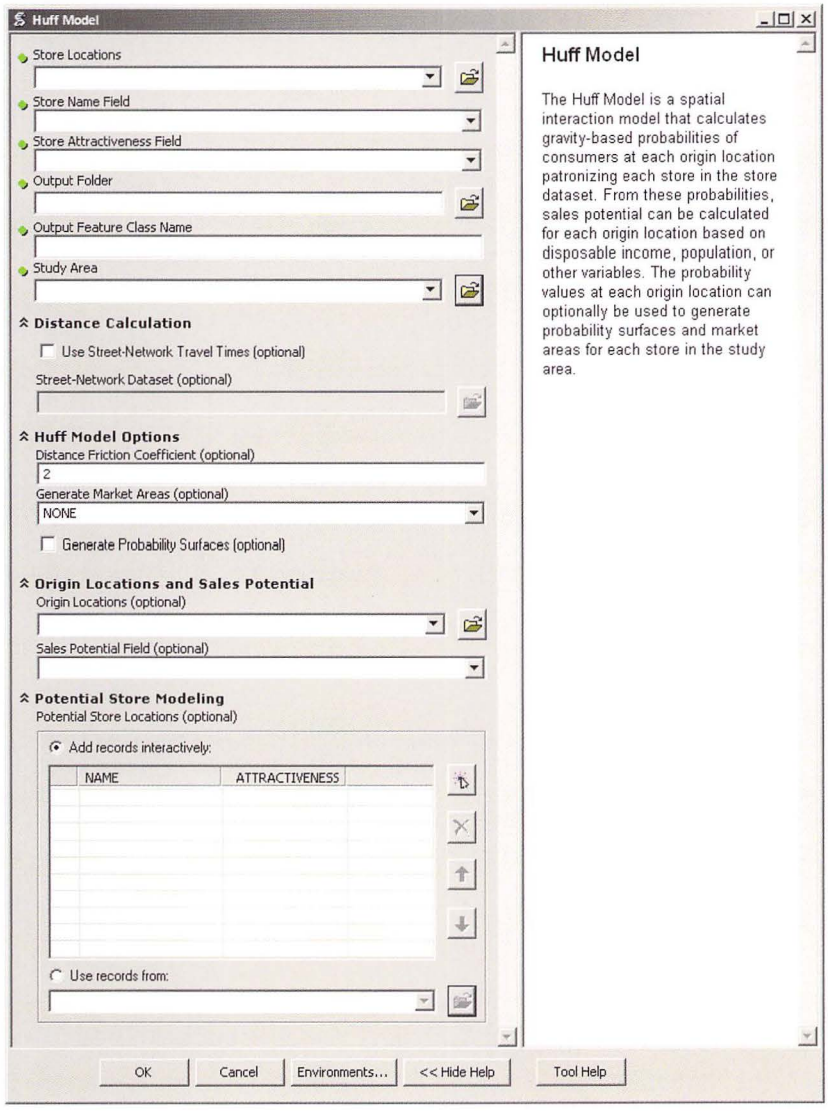
1. Feature Class 'Stores' contains three point features representing retail stores arbitrarily located in the study area for demonstration purposes – they do not represent real store locations. 'Stores' will be used in the "Store Locations" parameter of the Huff Model tool. 'Stores' contains fields "Name" and "Sales" which will be used in the "Store Name Field" and "Store Attractiveness Field" parameters of the Huff Model tool, respectively.

2. Feature Class 'Study_Area' contains a single polygon feature roughly centered on the urban area of Akron, Ohio, United States. 'Study_Area' will be used in the "Study Area" parameter of the Huff Model tool.

3. Feature Class 'Block_Groups' contains 189 polygon features which are U.S. Census Bureau block groups covering the same urban area of Akron, Ohio, United States. 'Block_Groups' can optionally be used in the "Origin Locations" parameter of the Huff Model tool (under the Origin Locations and Sales Potential category). 'Block_Groups' has a number of demographic indicator fields, one of which can optionally be used in the "Sales Potential Field" parameter of the Huff Model tool (under the Origin Locations and Sales Potential category). Suggested fields are "POP2007" or "HOUSEHOLDS".

Market Analysis Tutorial

1. Add the above feature classes to a new ArcMap document. Add MarketAnalysisTools.tbx to the ArcToolbox window. Open the Huff Model tool from the Market Analysis Tools toolbox. Opening each of the drop-down categories, the tool dialog should appear as below.



Finally, the script itself contains documentation in the form of comments.

HuffModel.py

```
# ---------------------------------------------------------------------------
# HuffModel.py
# Created: 4/13/2007 by Drew Flater
# Usage: Creating probability-based trade areas for retail stores
# ---------------------------------------------------------------------------

# Import system modules
import sys, string, arcgisscripting, os, traceback, shutil, re

# Create the Geoprocessor object
gp = arcgisscripting.create(93)

# Set overwrite
gp.overwriteoutput = 1

def AddPrintMessage(msg, severity):
    print msg
    if severity == 0: gp.AddMessage(msg)
    elif severity == 1: gp.AddWarning(msg)
    elif severity == 2: gp.AddError(msg)

# Start traceback Try-Except statement:
try:
    # Script parameters...
    stores = gp.getparameterastext(0)
    store_name = gp.getparameterastext(1)
    store_attr = gp.getparameterastext(2)
    outfolder = gp.getparameterastext(3)
    fc_name = gp.getparameterastext(4)
    studyarea = gp.getparameterastext(5)
    blockgroups = gp.getparameterastext(6)
```

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 files are not moved and folders are not renamed. To share tools, relative paths should be enabled for each script tool. 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 file with a .gpk extension that contains all the files necessary to run a particular geoprocessing workflow, including custom tools, input datasets, and other supporting files.

- Shared tools can be documented in various ways, including editing the Description page in ArcCatalog, using style sheets, and referencing compiled Help files.

Appendix A
Data source credits

Exercise 1
None

Exercise 2
\EsriPress\Python\Data\Exercise02\basin.shp, courtesy of Clay County, Minnesota
\EsriPress\Python\Data\Exercise02\floodzones.shp, courtesy of Clay County, Minnesota
\EsriPress\Python\Data\Exercise02\lakes.shp, courtesy of Clay County, Minnesota
\EsriPress\Python\Data\Exercise02\rivers.shp, courtesy of Clay County, Minnesota
\EsriPress\Python\Data\Exercise02\roads.shp, courtesy of Clay County, Minnesota
\EsriPress\Python\Data\Exercise02\soils.shp, courtesy of Clay County, Minnesota

Exercise 3
\EsriPress\Python\Data\Exercise03\zipcodes.shp, courtesy of City of Austin, Texas

Exercise 4
None

Exercise 5
\EsriPress\Python\Data\Exercise05\bike_routes.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise05\facilities.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise05\hospitals.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise05\parks.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise05\zip.shp, courtesy of City of Austin, Texas

Exercise 6

\EsriPress\Python\Data\Exercise06\amtrak_stations.shp,
courtesy of The National Atlas of the United States of America
\EsriPress\Python\Data\Exercise06\cities.shp, courtesy of The National Atlas of the United States of America
\EsriPress\Python\Data\Exercise06\counties.shp, courtesy of The National Atlas of the United States of America
\EsriPress\Python\Data\Exercise06\new_mexico.shp,
courtesy of The National Atlas of the United States of America
\EsriPress\Python\Data\Exercise06\railroads.shp, courtesy of The National Atlas of the United States of America

Exercise 7

\EsriPress\Python\Data\Exercise07\airports.shp, courtesy of The National Atlas of the United States of America
\EsriPress\Python\Data\Exercise07\alaska.shp, courtesy of The National Atlas of the United States of America
\EsriPress\Python\Data\Exercise07\roads.shp, courtesy of The National Atlas of the United States of America

Exercise 8

\EsriPress\Python\Data\Exercise08\coordinates.txt, created by the author
\EsriPress\Python\Data\Exercise08\dams.shp, courtesy of The National Atlas of the United States of America
\EsriPress\Python\Data\Exercise08\hawaii.shp, courtesy of The National Atlas of the United States of America
\EsriPress\Python\Data\Exercise08\rivers.shp, courtesy of The National Atlas of the United States of America

Exercise 9

\EsriPress\Python\Data\Exercise09\elevation, courtesy of US Geological Survey
\EsriPress\Python\Data\Exercise09\landcover.tif, courtesy of US Geological Survey
\EsriPress\Python\Data\Exercise09\tm.img, courtesy of US Geological Survey

Exercise 10

\EsriPress\Python\Data\Exercise10\counties.shp, from Esri Data & Maps 2009, from Esri, derived from Tele Atlas, US Census, Esri (Pop2008 field)
\EsriPress\Python\Data\Exercise10\Austin\addresses.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise10\Austin\base.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise10\Austin\buildings.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise10\Austin\facilities.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise10\Austin\historical_landmarks.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise10\Austin\hospitals.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise10\Austin\parks.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise10\Austin\sidewalks.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise10\Austin\trees.shp, courtesy of City of Austin, Texas

Exercise 11

\EsriPress\Python\Data\Exercise11\bike_routes.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise11\county.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise11\facilities.shp, courtesy of City of Austin, Texas
\EsriPress\Python\Data\Exercise11\parks.shp, courtesy of City of Austin, Texas

Exercise 12

\EsriPress\Python\Data\Exercise12\parcels.shp, created by the author
\EsriPress\Python\Data\Exercise12\streets.shp, courtesy of Clay County, Minnesota

Exercise 13

\EsriPress\Python\Data\Exercise13\points.shp, courtesy of City of Austin, Texas

Exercise 14

None

Appendix B
Data license agreement

Important: *Read carefully before opening the sealed media package.*

Environmental Systems Research Institute, Inc. (Esri), is willing to license the enclosed data and related materials to you only upon the condition that you accept all of the terms and conditions contained in this license agreement. Please read the terms and conditions carefully before opening the sealed media package. By opening the sealed media package, you are indicating your acceptance of the Esri License Agreement. If you do not agree to the terms and conditions as stated, then Esri is unwilling to license the data and related materials to you. In such event, you should return the media package with the seal unbroken and all other components to Esri.

Esri License Agreement

This is a license agreement, and not an agreement for sale, between you (Licensee) and Environmental Systems Research Institute, Inc. (Esri). This Esri License Agreement (Agreement) gives Licensee certain limited rights to use the data and related materials (Data and Related Materials). All rights not specifically granted in this Agreement are reserved to Esri and its Licensors.

Reservation of Ownership and Grant of License

Esri and its Licensors retain exclusive rights, title, and ownership to the copy of the Data and Related Materials licensed under this Agreement and, hereby, grant to Licensee a personal, nonexclusive, nontransferable, royalty-free, worldwide license to use the Data and Related Materials based on the terms and conditions of this Agreement. Licensee agrees to use reasonable effort to protect the Data and Related Materials from unauthorized use, reproduction, distribution, or publication.

Proprietary Rights and Copyright

Licensee acknowledges that the Data and Related Materials are proprietary and confidential property of Esri and its Licensors and are protected by United States copyright laws and applicable international copyright treaties and/or conventions.

Permitted Uses

Licensee may install the Data and Related Materials onto permanent storage device(s) for Licensee's own internal use.

Licensee may make only one (1) copy of the original Data and Related Materials for archival purposes during the term of this Agreement unless the right to make additional copies is granted to Licensee in writing by Esri.

Licensee may internally use the Data and Related Materials provided by Esri for the stated purpose of GIS training and education.

Uses Not Permitted

Licensee shall not sell, rent, lease, sublicense, lend, assign, time-share, or transfer, in whole or in part, or provide unlicensed Third Parties access to the Data and Related Materials or portions of the Data and Related Materials, any updates, or Licensee's rights under this Agreement.

Licensee shall not remove or obscure any copyright or trademark notices of Esri or its Licensors.

Term and Termination

The license granted to Licensee by this Agreement shall commence upon the acceptance of this Agreement and shall continue until such time that Licensee elects in writing to discontinue use of the Data or Related Materials and terminates this Agreement. The Agreement shall automatically terminate without notice if Licensee fails to comply with any provision of this Agreement. Licensee shall then return to Esri the Data and Related Materials. The parties hereby agree that all provisions that operate to protect the rights of Esri and its Licensors shall remain in force should breach occur.

Disclaimer of Warranty

The Data and Related Materials contained herein are provided "as-is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or noninfringement. Esri does not warrant that the Data and Related Materials will meet Licensee's needs or expectations, that the use of the Data and Related Materials will be uninterrupted, or that all nonconformities, defects, or errors can or will be corrected. Esri is not inviting reliance on the Data or Related Materials for commercial planning or analysis purposes, and Licensee should always check actual data.

Data Disclaimer

The Data used herein has been derived from actual spatial or tabular information. In some cases, Esri has manipulated and applied certain assumptions, analyses, and opinions to the Data solely for educational training purposes. Assumptions, analyses, opinions applied, and actual outcomes may vary. Again, Esri is not inviting reliance on this Data, and the Licensee should always verify actual Data and exercise their own professional judgment when interpreting any outcomes.

Limitation of Liability

Esri shall not be liable for direct, indirect, special, incidental, or consequential damages related to Licensee's use of the Data and Related Materials, even if Esri is advised of the possibility of such damage.

No Implied Waivers

No failure or delay by Esri or its Licensors in enforcing any right or remedy under this Agreement shall be construed as a waiver of any future or other exercise of such right or remedy by Esri or its Licensors.

Order for Precedence

Any conflict between the terms of this Agreement and any FAR, DFAR, purchase order, or other terms shall be resolved in favor of the terms expressed in this Agreement, subject to the government's minimum rights unless agreed otherwise.

Export Regulation

Licensee acknowledges that this Agreement and the performance thereof are subject to compliance with any and all applicable United States laws, regulations, or orders relating to the export of data thereto. Licensee agrees to comply with all laws, regulations, and orders of the United States in regard to any export of such technical data.

Severability

If any provision(s) of this Agreement shall be held to be invalid, illegal, or unenforceable by a court or other tribunal of competent jurisdiction, the validity, legality, and enforceability of the remaining provisions shall not in any way be affected or impaired thereby.

Governing Law

This Agreement, entered into in the County of San Bernardino, shall be construed and enforced in accordance with and be governed by the laws of the United States of America and the State of California without reference to conflict of laws principles. The parties hereby consent to the personal jurisdiction of the courts of this county and waive their rights to change venue.

Entire Agreement

The parties agree that this Agreement constitutes the sole and entire agreement of the parties as to the matter set forth herein and supersedes any previous agreements, understandings, and arrangements between the parties relating hereto.

Appendix C
Installing the data and software

Python Scripting for ArcGIS includes a DVD containing data and exercises. A free, fully functioning 180-day trial version of ArcGIS 10.1 for Desktop Advanced software can be downloaded at esri.com/pythonscriptingforArcGIS10-1. You will find an authorization number printed on the inside back cover of this book. You will use this number when you are ready to install the software.

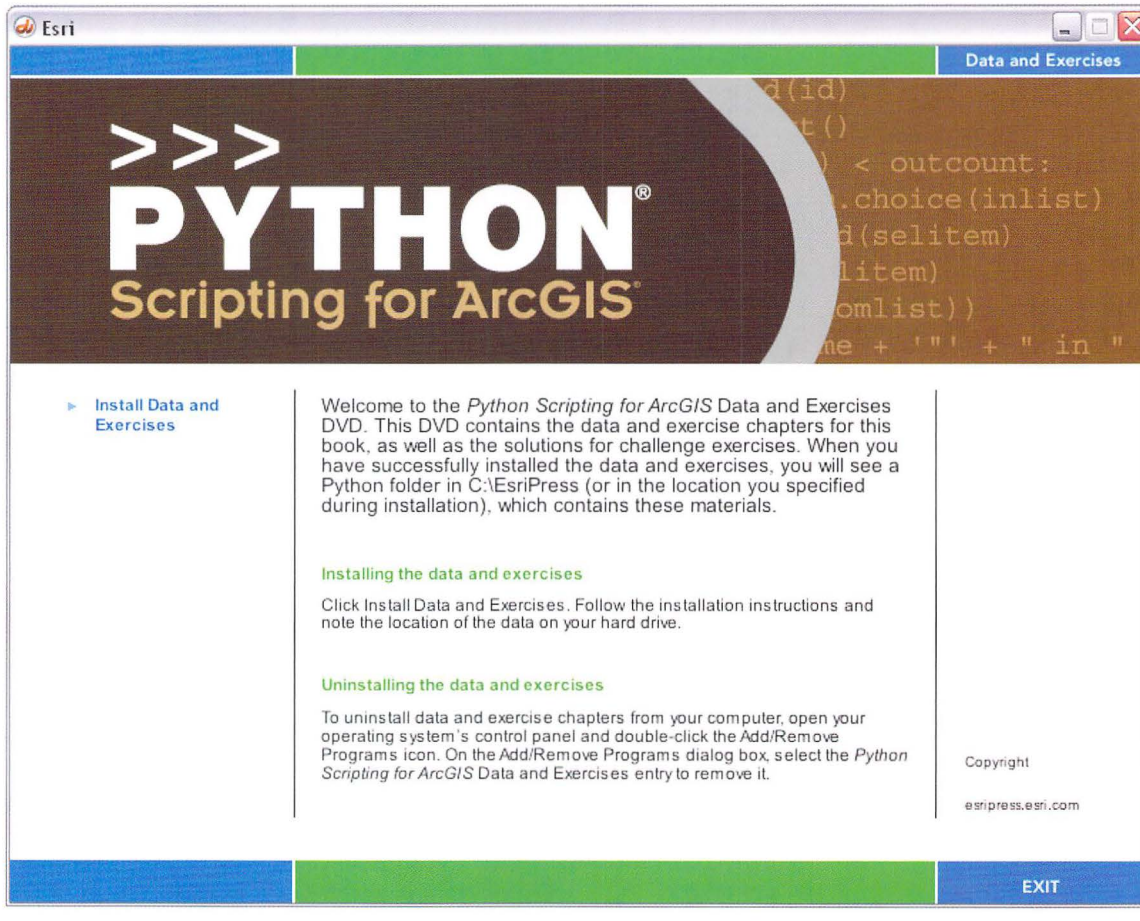
If you already have a licensed copy of ArcGIS 10.1 for Desktop Advanced software installed on your computer (or have access to the software through a network), do not install the trial software. Use your licensed software to do the exercises in this book. If you have an older version of ArcGIS software installed on your computer, you must uninstall it before you can install the software that is provided with this book.

.NET Framework 3.5 SP1 must be installed on your computer before you install ArcGIS 10.1 for Desktop software. Some features of ArcGIS 10.1 for Desktop software require Windows Internet Explorer version 8.0. If you do not have Internet Explorer version 8.0, you must install it before installing the ArcGIS 10.1 for Desktop software.

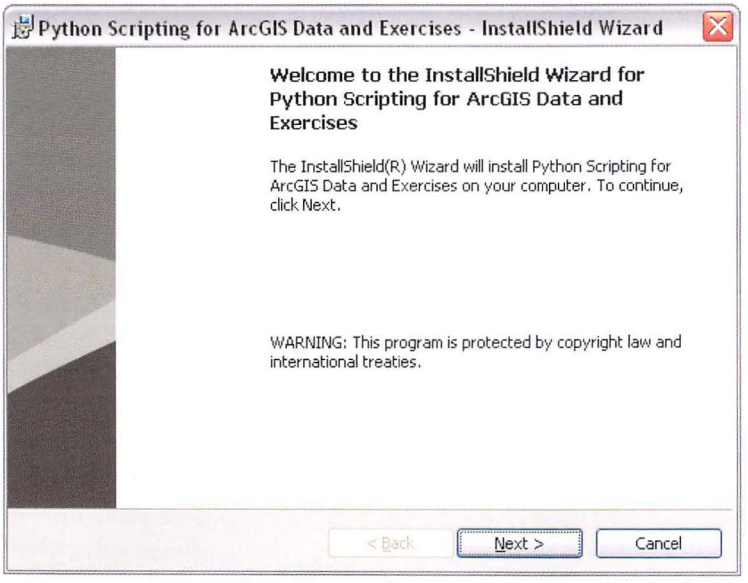
Installing the exercise data

Follow the steps below to install the exercise data.

1. Put the data DVD into your computer's DVD drive. A splash screen will appear.

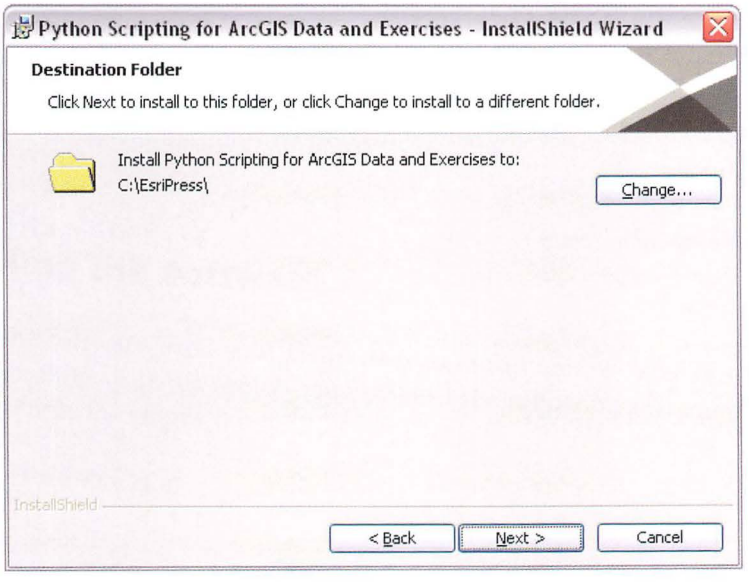


2. Read the welcome, and then click the Install Resources link. This launches the InstallShield Wizard.



3. Click Next. Read and accept the license agreement terms, and then click Next.

4. Accept the default installation folder or click Browse and navigate to the drive or folder location where you want to install the data.



Index