Part 2

Writing scripts

Chapter 5
Geoprocessing using Python

5.1 Introduction

This chapter describes the ArcPy site package, which allows for a close integration of ArcGIS and Python. ArcPy modules, classes, and functions, which give access to all the geoprocessing tools in ArcGIS, are introduced. Several additional nontool functions related to running geoprocessing tasks are also covered, including establishing environment settings, setting paths to data, and managing licenses.

5.2 Using the ArcPy site package

The geoprocessing functionality of ArcGIS can be accessed through Python using the ArcPy site package. A site package in Python is like a library of functions that add functionality to Python. The site package works very much like a module, but a package contains multiple modules as well as functions and classes.

ArcPy was introduced in version 10 of ArcGIS with the goal of making Python scripting easier and more powerful. Prior to ArcGIS 10, the geoprocessing functionality of ArcGIS was accessed through Python using the `ArcGISscripting` module. Scripts written for earlier versions of ArcGIS typically use this module. The focus in this book is on ArcPy, and the older module is not covered in detail. However, because you may sometimes be working with older scripts, the `ArcGISscripting` module is briefly explained in section 5.4. This module is still supported in ArcGIS 10, so older scripts using this module continue to work.

ArcPy is organized in modules, functions, tools, and classes, which are described later in this chapter.

5.3 Importing ArcPy

Working with ArcPy starts with importing the site package. A typical geoprocessing script therefore starts with the following line of code:

```
import arcpy
```

Once you import ArcPy, you can run all the geoprocessing tools found in the standard toolboxes installed with ArcGIS.

ArcPy contains many modules, including two specialized ones: a map automation module (`arcpy.mapping`) and a map algebra module (`arcpy.sa`). To import these modules, you can use the following syntax:

```
import arcpy.mapping
```

Once you import ArcPy or one of its specialized modules, you can start using its modules, functions, and classes.

One of the first tasks typically is to set the current workspace. For example, here is how you would set the current workspace to C:\Data:

```
import arcpy
arcpy.env.workspace = "C:/Data"
```

Note: *Remember that you should not use a backslash (\) for paths because Python views it as an escape character.*

Notice that the path is a string variable.

Environment settings are exposed as properties of the ArcPy `env` class. Classes are explained in more detail in section 5.8. These properties can be used to retrieve the current values or to set them. In the preceding code, `env` is a class and `workspace` is a property of this class. When using classes, the syntax is as follows:

```
arcpy.<class>.<property>
```

You will see more examples of this syntax later in this chapter.

Often you may not need to use the entire module. You can use the `from-import` statement to import only a portion of a module. The following code imports just the `env` class. Instead of accessing environments using `arcpy.env`, you can simplify it to `env`.

```
from arcpy import env
env.workspace = "C:/Data"
```

You can further control the importing of modules by giving a module or part of a module a custom name using the `from-import-as` statement as follows:

```
from arcpy import env as myenv
myenv.workspace = "C:/Data"
```

Although using custom names does not shorten the length of your code, it can make it easier to read.

The `from-import-as-*` statement goes a step further. In this case, the contents of the module are imported directly into the namespace, meaning that you no longer need to add a prefix to the contents, such as "myenv," or use the module name.

```
from arcpy import env as *
workspace = "C:/Data"
```

Note: *The use of the* `from-import-as-*` *statement can reduce the length of your code, but you have to be very careful because other objects, variables, and modules that have the same name will be overwritten. In general, it is recommended that you stick with* `from arcpy import env`.

5.4 Working with earlier versions of ArcGIS

The ArcPy site package was introduced with ArcGIS 10. Prior to the introduction of ArcPy, geoprocessing tools were accessed from Python using the `ArcGISscripting` module. Thus, Python scripts created prior to ArcPy use a syntax that is slightly different from the `import arcpy` statement to import geoprocessing functionality.

Using the 9.3 version of the geoprocessor object, the syntax is as follows:

```
import ArcGISscripting
gp = ArcGISscripting.create(9.3)
```

Using the pre-9.3 version of the geoprocessor object, the syntax is as follows:

```
import ArcGISscripting
gp = ArcGISscripting.create()
```

Notice that, in both cases, first the `ArcGISscripting` module is imported, and then a geoprocessor object is created. All geoprocessing capabilities of ArcGIS are exposed as methods of the geoprocessor object. In ArcPy, it is no longer necessary to create a geoprocessor object.

Prior to the introduction of the `ArcGISscripting` module, Python scripts could access the geoprocessing tools using the Python `win32com`

module to create the geoprocessor object. Here is what the code looks like using the `win32com` module:

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

Once the geoprocessor object is created using one of these three methods, the rest of the syntax is relatively similar, although not identical, to the use of ArcPy. For example, the code to set the current workspace is

```
gp.workspace = "C:/Data"
```

The Python `win32com` module is no longer installed with ArcGIS 10. As a result, any scripts that use the `win32com` module do not work under the default installation. However, the installation of the PythonWin editor includes an installation of the `win32com` module, which allows scripts using the module to work.

Despite the similarity, it can be a bit confusing to work with these earlier versions of ArcGIS and ArcPy at the same time. In this book, the focus is on the use of ArcPy because it has many advantages over the earlier versions. Many existing Python scripts (including many written by Esri) were written using the `ArcGISscripting` module. These scripts continue to work because ArcGIS 10 continues to support them. However, much of the additional functionality introduced in ArcGIS 10 is not available in the earlier versions of ArcGIS.

If you are just getting started with Python scripting, you will most likely need to learn how to work with the ArcPy site package only. If you need to edit older scripts, you may need to learn a bit more about the `ArcGISscripting` and `win32com` modules. Please note, however, that ArcGIS Desktop Help for version 10 no longer includes a description of these modules or any sample code using these modules. As a result, you would need to fall back on the Help for version 9.3. These Help files remain available online at http://webhelp.esri.com/ArcGISdesktop/9.3.

Note: *The book includes no further details on the* `ArcGISscripting` *or* `win32com` *modules.*

5.5 Using tools

ArcPy gives you access to ArcGIS for Desktop geoprocessing tools. When working with geoprocessing tools in Python, the tools are referred to by name. This does not correspond exactly to the tool label, which is how the tool appears in ArcToolbox. A tool name is generally very similar to the tool label but contains no spaces. For example, the name of the Add Field tool in the Data Management toolbox is AddField.

In addition to using the tool name rather than the tool label, a reference to a particular tool also requires the toolbox alias. It is because multiple tools in different toolboxes can share the same name. For example, there

are several Clip tools: one in the Analysis toolbox and one in the Data Management toolbox. The toolbox alias is not the same as either the name or the label of the toolbox—it is typically an abbreviated version. For example, the alias of the Data Management toolbox is "management."

The Clip tool in the Data Management toolbox is therefore referenced as Clip_management. Notice that the name of the toolset (Raster > Raster Processing) is not referenced in any way.

There are two ways to access a tool in a line of Python code. The easiest way to call a tool is to call its corresponding function. All tools are available as functions in ArcPy. An ArcPy function is a defined bit of functionality that does a specific task. The syntax for calling a tool by its function is as follows:

```
arcpy.<toolname_toolboxalias>(<parameters>)
```

For example, the following code runs the Clip tool:

```
import arcpy
arcpy.env.workspace = "C:/Data"
arcpy.Clip_analysis("streams.shp", "study.shp", "result.shp")
```

Tools are also available in modules that match the toolbox alias name. An alternative way to access a tool is to first call the toolbox as a module and then the tool as a function, followed by the tool's parameters. The syntax is as follows:

```
arcpy.<toolboxalias>.<toolname>(<parameters>)
```

Here is what the example looks like for running the Clip tool:

```
import arcpy
arcpy.env.workspace = "C:/Data"
arcpy.analysis.Clip("streams.shp", "study.shp", "result.shp")
```

Both methods are correct, and the approach you use is a matter of preference and coding habits.

Note: The coding style adopted for this book uses the `arcpy.Clip_analysis` *style of calling tools, but you may see the other style in example scripts from other sources.*

Just a few quick reminders regarding Python syntax:

- Python is case sensitive, so `Clip` is correct, but `clip` is not.
- The use of spaces, or whitespace, in a line of code has no effect on its execution. For example, `workspace="C:/Data"` is the same as `workspace = "C:/Data"`. Whitespace is recommended to improve readability but is not required. However, do not include spaces between modules, functions, classes, methods, and properties, so `env.workspace` is correct, but `env. workspace` is not. Also do not include spaces between functions and their arguments, so

`<toolname>(<parameters>)` is correct, but `<toolname> (<parameters>)` is not.

- Quotation marks in Python are straight and not curly, so use " " and not “ ”. When typing code using a Python editor, the correct style of quotation marks is entered automatically. However, incorrect quotation marks can occur when copying code from other applications, such as a Microsoft Word document or a PDF file.

A key aspect of running geoprocessing tools is to get the syntax right for the parameters. Every geoprocessing tool has parameters, required and optional, that provide the tool with the information it needs for execution. Common parameters are input datasets, output datasets, and keywords that control the execution of the tool. Parameters themselves have properties such as the following:

- Name: a unique name for each tool parameter
- Type: the type of data expected, such as feature class, integer, string, or raster
- Direction: whether the parameter defines input or output values
- Required: whether a value must be provided for a parameter or is optional

The documentation of each tool describes its parameters and properties. Once a valid set of parameters is provided, the tool is ready to be run. Most parameters are specified as a simple string. Strings consist of text that identifies a parameter value, such as a path to a dataset or a keyword.

The Clip tool's documentation, as shown in the figure, describes its parameters.

Parameter	Explanation	Data Type
in_features	The features to be clipped.	Feature Layer
clip_features	The features used to clip the input features.	Feature Layer
out_feature_class	The feature class to be created.	Feature Class
cluster_tolerance (Optional)	The minimum distance separating all feature coordinates as well as the distance a coordinate can move in X or Y (or both). Set the value to be higher for data with less coordinate accuracy and lower for data with extremely high accuracy.	Linear unit

The Clip tool has four parameters, with the last one (cluster_tolerance) being optional. The syntax of the Clip tool is

```
Clip_analysis(in_features, clip_features, out_feature_class, ➔
➔ {cluster_tolerance})
```

The name of the Clip tool is followed by the tool's parameters in parentheses. Parameters are separated by a comma (,). Optional parameters are surrounded by curly brackets ({ }).

The syntax of geoprocessing tools typically follows the same general pattern, as follows:

- Required parameters come first, followed by optional parameters.
- The input datasets are usually the first parameter or parameters, followed by the output dataset if there is one. Next are additional required parameters, and finally, optional parameters.
- Parameter names for input datasets are prefixed by "in_" (such as, in_data, in_features, in_table, in_workspace) and parameter names for output datasets are prefixed by "out_" (such as, out_data, out_features, out_table).

Listing required parameters first makes it easy to simply leave out the optional parameters when they are not needed. Sometimes, however, some of the optional parameters need to be set. Because parameters need to be specified in the order that they are listed in the tool syntax, it can mean that some optional parameters may need to be skipped.

Consider, for example, the syntax of the Buffer tool:

```
Buffer_analysis (in_features, out_feature_class, buffer_distance_or_
field, {line_side}, {line_end_type}, {dissolve_option}, {dissolve_field})
```

A code example of the Buffer tool is as follows:

```
import arcpy
arcpy.env.workspace = "C:/Data/study.gdb"
arcpy.Buffer_analysis("roads", "buffer", "100 METERS")
```

Using this example, how would you specify the optional dissolve_option parameter and skip the other optional parameters that follow the required parameters? It can be accomplished in different ways, as follows:

- By setting the optional parameters using an empty string ("") or the number sign ("#")
- By specifying by name the parameter that needs to be set, bypassing all others

The Buffer tool has three required parameters and four optional parameters. To specify a dissolve option and the field to use in this dissolve, two optional parameters need to be skipped. This can be done in three ways:

```
arcpy.Buffer_analysis("roads", "buffer", "100 METERS", "", "", "LIST",
"Code")

arcpy.Buffer_analysis("roads", "buffer", "100 METERS", "#", "#", "LIST",
"Code")

arcpy.Buffer_analysis("roads", "buffer", "100 METERS", dissolve_
option="LIST", dissolve_field="Code")
```

In each of these three cases, the optional parameters line_side and line_end_type are left as default values.

Note: Although each of these three options is correct, the examples in this book typically use the empty string ("").

In the examples so far, the parameters of the tool use the actual file name (for example, "roads"). This means the file names are *hard-coded*. That is, the parameters are not set as variables, but use the values directly. Although this syntax is correct and works fine, it is often more useful to make your code flexible by using variables for parameters instead of using file names. First, create a variable and assign it a value. Then you can use the variable for a parameter. These variable values are passed to the tool. For example, in the case of the Clip tool, it would look as follows:

```
import arcpy
arcpy.env.workspace = "C:/Data"
infc = "streams.shp"
clipfc = "study.shp"
outfc = "result.shp"
arcpy.Clip_analysis(infc, clipfc, outfc)
```

In this example script, the names of the datasets are still hard-coded in the script itself but not in the specific line of code that calls the Clip tool. The next logical step is to have the values of the variables provided by a user or another tool, which means the actual file names would no longer appear in the script. For example, the following code runs the Copy tool, and the input and output feature classes are obtained from user input using the `GetParameterAsText` function:

```
import arcpy
infc = arcpy.GetParameterAsText(0)
outfc = arcpy.GetParameterAsText(1)
arcpy.Copy_management(infc, outfc)
```

Functions are covered later in this chapter and the `GetParameterAsText` and related functions are covered in chapter 13. Setting tool parameters based on user input is commonly used for script tools. Working with

variables in this way gives you more flexibility and makes much of your code reusable.

Here are just a few quick reminders regarding variable names in Python. Variable names can consist of any combination of valid characters. However, adopting a consistent coding style is recommended. For example, the *Style Guide for Python Code* recommends using all lowercase characters and using underscores (_) only if it improves readability—for example, `my_clip` or `clip_result`. Variable names should also be kept short (but meaningful) to limit the need for typing and associated typos—for example, `clipfc` instead of `clipfeatureclass`.

ArcPy returns the output of a tool as a result object. When the output of a tool is a new or updated feature class, the result object includes the path to the dataset. For other tools, however, the result object can consist of a string, a number, or a Boolean value. One of the advantages of result objects is that you can keep track of information about the running of tools. This includes not only the output, but also messages and parameters.

For example, in the following code, a geoprocessing tool is run and the output is returned as a result object:

```
import arcpy
arcpy.env.workspace = "C:/Data"
mycount = arcpy.GetCount_management("streams.shp")
print mycount
```

This code displays the string representation of the result object. For example:

```
3153
```

When the output of a tool consists of a feature class, the result object includes the path to the dataset. For example, the following code runs the Clip tool:

```
import arcpy
arcpy.env.workspace = "C:/Data"
myresult = arcpy.analysis.Clip("streams.shp", "study.shp", "result.shp")
print myresult
```

Running the code displays the string representation of the path to the output dataset:

```
C:/Data/result.shp
```

The result object has properties and methods, which are covered in more detail later in this chapter.

The result object can be used as an input to another function. For example, in the following code, a feature class is buffered using the Buffer tool.

The output polygon feature class is returned as an object and the object is used as the input to the Get Count tool, as follows:

```
import arcpy
arcpy.env.workspace = "C:/Data/study.gdb"
buffer = arcpy.Buffer_analysis("str", "str_buf", "100 METERS")
count = arcpy.GetCount_management(buffer)
print count
```

Although many tools have only a single output, some tools have multiple outputs. The `getOutput` method of the result object can be used to obtain a specific output by using an index number. A more generic way to get a geoprocessing result follows:

```
import arcpy
arcpy.env.workspace = "C:/Data/study.gdb"
buffer = arcpy.Buffer_analysis("str", "str_buf", "100 METERS")
count = arcpy.GetCount_management(buffer).getOutput(0)
print str(count)
```

You can probably see where this is going. You can create a series of geoprocessing operations, just as in ModelBuilder, and only the final desired output is returned back to the application that called the script.

5.6 Working with toolboxes

When the ArcPy site package is imported into Python, all the system toolboxes are available. When custom tools are created and stored in a custom toolbox, these tools can be accessed in Python only by importing the custom toolbox. So even if a custom toolbox has been added to ArcToolbox in ArcMap or ArcCatalog, Python is not aware of this toolbox until it has been imported. This is accomplished using the `ImportToolbox` function. The following code illustrates how to import a toolbox:

```
import arcpy
arcpy.ImportToolbox("C:/Data/sampletools.tbx")
```

Notice that the `ImportToolbox` function references the actual file on disk—that is, the toolbox (.tbx) file, not the name of the toolbox.

After importing a toolbox, the syntax for accessing a tool in Python is as follows:

```
arcpy.<toolname>_<toolboxalias>
```

This syntax is exactly the same as the syntax for accessing the tools from the system toolboxes. ArcPy depends on toolbox aliases to access and run the correct tool. System toolboxes have well-defined aliases, but this may not be the case for custom toolboxes. The alias of a toolbox is different from both the name (which is the file name of the .tbx file) and the label (which is the display name of the toolbox) and needs to be specified separately; there is no default alias. It is therefore good practice to always define a custom toolbox alias. However, if a toolbox alias is not defined, a temporary alias can be set as the second parameter of the `ImportToolbox` function, as follows.

```
arcpy.ImportToolbox("sampletools.tbx", mytools)
```

Once the alias is set, tools in the toolbox can be accessed using Python. For example, if the `sampletools.tbx` file contains a tool called MyModel, the syntax to access this tool would look as follows:

```
arcpy.MyModel_mytools(<parameters>)
```

Or alternatively:

```
arcpy.mytools.MyModel(<parameters>)
```

The `ImportToolbox` function can also be used to add geoprocessing services from an Internet or local server.

Although ArcPy provides access to all the geoprocessing tools in ArcGIS, the tools that are available depend on the product (ArcGIS for Desktop Basic, ArcGIS for Desktop Standard, and ArcGIS for Desktop Advanced) and whether any extensions are installed and licensed. In addition, custom toolboxes may be installed, which adds new tools.

Once a particular tool is identified, the tool's syntax can be accessed from Python using the `Usage` function. For example, the following code prints the syntax of all the tools in the Editing Tools toolbox:

```
import arcpy
tools = arcpy.ListTools("*_analysis")
for tool in tools:
print arcpy.Usage(tool)
```

The syntax can also be accessed in the Help file for each tool, but the `Usage` function gives you access to the syntax within Python.

Another way to access the syntax directly is to use Python's built-in `help` function. For example, the following code prints the syntax of the Clip tool:

```
print help(arcpy.Clip_analysis)
```

5.7 Using functions

A function in Python is a specific bit of functionality that does a specific task. All geoprocessing tools are provided as functions. In addition, ArcPy provides a number of functions that are not tools. Functions can be used to list datasets, retrieve properties of a dataset, check for the existence of data, validate names of datasets, and perform many other useful tasks. These functions are designed for Python workflows and therefore are available only from ArcPy and not as tools in ArcToolbox. So in ArcPy, all tools are functions, but not all functions are tools.

The general form of a function is very similar to that of a tool. A function has parameters (also referred to as arguments), which can be required or optional. A function returns values. Returned values for most functions are result objects. It can be the path to a dataset, a string, a number, a Boolean value, or a geoprocessing object.

The syntax of a function is the same as for tools:

```
arcpy.<functionname>(<arguments>)
```

For example, the following code determines whether a particular dataset exists, and then prints either `True` or `False`:

```
import arcpy
print arcpy.Exists("C:/Data.streams.shp")
```

The `arcpy.Exists` function returns a Boolean value. Other functions return other types of values, including strings and numbers.

There are a large number of functions, which can be divided into the following categories:

- Cursors
- Describing data
- Environment and settings
- Fields
- Geodatabase administration
- General
- General data functions
- Getting and setting parameters
- Licensing and installation
- Listing data
- Log history
- Messaging and error handling
- Progress dialog boxes
- Raster
- Spatial reference and transformations
- Tools and toolboxes

These categories are created primarily to provide a logical organization of the functions, but the names of these categories do not appear in Python syntax. So unlike tools, these functions are always called directly, without reference to the categories. ArcGIS Desktop Help provides a complete list of ArcPy functions and a detailed description of each. Several of these functions are revisited in later chapters of the book.

Technically speaking, all geoprocessing tools are functions in ArcPy, and they are accessed like any other Python function. To avoid confusion, ArcPy functions are divided into tool functions and nontool functions. There are a number of important distinctions between the two:

- The documentation is located in different sections of ArcGIS Desktop Help. Tools are documented in the Geoprocessing Tool Reference. It can also be obtained from the tool dialog box when running a tool from ArcToolbox. Nontool functions are documented only in the ArcPy documentation.

- Tools are licensed by product level (ArcGIS for Desktop Basic, ArcGIS for Desktop Standard, and ArcGIS for Desktop Advanced) and by extension (ArcGIS 3D Analyst, ArcGIS Network Analyst, ArcGIS Spatial Analyst, and more). The Geoprocessing Tool Reference indicates the license level that is required for each tool. Nontool functions, on the other hand, are not licensed separately. All the ArcPy nontool functions are installed with ArcPy, independent of the license level.

- Tools produce geoprocessing messages, which can be accessed through a variety of functions. Nontool functions do not produce these messages.

- Calling tools requires the use of the toolbox alias or the module name, whereas nontool functions do not.

- Tools return a result object, whereas nontool functions do not.

5.8 Using classes

Many tool parameters are straightforward—for example, feature classes, field names, and numerical values. They are relatively easy to work with and can often be specified using a simple string value. Some tool parameters, however, are more complex—for example, using a coordinate system as a parameter, or a reclassification table when working with raster data. This is where classes come into play. ArcPy classes are often used as shortcuts for tool parameters that would otherwise have a more complicated equivalent. Classes can be used to create objects, and once the object is

created, its properties and methods can be used. So, the spatial reference or the reclassification table is passed as an object.

Consider the example of a class property. Earlier in this chapter, you worked with the ArcPy `env` class. Environment settings are exposed as properties of the `env` class. For example, workspace is a property of the `env` class, so the syntax becomes `env.workspace`.

The syntax for setting the property of a class is

```
<classname>.<property> = <value>
```

As discussed earlier, the code to set the current workspace is as follows:

```
import arcpy
arcpy.env.workspace = "C:/Data"
```

Another frequently used ArcPy class is the `SpatialReference` class. This class has a number of properties, including the coordinate system parameters. To work with these properties, however, the class first has to be *instantiated*. The syntax for using a method to initialize a new instance of a class is

```
arcpy.<classname>(parameters)
```

The code to initialize a new instance of the `SpatialReference` class is as follows:

```
import arcpy
prjfile = "C:/Data/myprojection.prj"
spatialref = arcpy.SpatialReference(prjfile)
```

In this example, `SpatialReference` is the class that creates the `spatialref` object by reading an existing projection (.prj) file. The actual .prj file already exists on disk and is used to create the object. Once the object is created, you can work with the properties of the object. For example, in the case of the `SpatialReference` class, you can work with any of the specific parameters that define a spatial reference file, such as the coordinate system parameters, tolerances, and domains.

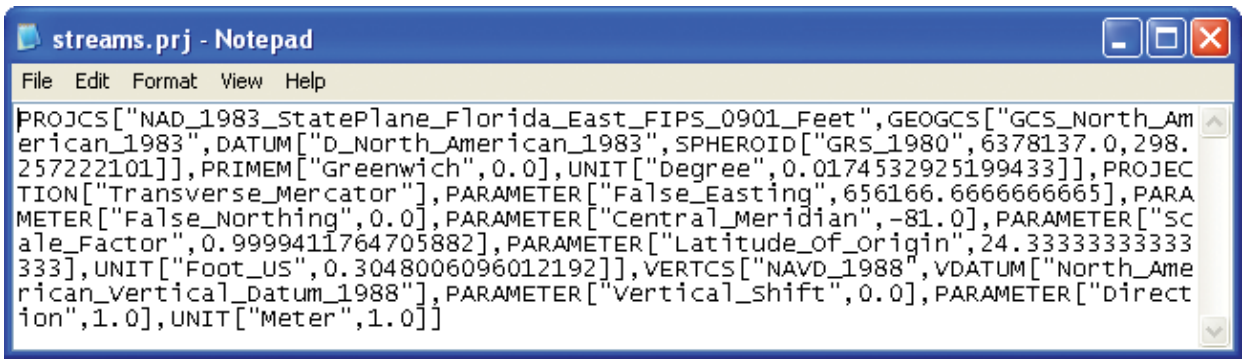
For example, the following code creates a spatial reference object based on an existing .prj file, and then uses the `name` property to get the name of the spatial reference:

```
import arcpy
prjfile = "C:/Data/streams.prj"
spatialref = arcpy.SpatialReference(prjfile)
myref = spatialRef.name
print myref
```

Running the code prints the name of the spatial reference, something like the following:

```
NAD_1983_StatePlane_Florida_East_FIPS_0901_Feet
```

Classes are often used to avoid having to use long and complicated strings. A good example is using classes for complex tool parameters. Most tool parameters are defined using simple strings, including dataset names, paths, keywords, field names, domain names, tolerances, and others. However, some parameters are harder to define by simple strings because the parameters require more properties. For example, think of the coordinate system of a feature class. For a shapefile, it is stored in a .prj file—a regular text file with the file extension .prj. Opening a .prj file in Notepad looks something like the example in the figure.



Working with this type of string can be a bit cumbersome. It would be much easier if you could refer to it simply by using the name of the coordinate system or by referencing the .prj file that contains the string value. Working with the `SpatialReference` class is the way to do it.

For example, a `SpatialReference` object can be created and used to define the output coordinate system of a new feature class. The new feature class is created using the Create Feature Class tool. The syntax of the Create Feature Class tool is as follows:

```
CreateFeatureclass_management(out_path, out_name, {geometry_type},
{template}, {has_m}, {has_z}, {spatial_reference}, {config_keyword},
{spatial_grid_1}, {spatial_grid_2}, {spatial_grid_3})
```

The following code creates a spatial reference object and uses it to define the output coordinate system of a new feature class:

```
import arcpy
out_path = "C:/Data"
out_name = "lines.shp"
prjfile = "C:/Data/streams.prj"
spatialref = arcpy.SpatialReference(prjfile)
arcpy.CreateFeatureclass_management(out_path, out_name, "POLYLINE", "",
"", "", spatialref)
```

Using the `SpatialReference` object is a lot easier than trying to work with the actual string value contained in the .prj file.

> ***Note:*** *As a reminder, optional tool parameters can be left out of the syntax unless they are followed by an optional parameter that is being specified differently from the default. A blank string ("") signifies the use of default values for optional parameters. In the preceding example code, the spatial reference parameter is preceded by three optional parameters—hence the use of three empty strings.*

5.9 Using environment settings

Environment settings are essentially hidden parameters that influence how a tool runs. You have already seen how to set the environments in Python using the `env` class. This section covers these settings in a bit more detail, because environments are fundamental to controlling geoprocessing workflows.

Environment settings are exposed as properties of the `env` class. These properties can be used to retrieve the current values or to set them. Each property has a name and a label. The labels are displayed on the Environment Settings dialog box in ArcGIS, but Python works with names only. The syntax for accessing the properties from the environment class is

```
arcpy.env.<environmentName>
```

For example, to set the current workspace, the following code is used:

```
import arcpy
arcpy.env.workspace = "C:/Data"
```

Alternatively, environments can be accessed using the `from-import` statement:

```
import arcpy
from arcpy import env
env.workspace = "C:/Data"
```

The `env` class also has many other properties. A complete list can be found in the ArcPy documentation. Some important properties include the extent, the output coordinate system, the scratch workspace, and the XY domain. Some properties are specific either to feature classes or to raster datasets. For example, cell size, compression, and mask are used for raster datasets only. The following code sets the cell size to 30:

```
import arcpy
from arcpy import env
env.cellSize = 30
```

The properties of the `env` class not only specify environments, but can also be used to retrieve their current values. For example, the following code retrieves the current settings for the XY tolerance:

```
import arcpy
from arcpy import env
print env.XYTolerance
```

Running the code prints the current value of the XY Tolerance parameter. The default value of `None` is printed unless the value has previously been set.

To get a complete list of properties, you can use the ArcPy `ListEnvironments` function:

```
import arcpy
print arcpy.ListEnvironments()
```

Running the code prints the alphabetical list of all properties.

There is one additional environment setting that is of special interest. You may recall from chapter 2, the geoprocessing options include the option to overwrite the outputs of geoprocessing operations. In ArcMap, this is not part of the Environment Settings dialog box but is a separate option on the menu bar under Geoprocessing > Options. In Python, this is a property of the `env` class. The default value of this `overwriteOutput` property is `False`. The following code sets the value to `True`:

```
import arcpy
from arcpy import env
env.overwriteOutput = True
```

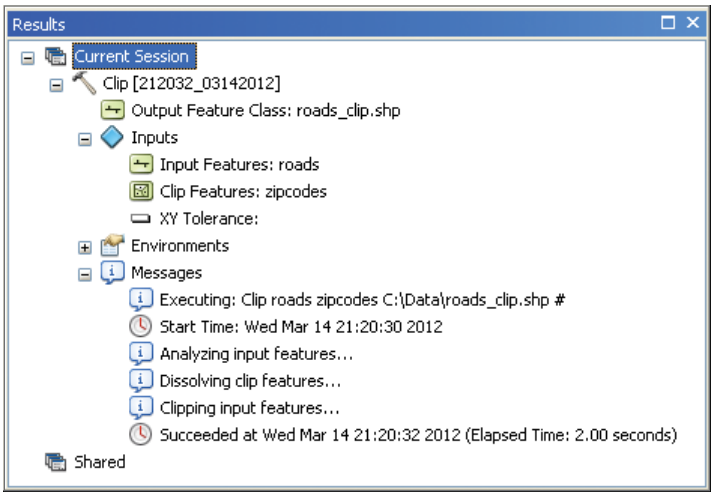
Environment settings are revisited again in later chapters, especially how to transfer settings from one script to another.

5.10 Working with tool messages

When tools are run, messages are written about the tool's success or failure of execution. Communication between tools and users is accomplished using messages. Typical information in these messages consists of the following:

- The exact time when running the tool started and ended
- The parameter values used to run the tool
- Information about the progress in running the tool (information messages)
- Warnings of potential problems in running the tool (warning messages)
- Any errors that prevented the tool from running (error messages)

When a tool is run from ArcToolbox, these geoprocessing messages appear on the progress dialog box when background processing is disabled. When a tool is run from the Python window, only error messages that indicate a particular situation prevented the tool from running appear. When a tool is run from within an ArcGIS for Desktop application, messages also appear in the Results window. A typical Results window after a tool has been run looks like the example in the figure. ➔



When a stand-alone Python script is run, these messages are not added to the Results window. Instead, you can obtain them from within the script. You can print the messages, write them to a file, or query them.

All messages have a severity property. This property is an integer with a value of 0 (information), 1 (warning), or 2 (error). Table 5.1 describes these three levels of severity in more detail.

Table 5.1 Severity of messages

Severity	Description
Severity = 0 Information message	Information messages provide information about tool execution. This includes general information such as the tool's progress, the start and completion time of the tool, and information about the tool results.
Severity = 1 Warning message	Warning messages indicate a possible problem. This could be a situation that may cause a problem during the tool's execution or a situation where the result may not be what you might expect. Warning messages do not prevent a tool from being executed, but they do warrant inspection.
Severity = 2 Error message	Error messages indicate that the tool has failed. Typically, this means that one or more parameter settings are invalid.

Both warning and error messages are accompanied by a six-digit ID code. The ID codes are documented, and the description of each ID code may be helpful in identifying the causes of potential problems and how they can be dealt with.

Messages from the last tool run are maintained by ArcPy and can be retrieved using the `GetMessages` function. This function returns a single string containing all the messages from the tool that was last run. The messages can be filtered by providing the severity argument.

The basic syntax for retrieving messages and printing them is

```
print arcpy.GetMesssages()
```

For example, when a Clip tool is run, the messages can be retrieved as follows:

```
import arcpy
arcpy.env.workspace = "C:/Data"
infc = "streams.shp"
clipfc = "study.shp"
outfc = "result.shp"
arcpy.Clip_analysis(infc, clipfc, outfc)
print arcpy.GetMessages()
```

Running the code results in a list of messages similar to the following:

```
Executing: Clip C:\Data\streams.shp C:\Data\study.shp C:\Data\result.shp #
Reading Features...
Cracking Features...
Assembling Features...
Succeeded at Fri Apr 30 17:12:05 2010 (Elapsed Time: 2.00 seconds)
```

Individual messages can be retrieved using the `GetMessage` function (note, this is different from the `GetMessages` function). This function has only one parameter, which is the index position of the message. For example, the following code retrieves the first message only:

```
print arcpy.GetMessage(0)
```

The result is

```
Executing: Clip C:\Data\streams.shp C:\Data\study.shp C:\Data\result.shp #
```

Note: *The index position starts at zero (0).*

The number of messages from the last tool that is run can be obtained using the `GetMessageCount` function. This function is particularly useful for retrieving just the last message. Since you typically will not know in advance how many messages may have resulted from running a tool, you can use the message count to retrieve the last message. The code to obtain the message count is

```
arcpy.GetMessageCount()
```

To retrieve the last message only, you would use the following:

```
count = arcpy.GetMessageCount()
print arcpy.GetMessage(count-1)
```

The result would be

```
Succeeded at Fri Apr 30 17:12:05 2010 (Elapsed Time: 2.00 seconds)
```

In addition to getting the number of messages, you can also query the maximum severity of the messages using the `GetMaxSeverity` function as follows:

```
print arcpy.GetMaxSeverity()
```

In the prior example of running the Clip tool, running the code would return the value of 0 because there are only information messages.

Although the `GetMessage`, `GetMessageCount`, and `GetMaxSeverity` functions are useful, in practice the `GetMessage` function is the most widely used. Messages are most important when a tool fails and so the `GetMessage` function is commonly used in combination with an error-handling technique. This is covered in chapter 11.

The functions discussed thus far allow you to retrieve the messages from the last tool that was run because the messages are maintained by ArcPy. However, as soon as another tool is run, you can no longer retrieve messages from tools run prior to that. To retrieve messages even after multiple tools have been run, the result class can be used to create a result object. The result object can then be used to retrieve and interpret geoprocessing tool messages. So rather than a tool being run for output files, the result of the geoprocessing operation is returned as an object. For example:

```
import arcpy
arcpy.env.workspace = "C:/Data"
result = arcpy.GetCount_management("streams.shp")
```

The result class has a number of properties and methods. The `messageCount` property returns the number of messages and `getMessage` returns a specific message. For example, running the following code retrieves the number of messages, followed by the last message:

```
import arcpy
arcpy.env.workspace = "C:/Data"
result = arcpy.GetCount_management("streams.shp")
count = result.messageCount
print result.getMessage(count-1)
```

Notice that the syntax is similar, but not identical, to using the general message function. When using `arcpy.GetMessage()`,you are calling a function, whereas using `<objectname>.getMessage()`, you are retrieving the properties of an object. The result class has a number of advantages over calling message functions, most notably the fact that messages can be maintained after running multiple tools. The result class also has a number of additional properties and methods, including options to count the number of outputs and the ability to work with specific outputs from a geoprocessing tool.

5.11 Working with licenses

Running geoprocessing tools requires a license for an ArcGIS product, such as ArcGIS for Desktop or ArcGIS for Server. This is true for running a tool from ArcToolbox within any ArcGIS for Desktop application, but it also applies to running a stand-alone Python script that uses tools. If a license is unavailable, a tool will fail and return an error message. Higher license levels provide access to a greater number of tools. For example, if you have an ArcGIS for Desktop Basic license and attempt to run a tool that is part of ArcGIS for Desktop Advanced only, the tool will fail to run.

A tool from an ArcGIS extension, such as ArcGIS 3D Analyst for Desktop or ArcGIS Spatial Analyst for Desktop, requires an additional license for that extension. Thus, if you do not have an ArcGIS Spatial Analyst license and attempt to run a tool that is part of the Spatial Analyst toolbox, the tool will fail to run. For example, in the following code, the Slope function from the Spatial Analyst module is called with a raster digital elevation model (DEM) as input:

```
import arcpy
arcpy.sa.Slope("C:/Data/dem", "DEGREE")
```

If no Spatial Analyst license is available, the following error is generated:

```
ERROR 000824: The tool is not licensed.
Failed to execute (Slope).
```

Every tool checks to ensure that it has the proper license. To avoid having a script fail because of an unlicensed tool, it is a good practice to check for licenses at the beginning of the script.

Licenses for the following six products can be checked:

1. `arcview`
2. `arceditor`
3. `arcinfo`
4. `engine`
5. `enginegeodb`
6. `arcserver`

The product level can be set by importing the appropriate product module prior to importing ArcPy. For example, to set the desktop product license level to ArcGIS for Desktop Basic (formerly ArcView), a script would start with the following code:

```
import arcview
import arcpy
```

Notice that Python still references the original license levels.

The desktop product level cannot be set once ArcPy is imported. If a license is not explicitly set, the license is initialized based on the highest available license level the first time ArcPy is imported. In general, therefore, there is no need to set the product level in Python, and you will not see many scripts that include it.

Note: *The setting of the product and extensions is necessary only within stand-alone scripts. If you are running tools from the Python window or using script tools, the product is already set from within the application, and the active extensions are based on the Extensions dialog box.*

The `CheckProduct` function can be used to check whether the requested license is available. For example, the following code determines whether an ArcGIS for Desktop Advanced (formerly ArcInfo) license is available:

```
if arcpy.CheckProduct("arcinfo") == "Available":
```

Note: *The only parameter for* `CheckProduct` *is a single string, which should be one of the six product codes listed previously. It is not case sensitive, so "arcinfo" is the same in Python as "ArcInfo."*

The result of the `CheckProduct` function is a string, which can have one of five possible values:

1. `AlreadyInitalized`—license has already been set in the script.
2. `Available`—requested license is available to be set.
3. `Unavailable`—requested license is unavailable to be set.
4. `NotLicensed`—requested license is not valid.
5. `Failed`—system failure occurred during the request.

The `ProductInfo` function reports what the current product license is, as follows:

```
import arcpy
print arcpy.ProductInfo()
```

The `ProductInfo` function returns a string value that has the value `NotInitialized` if no license has been set yet, or else it returns the current product license.

Licenses for extensions can be retrieved for use in a script and returned once they are no longer needed. This is analogous to checking out licenses from within ArcMap or ArcCatalog using the Customize > Extensions option. The `CheckExtension` function is used to check whether a license is available to be checked out. For example:

```
import arcpy
arcpy.CheckExtension("spatial")
```

Note: *Similar to the product codes, license names are not case sensitive.*

The `CheckExtension` function returns a string and can have one of four possible values:

1. `Available`—requested license is available to be set.
2. `Unavailable`—requested license is unavailable to be set.
3. `NotLicensed`—requested license is not valid.
4. `Failed`—system failure occurred during license request.

Once the availability of a license is determined, the `CheckOutExtension` function can be used to actually obtain the license. Once a script runs the tools that required the particular license, the `CheckInExtension` function can be used to return the license to the license manager. For example, the following code first checks the availability of a license for ArcGIS 3D Analyst, and if the license is available, a license is obtained and then returned after the tool is finished running:

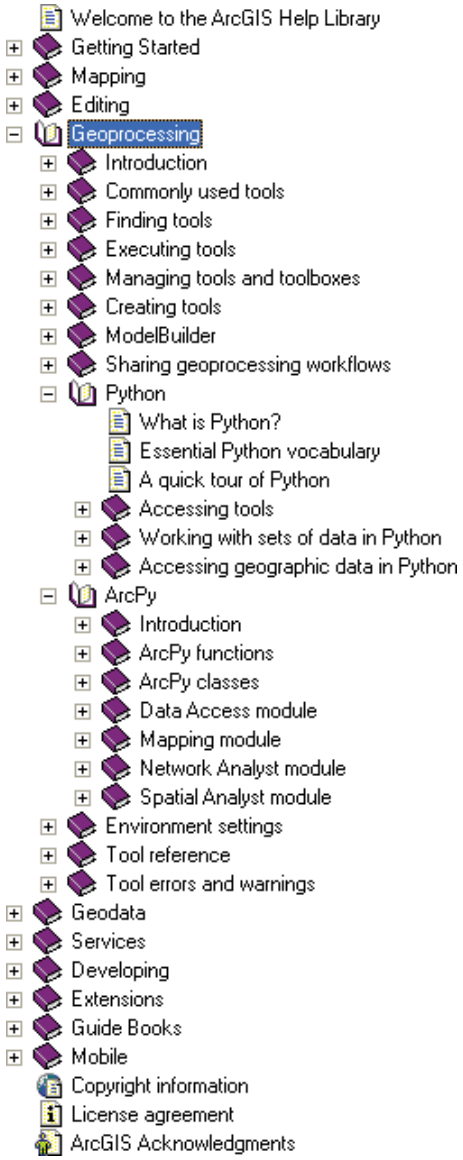
```
import arcpy
from arcpy import env
env.workspace = "C:/Data"
if arcpy.CheckExtension("3D") == "Available":
    arcpy.CheckOutExtension("3D")
    arcpy.Slope_3d("dem", "slope", "DEGREES")
    arcpy.CheckInExtension("3D")
else:
    print "3D Analyst license is unavailable."
```

The `CheckOutExtension` function returns a string, and there are three possible return values: (1) `NotInitialized`, (2) `Unavailable`, and (3) `CheckedOut`. Typically, you would use the `CheckExtension` function to first determine the availability of the license before using the `CheckOutExtension` function. The `CheckInExtension` function returns a string, and there are three possible return values: (1) `NotInitialized`, (2) `Failed`, and (3) `CheckedIn`.

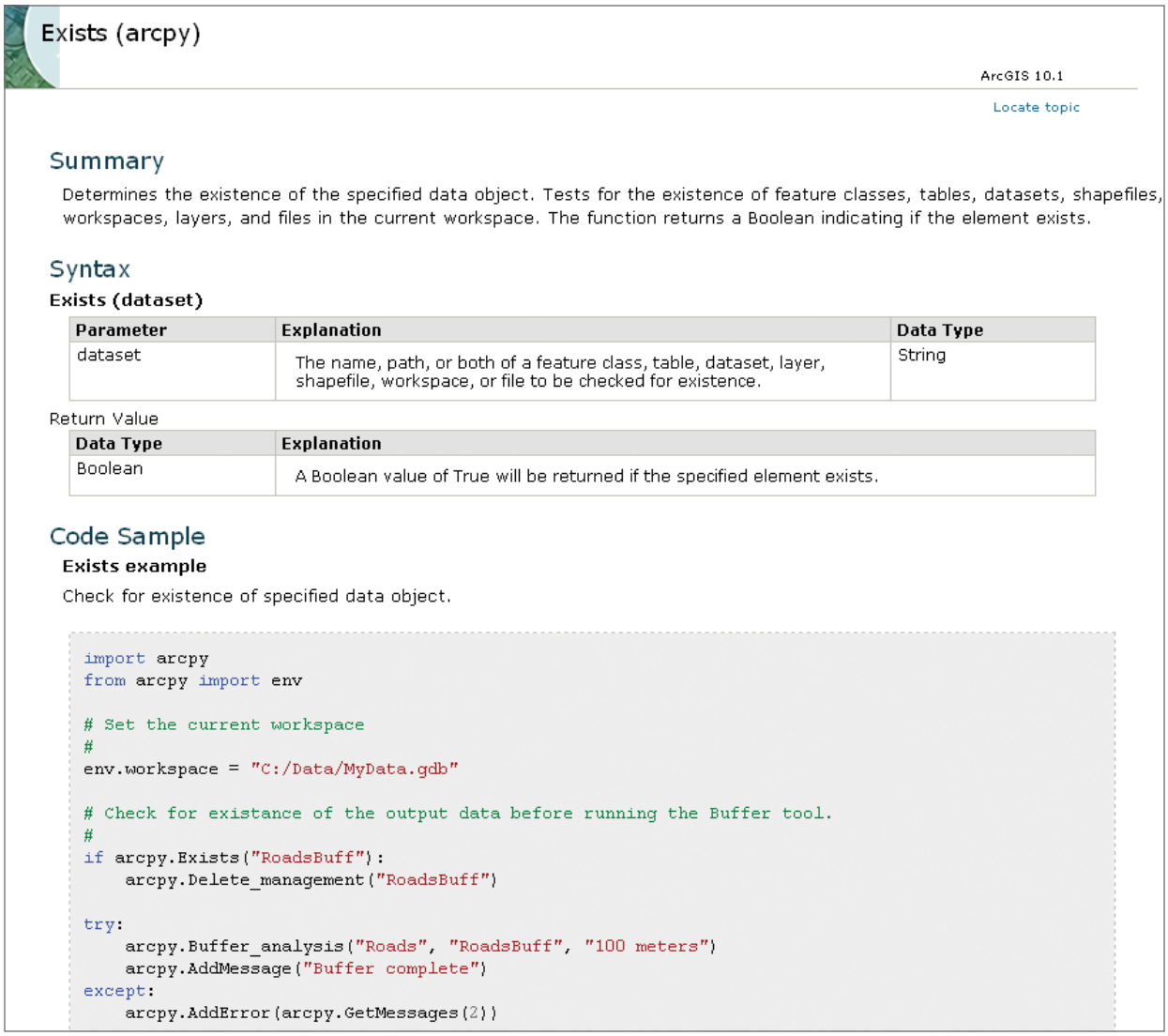
5.12 Accessing ArcGIS Desktop Help

The ArcGIS Help Library contains separate sections on using Python for geoprocessing and using the ArcPy site package. In ArcMap on the menu bar, click Help > ArcGIS Desktop Help. Or on the taskbar, click the Start button, and then, on the Start menu, click All Programs > ArcGIS > ArcGIS for Desktop Help > ArcGIS 10.1 for Desktop Help, and under the Contents tab, click Geoprocessing > Python. This documentation contains explanations of the basics of Python and how to carry out geoprocessing tasks in ArcGIS for Desktop using Python code.

Also under the Contents tab, click Geoprocessing > ArcPy for the ArcPy site package. All ArcPy functions and classes are listed and described in detail, and sample code is provided. There are also separate sections on the Data Access, Mapping, Network Analyst, and Spatial Analyst modules. ➔



All the Help pages contain sample code. For example, the Help page for the `Exists` function (under ArcPy > ArcPy functions > General data functions) looks like the example in the figure.



Exists (arcpy)

ArcGIS 10.1

Locate topic

Summary

Determines the existence of the specified data object. Tests for the existence of feature classes, tables, datasets, shapefiles, workspaces, layers, and files in the current workspace. The function returns a Boolean indicating if the element exists.

Syntax

Exists (dataset)

Parameter	Explanation	Data Type
dataset	The name, path, or both of a feature class, table, dataset, layer, shapefile, workspace, or file to be checked for existence.	String

Return Value

Data Type	Explanation
Boolean	A Boolean value of True will be returned if the specified element exists.

Code Sample

Exists example

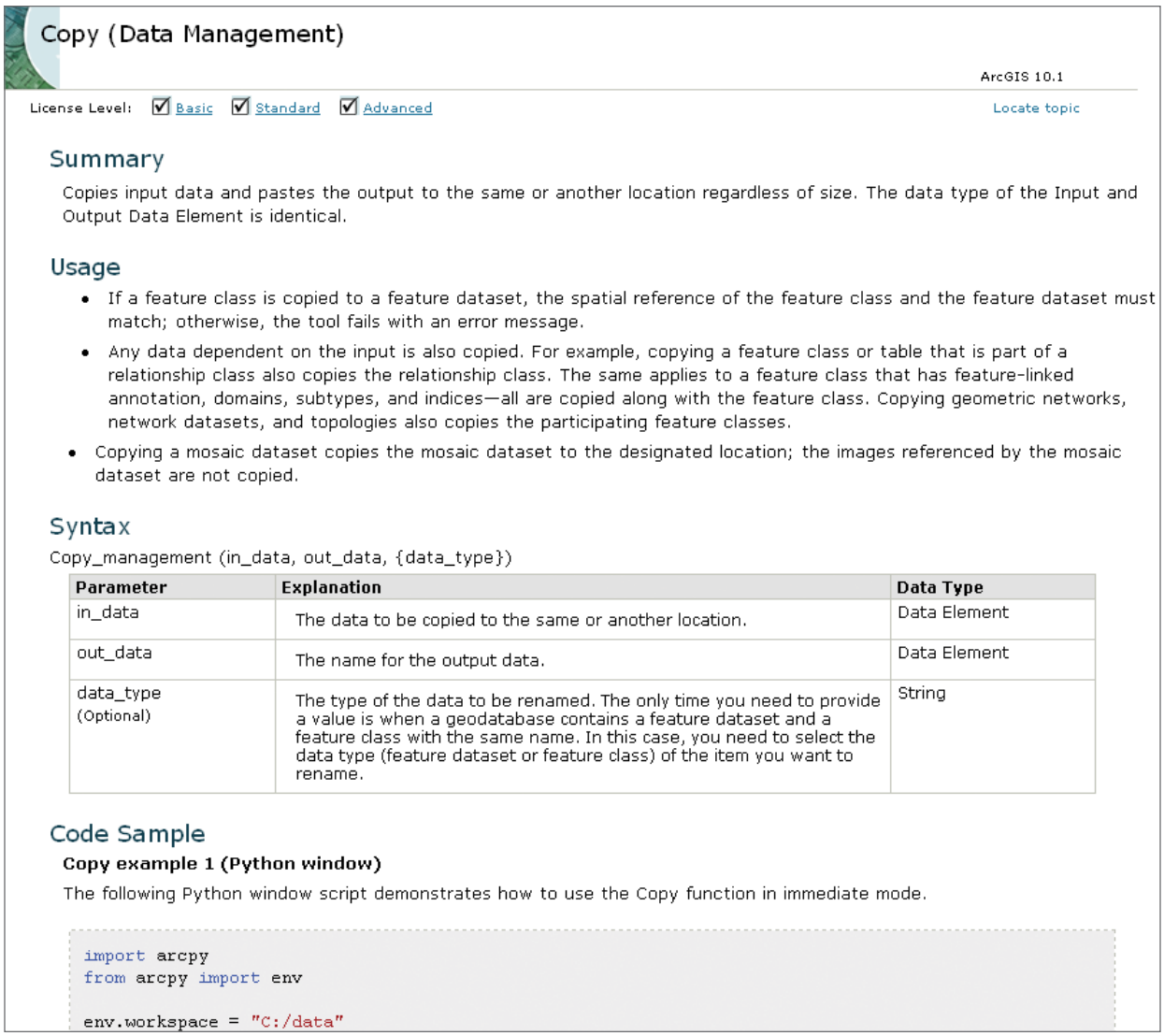
Check for existence of specified data object.

```
import arcpy
from arcpy import env

# Set the current workspace
#
env.workspace = "C:/Data/MyData.gdb"

# Check for existance of the output data before running the Buffer tool.
#
if arcpy.Exists("RoadsBuff"):
    arcpy.Delete_management("RoadsBuff")

try:
    arcpy.Buffer_analysis("Roads", "RoadsBuff", "100 meters")
    arcpy.AddMessage("Buffer complete")
except:
    arcpy.AddError(arcpy.GetMessages(2))
```

The sample code assumes familiarity with Python scripting but typically provides good examples of how a particular function (or class) is used. You can copy all or parts of the code and paste it into a Python editor.

Individual geoprocessing tools also include Help with an explanation of how the tool works, as well as the tool syntax and sample Python code. Help can be accessed in several ways: (1) in ArcToolbox, right-click a tool and click Help; (2) on a tool dialog box, click the Show Help button to show the Help panel for the tool, and then click the Tool Help button; and (3) in ArcGIS 10.1 Help, under the Contents tab, click Geoprocessing > Tool reference—this allows you to navigate to a tool's Help page using the same structure of toolboxes, toolsets, and tools as in ArcToolbox.

For example, the Help page for the Copy tool in the Data Management toolbox (under the General toolset) looks like the example in the figure.



Copy (Data Management)

ArcGIS 10.1

License Level: ☑ Basic ☑ Standard ☑ Advanced Locate topic

Summary

Copies input data and pastes the output to the same or another location regardless of size. The data type of the Input and Output Data Element is identical.

Usage

- If a feature class is copied to a feature dataset, the spatial reference of the feature class and the feature dataset must match; otherwise, the tool fails with an error message.
- Any data dependent on the input is also copied. For example, copying a feature class or table that is part of a relationship class also copies the relationship class. The same applies to a feature class that has feature-linked annotation, domains, subtypes, and indices—all are copied along with the feature class. Copying geometric networks, network datasets, and topologies also copies the participating feature classes.
- Copying a mosaic dataset copies the mosaic dataset to the designated location; the images referenced by the mosaic dataset are not copied.

Syntax

Copy_management (in_data, out_data, {data_type})

Parameter	Explanation	Data Type
in_data	The data to be copied to the same or another location.	Data Element
out_data	The name for the output data.	Data Element
data_type (Optional)	The type of the data to be renamed. The only time you need to provide a value is when a geodatabase contains a feature dataset and a feature class with the same name. In this case, you need to select the data type (feature dataset or feature class) of the item you want to rename.	String

Code Sample

Copy example 1 (Python window)

The following Python window script demonstrates how to use the Copy function in immediate mode.

```
import arcpy
from arcpy import env

env.workspace = "C:/data"
```

The syntax is provided, with a detailed explanation of each parameter. The code samples are typically quite short but demonstrate the specific use of the tool.

Points to remember

- The ArcPy site package introduced in ArcGIS version 10 provides access to the Python geoprocessing functionality in ArcGIS. It is the successor to the `ArcGISscripting` module from earlier versions. ArcPy is organized in modules, functions, and classes.

- All geoprocessing tools in ArcGIS are provided as functions. Once ArcPy is imported to a Python script, you can run all the geoprocessing tools found in the standard toolboxes that are installed with ArcGIS. The syntax for running a tool is `arcpy.<toolname_toolboxalias>(<parameters>)`. The documentation on each tool provides details on the required and optional parameters needed for a tool to run. Additional nontool functions in ArcPy are available to support geoprocessing tasks.

- Classes in ArcPy are used to create objects. Commonly used classes are the `env` class and the `SpatialReference` class. The syntax for setting the property of a class is

  ```
  arcpy.<classname>.<property> = <value>.
  ```

- Messages that result from running a tool can be retrieved using message functions, including `GetMessages`, `GetMessage`, and `GetMaxSeverity`. Messages can consist of information, warning, or error messages.

- Several functions are available to check available licenses for products and extensions, to check out licenses, and to check licenses back in.

- ArcGIS for Desktop Help contains many examples of Python code, including the Help page for individual geoprocessing tools.