Chapter 5
ArcGIS and Python

Abstract ArcGIS provides a palette of sophisticated tools for processing and analyzing geographic data. There are several ways in which these tools can be used. For example, ArcToolbox tools can be run from ArcToolbox by clicking on the tool and filling out a form to specify the parameters; they can be run from ModelBuilder Models, and they can be run from Python scripts using the arcpy package. This chapter discusses ArcToolbox, ModelBuilder, ArcCatalog, Python's ArcGIS capabilities, the arcpy package, arcpy functions, and environment settings.

Chapter Objectives
After reading this chapter, you'll be able to do the following:

- Describe the ArcToolbox hierarchy.
- Search for tools in ArcCatalog.
- Locate tool help on ArcGIS Resources online.
- Export a script from a visual workflow model.
- Modify and run exported scripts.
- Preview geoprocessing output.
- Release locks on data.
- Explain, in general terms, the capabilities of the `arcpy` package.
- Define the Python terms *module* and *package*.
- Set geoprocessing environment variables.

5.1 ArcToolbox

ArcGIS provides a large suite of tools for processing data. The ArcToolbox  panel in ArcCatalog (and ArcMap) lists the toolboxes—3D Analyst, Analysis, Cartography, and so forth (Figure 5.1). The tools are grouped into toolboxes by the type of actions they perform and each toolbox contains toolsets that further group the tools by their functionality. Each toolbox () and toolset () can be expanded to show the contents. Figure 5.2 shows the Extract, Overlay, Proximity, and Statistics toolsets in the Analysis toolbox. In this figure, the Proximity toolset is also expanded.

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_5

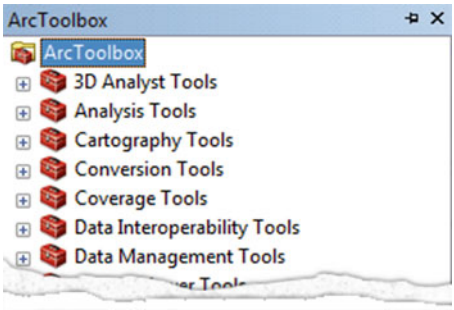


Figure 5.1 The ArcToolbox application contains numerous toolboxes (only a few shown are here).

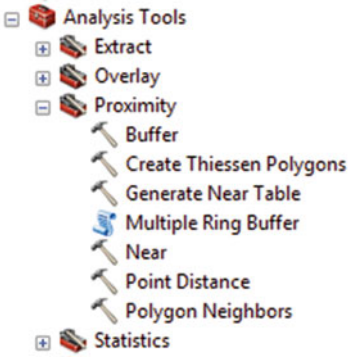


Figure 5.2 The Analysis toolbox and the Proximity toolset.

This toolset contains six tools, Buffer, Create Thiessen Polygons, etc. Though the icons vary on the third level, these are all tools. A tool can be run from ArcToolbox, by clicking on the tool, filling out the form that it launches, and clicking 'OK'.

The availability of tools depends on the license level of ArcGIS desktop installed (basic, standard, or advanced). Some tools, such as Spatial Analyst tools, are not available at the basic and standard levels. When these tools are available, an extension needs to be checked out for the tools to become functional. For example, to use Spatial Analyst tools with an advanced install, the Spatial Analyst extension must be checked out (ArcCatalog > Customize > Extensions > Check Spatial Analyst). Scripts need to check out these extensions too, as shown in the upcoming chapter.

The 'Search' panel available in ArcCatalog (and ArcMap) is useful for navigating the tools. Click the 'Search' button to open this panel, select the 'Tools' option, and type a tool name. Figure 5.3 shows the results of a search. Click on a tool in the results (e.g., Buffer (Analysis) Tool) and it opens the GUI to run the tool. The form has a 'Tool Help' button which will launch the local ArcGIS Desktop help for that tool. As you work through the examples in this book, you can locate tools in this way and become familiar with their functionality by reading the help and running them using the GUI interface.

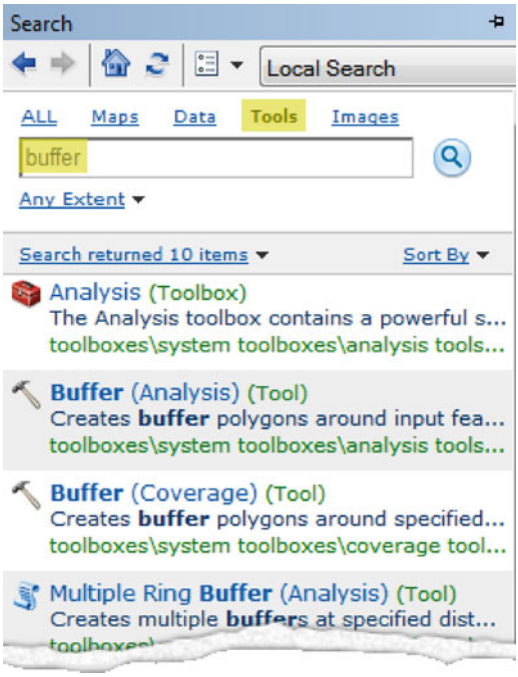


Figure 5.3 The search panel.

5.2 ArcGIS Python Resources

The 'ArcGIS Resources' site (resources.arcgis.com) is the foremost reference you will need for working with Python in ArcGIS. The site provides full documentation for ArcGIS Python functionality. The 'Search ArcGIS Resources' box is indispensable for working with Python in ArcGIS. Use the search box to get a list of pages matching your query and use the search filters to narrow the search (Figure 5.4). For example, enter 'buffer' in the search box. This returns thousands of results including blogs, bug reports, web mapping help, and so forth. Narrow the search by using the 'Help' and 'Desktop' filters as shown Figure 5.5.

Notice that the results are different from the ArcCatalog search results for the same term (Figure 5.3); the ArcGIS Desktop help is organized differently than the online help. The online help provides the most current, comprehensive documentation. A set of descriptive identifiers is provided for each link in the online help. The Buffer (Analysis) link has the identifiers 'Tool Reference' and 'ANALYSIS' (Figure 5.6). The last identifier shows the date the content was last modified. Each ArcGIS tool has a 'Tool Reference' page that corresponds to the built-in help for that tool. Chapter 6 discusses components in 'Tool Reference' pages.

This site is referred to as the 'ArcGIS Resources' site in this book. Key search terms will be provided to direct you to specific help topics within this site.

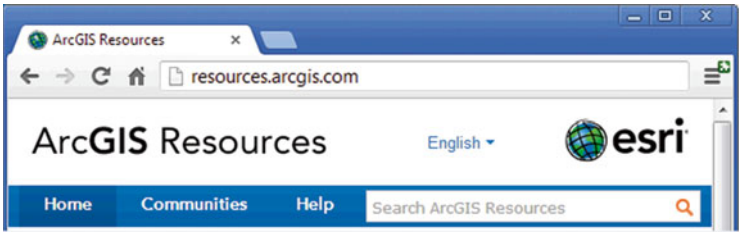


Figure 5.4 Search for help in the ArcGIS Resource help box.

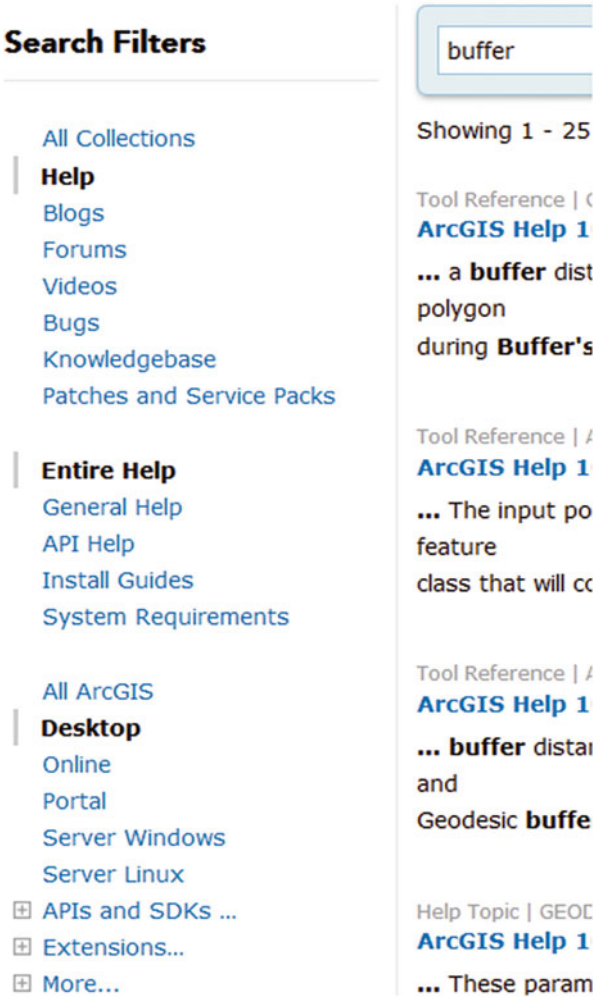


Figure 5.5 Filter the search for relevant results. The search shown here only returns 'Help' for ArcGIS 'Desktop' topics.

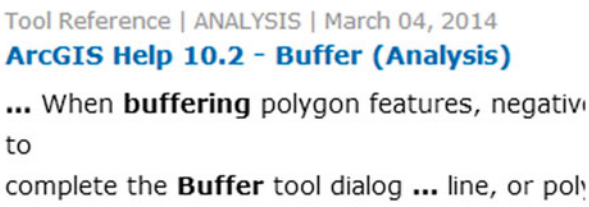


Figure 5.6 Descriptive identifiers such as 'Tool reference' and 'ANALYSIS' appear above each search result.

5.3 Exporting Models

Python can call almost all the tools in ArcToolbox and, in this way, repetitive processes can be automated. Before we begin writing scripts from scratch, we'll start with an example automatically generated by the ArcGIS ModelBuilder application. ModelBuilder is an application built into ArcCatalog (and ArcMap) that allows users to create a workflow visualization, called a *model*. Models not only visualize the workflow, but can also be can be run to execute the workflow. ArcGIS Toolbox tools can also be run via ModelBuilder; Tools can be dragged into the model panel and connected to create the workflow. When a model runs, it executes tools and the underlying code statements that correspond to pieces of the model. The underlying code can be exported to a Python script and we can compare the workflow visualization with the code. Follow steps 1–3 to create and export a simple model.

1. In ArcCatalog, to create a model like the one in Figure 5.7:

- Launch ModelBuilder from the button on the Standard toolbar.
- Open ArcToolbox with the ArcToolbox button on the Standard toolbar.
- Locate the Buffer (Analysis) tool in ArcToolbox (ArcToolbox > Analysis Tools > Proximity > Buffer)
- Click the Buffer tool and drag it into ModelBuilder from its toolbox in ArcToolbox. A rectangle labeled 'Buffer' will appear.
- Right-click on the new Buffer rectangle > Make variable > From Parameter > Input Features.
- Right-click on the Buffer rectangle (again) > Make variable > From Parameter > Distance.
- Double click on the Input Features oval and browse to: 'C:/gispy/data/ch05/park.shp'.
- Double click on the Distance oval and set it to 100 feet.
- Right-click on the Input Features oval > rename 'inputFeatures'.
- Right-click on the Distance oval > rename 'distance'.
- Right-click on the Output Feature Class oval > rename 'outputFeatures'.
- Check that the result looks like Figure 5.7.

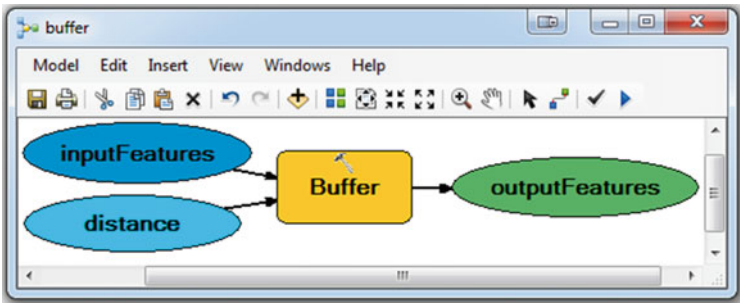


Figure 5.7 ModelBuilder Model to buffer input features.

```
1  # -*- coding: utf-8 -*-
2  # ---------------------------------------------------------------------------
3  # buffer.py
4  # Created on: 20xx-05-22 12:17:03.00000
5  #   (generated by ArcGIS/ModelBuilder)
6  # Description:
7  # ---------------------------------------------------------------------------
8
9  # Import arcpy module
10 import arcpy
11
12
13 # Local variables:
14 inputFeatures = "C:\\gispy\\data\\ch05\\park.shp"
15 distance = "100 Feet"
16 outputFeatures = "C:\\gispy\\data\\ch05\\park_Buffer.shp"
17
18 # Process: Buffer
19 arcpy.Buffer_analysis(inputFeatures, outputFeatures, distance, "FULL", "ROUND", "NONE", "")
```

Figure 5.8 A script exported from the model shown above.

2. Run the model to confirm that it works (Model menu>Run).
3. Export the model as a script (Model>Export>Pythons Script). This should generate a script like Figure 5.8, shown with line numbers to the left of the code. Open the script in PythonWin to view the code.

To compare the model and script, we'll look at each line of code.

- Lines 1–7, 9, 13, and 18 are comments.
- Line 10 imports `arcpy`. This line of code enables the script to use ArcGIS commands. We'll talk more about this in a moment.
- Lines 14–16 are assignment statements for string variables, `inputFeatures`, `distance`, and `outputFeatures`. These correspond to the model variables, the ovals that specify input and output for the tool. The string literal assigned to the script variables depends on the value assigned in ModelBuilder. For example, on line 13, `inputFeatures` is being assigned the value 'C:\\gispy\\data\\ch05\\park.shp' because the model variable was given this value before the model was exported.

- Line 19 calls the Buffer (Analysis) tool. Running a tool in Python is like calling a function. We call the tool and pass arguments into it. The tool does our bidding and creates output or returns values. The variables and string literals in the parentheses are passing information to the tool. The Buffer tool requires three input parameters (the other parameters are optional). These required parameters are represented by the ovals in our model. The first three arguments in the Python correspond to these parameters. When the code was exported, it filled in the default values for the rest of the parameters. In summary, line 19 acts like the rectangle in the model; it creates a buffer around the input features by the given distance and saves the results in the output features.

With these observations, it's possible to get a feel for the connection between the model components and the lines of code in the script. In theory, exported models could be used as a starting point for scripts, but this approach can be cumbersome for several reasons. First, scripting enables a more flexible, reusable complex workflow, including functionality beyond ArcGIS geoprocessing. Second, exported scripts usually require modification to perform as desired, making it more efficient to modify existing code samples than to build and export models.

The model/script comparison above provides some intuition for how the Python code is working, though more detailed explanation is needed. The next sections address this need with discussions on importing `arcpy`, using dot notation with `arcpy`, and calling tools.

5.4 Working with GIS Data

Each time you run a Python script that generates geoprocessing output, you will want to check the results. Esri geographic data features and the tables associated with them can be viewed in the ArcCatalog 'Preview' tab. Browse to a shapefile in the ArcCatalog 'Catalog Tree' and select the 'Preview' tab. Select 'Geography' preview from the bar at the bottom of the Preview pane to view the geographic features. Then select 'Table' to see the associated attribute table. Figures 5.9 and 5.10 show the geography and table views of 'park.shp'. Only seven rows of the table are shown in Figure 5.10. There are 426 rows in total, one data record for each polygon.

When the data is being viewed in ArcCatalog, it is locked, so that other programs can't modify it simultaneously. A file with an 'sr.lock' extension appears in Windows Explorer when the data is locked. For example, the file could be named something like 'park.shp.HAL.5532.5620.sr.lock' on a computer named 'Hal'. When you perform processing on the file in a Python script, you need to make sure that the data is not locked. To unlock the data after previewing it in ArcCatalog, select the parent workspace ('C:/gispy/data/ch05' in Figures 5.9 and 5.10) and refresh ArcCatalog (press F5). Selecting another file within the same workspace and refreshing the Catalog Tree will not release the lock; the parent workspace must be refreshed.

Note Unlocking the data before performing Python geoprocessing is critical, else the script may give unforeseen errors.

To see the geographic view of more than one file at a time, you need to use ArcMap. To view your data in ArcMap, the simplest approach is to browse to the data in the ArcCatalog tree embedded in ArcMap and drag/drop it onto a blank map. As long as ArcMap is still running, data used in this way will be locked. Even if the map document is closed, the locks may not be released until the program itself is exited.

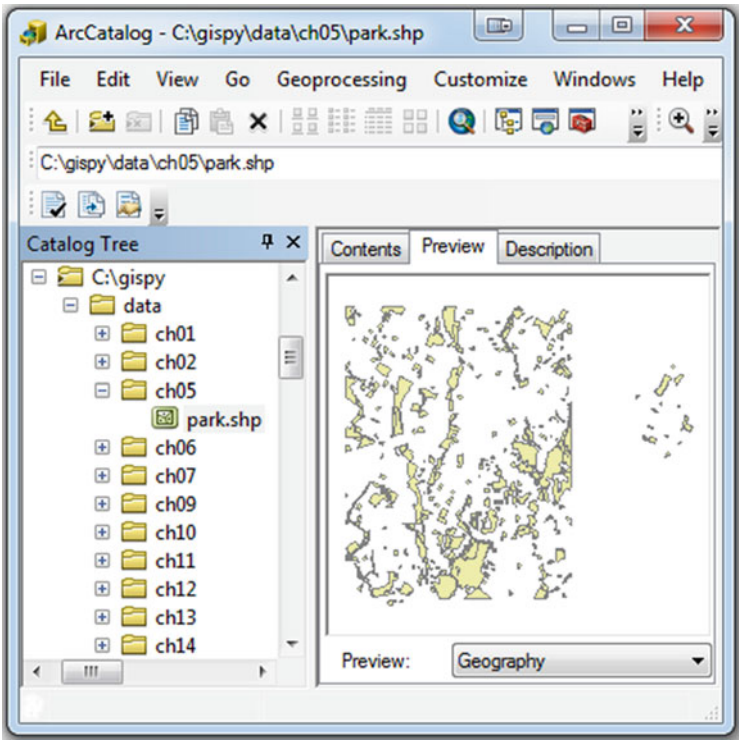


Figure 5.9 Geography preview of 'park.shp'.

5.5 ArcGIS + Python = arcpy

Now that you're familiar with ArcToolbox, ModelBuilder, and ArcCatalog data previews, it is time to introduce the `arcpy` Package. Geoprocessing scripts begin by importing `arcpy` (e.g., Figure 5.8); The `arcpy` package is Python's

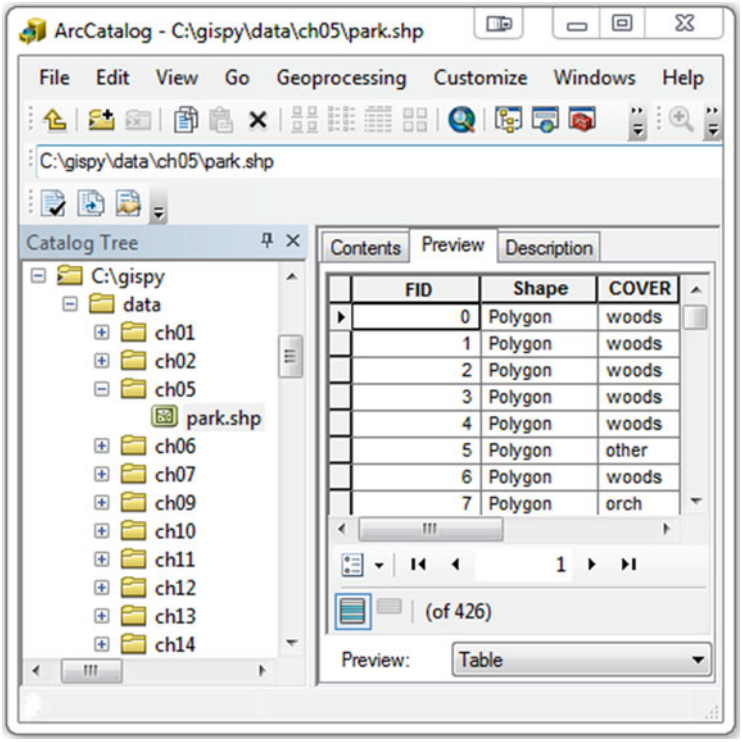


Figure 5.10 Table preview of 'park.shp'.

means for accessing ArcGIS from Python. To use ArcGIS functionality in Python, a script needs to import `arcpy`. The terms `import`, package, and `arcpy` are explained here:

- The keyword `import` is used to reference a module or a package. This provides access to functionality beyond the built-in Python functionality. As discussed in Chapter 2, the term following the `import` keyword is a module or a package.
- Recall that a *module* is a single Python script ('.py' file) containing tightly related definitions and statements. A *package* is a special way of structuring a set of related Python modules. A package is a directory containing modules and sometimes subpackages, which also contain modules. A module named '__init __ .py' tells Python that the directory contains a package. Modules structured within a package and items in those modules can be accessed using the dot notation.
- `arcpy` is a package installed with ArcGIS. `arcpy` can also be thought of as a Python object that has numerous methods, including geoprocessing tools. Search under the ArcGIS install and you will find an `arcpy` directory. This is the package being imported. `arcpy` provides a variety of functionality that we'll be using throughout the remainder of the book. Table 5.1 lists the main `arcpy` topics covered in this book.

Table 5.1 Highlights of `arcpy` functionality.

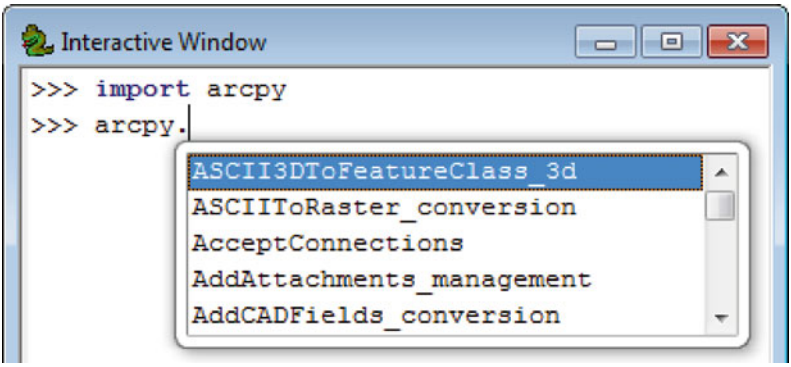
ArcGIS topic	Functionality	Code sample
Tools	Call ArcToolbox tools	`arcpy.Buffer_analysis('park.shp', 'output.shp', '1 Mile')`
Other functions	Licensing, data management, and other miscellaneous needs	`arcpy.CheckOutExtension('Spatial')`
Environment variables	Set and get environment variable values	`arcpy.env.overwriteOutput = True`
Describe	Describe data properties	`arcpy.Describe('park.shp')`
Listing data	List items (e.g., rasters, feature classes, and workspaces)	`arcpy.ListFeatureClasses()`
Cursor	Read/modify attribute table elements	`arcpy.da.SearchCursor('park.shp', fieldNames)`
Messaging	Get and print geoprocessing messages	`arcpy.GetMessages()`
Mapping	Manipulate existing map layers, add layers to maps, modifying surrounds, manipulate symbology	`arcpy.mapping.MoveLayer(df, refLayer, moveLayer, 'BEFORE')`

The `arcpy` package has an object-oriented design which can be defined using object-oriented terms, some of which were already used in Chapters 3 and 4:

- Everything in Python is an *object*, including modules and packages. `arcpy` is an object. `arcpy` also uses objects, such as a `ValueTable` object, a `Describe` object, and a `Result` object which are discussed in upcoming sections.
- Objects have methods associated with them. A *method* is a function that performs some action on the object. Methods are simply a specific type of functions. The terms 'calling methods', 'passing arguments', and 'returning values' apply to methods just as they apply to functions (see Section 2.4.4).

Dot notation generates context menus for `arcpy`, showing a list of available `arcpy` methods and properties. When you use dot notation with strings and lists, context menus automatically appear, but these are built-in data types. In order to view context menus for `arcpy`, you must first import `arcpy` to give Python access to the information it uses to populate the context menu. In other words, it recognizes the word `arcpy` as a package. Once you have imported `arcpy`, within a PythonWin session, the context menus are available for the entire session. Try the code shown in the screen shot below to see the context menu for `arcpy` properties are case-sensitive and must be spelled correctly. Selecting choices from the context menu ensures accurate spelling and capitalization. To use the context menu, you can start typing a name and the menu will scroll to the closest choice. Pressing the 'Tab' key

insert the current selection into the code. If you don't see your choice, you know your spelling or capitalization is incorrect. The `arcpy` menu contains a list of `arcpy` functions, environment settings, tools, and modules.



With each software release, `arcpy` functionality grows. It would be difficult to learn about every `arcpy` object, method, and property. In fact, you don't need to know all these details; they can be referenced in the help documentation. Instead, we aim for a general exposure to available capabilities. Figure 5.11 provides an overview of `arcpy` functionality. Symbols are used for properties, methods, and objects as shown in the key (bottom right). In this abbreviated object model diagram, the boxes enclose functionality categories and only a few examples are shown for each category. The contents are not exhaustive, but Figure 5.11 will provide a reference for the `arcpy` discussion. Packages, modules, and classes are Python constructs for organizing code. These constructs can contain functions (or methods) and properties. To see a complete list of functions/methods and properties for any of these constructs, search the online ArcGIS Resources site. The diagram is available as arcpyCheatSheet.pptx in the Chapter 5 data directory (C:/gispy/data/ch05). Chapters 6, 9, 11, 17, and 24 reference this document. If possible, print this document and keep it handy for upcoming chapters.

5.6 arcpy Functions

The `arcpy` functions (top left box in Figure 5.11) provide support for geoprocessing workflows. For example, functions can be used to list datasets, retrieve a dataset's properties, check for existence of data, validate a table name before adding it to a geodatabase, or perform many other useful scripting tasks. The syntax for calling functions in the `arcpy` package uses the dot notation with `arcpy` before the dot and the function name after the dot:

```
arcpy.functionName(argument1, argument2, argument3,...)
```

In the following example, the function name is 'CheckExtension' and it takes one argument, the extension code, '3D', for the 3D Analyst toolbox:

```
>>> import arcpy
>>> arcpy.CheckExtension('3D')
u'Available'
```

The response means the 3D Analyst Extension license is available to be checked out. The `u` in front stands for unicode, a way of encoding strings (see Section 3.6). But the string encoding doesn't have any practical repercussions for our interests, so you can ignore this. If you print the value, with the built-in `print` function, the `u` will not be included.

```
>>> print arcpy.CheckExtension('3D')
Available
```

Notice, the example above did not import `arcpy`. For a given PythonWin session, once `arcpy` is imported, it doesn't need to be imported again. But if PythonWin is closed and reopened, `arcpy` needs to be imported again.

Functions that don't require arguments still need to use parentheses. For example, the `ListPrinterNames` function lists the printers available to the caller and takes no arguments:

```
>>> arcpy.ListPrinterNames()
[u'Use Mac Printer', u'Send To OneNote 2010', u'Microsoft
XPS Document Writer', u'Fax', u'Adobe PDF']
```

Many `arcpy` functions return values. An assignment statement can capture the return value in a variable. The following code creates an `arcpy Point` object. The `CreateObject` function returns a `Point` object and it is stored in the variable named `pt`. The second line prints the value of this variable, a 2-dimensional `Point` object located at `(0,0)`:

```
>>> pt = arcpy.CreateObject('Point')
>>> pt
<Point (0.0, 0.0, #, #)>
```

The `arcpy` functions serve a variety of scripting needs, some dealing with administrative concerns such as licensing (e.g., `CheckExtension`), others dealing with data management. For example, the `Exists` function takes one argument, a dataset and checks if it exists:

```
>>> arcpy.Exists('C:/gispy/data/ch05/park.shp')
True
```

Other functions deal with topics such as geodatabase management, messaging, fields, tools, parameters, and cursors. Search for an 'alphabetical list of `arcpy` functions', on the ArcGIS Resources site to see a complete list of functions. Technically, the ArcToolbox tool functions are `arcpy` functions, but these are listed separately elsewhere on the site. The syntax for calling tools is similar to the syntax for calling other `arcpy` functions. Before learning to call tools you need to know a little about managing environment settings. The syntax for environment settings uses dot notation as well.

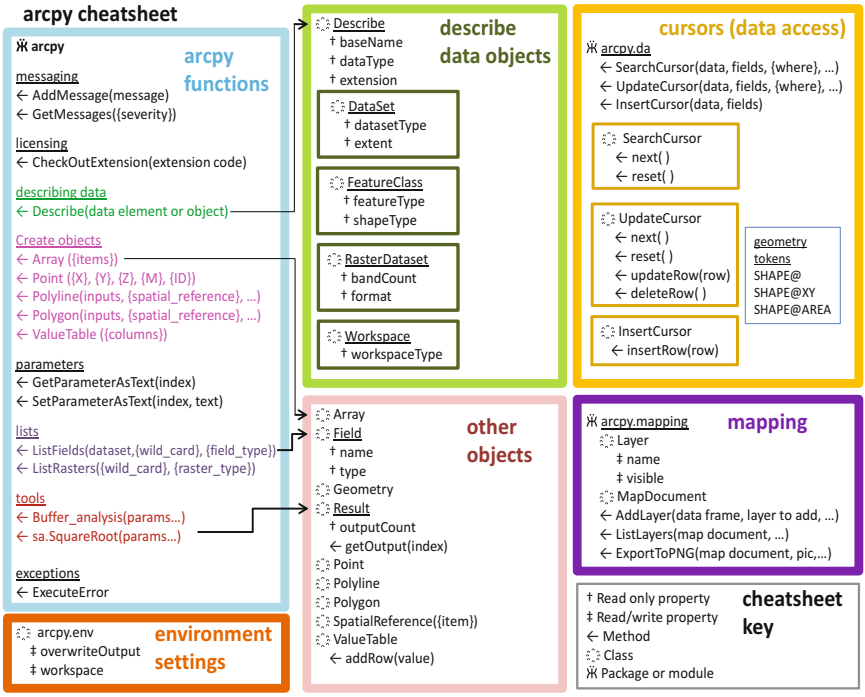


Figure 5.11 Main components of `arcpy` functionality with a few examples listed for each item.

5.7 Environment Settings

Each tool has settings it uses to execute an operation, such as a tolerance or output location. The *environment settings* are conditions that can be applied to all tools within an application (e.g., all operations in ArcCatalog can be limited to an extent set in the environment settings). ArcGIS has default values for these settings and users can modify the values via a dialog box in ArcGIS (Geoprocessing > Environments launches the dialog box shown in Figure 5.12). Python commands

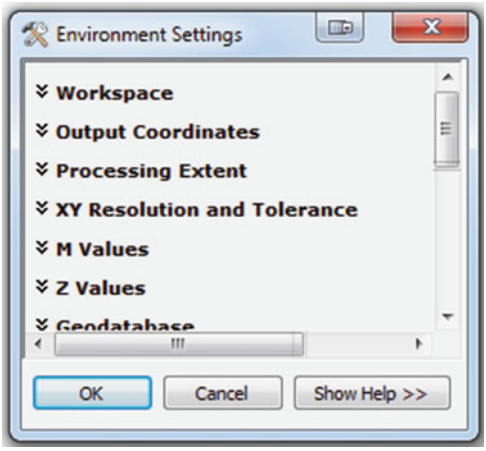


Figure 5.12 Interface for manually modifying ArcMap environment settings.

can also be used to get or set the values of these settings. `arcpy` has an `env` class, a special structure for holding related properties. The `env` properties control the environment settings. `env` belongs to `arcpy` and the properties belong to the `env` class, so the property names have two dots, one after `arcpy` and one after `env`. The format for setting these properties is:

```
arcpy.env.property = value
```

The format for getting these properties is:

```
variable = arcpy.env.property
```

The workspace path and the overwrite output status are important environment properties. The workspace path specifies a structure such as a directory or file geodatabase that contains pertinent data. Tools look for input data in the workspace and place the output in the workspace. If the input data resides in the workspace, only the name of the file needs to be used. `arcpy` will automatically search in the workspace. Similarly, the workspace is the default location for output, unless otherwise specified.

```
>>> # Setting the current workspace path
>>> arcpy.env.workspace = 'C:/Data/Forestry'
>>> # Getting the current workspace path
>>> mydir = arcpy.env.workspace
>>> mydir
u'C:/Data/Forestry'
```

Setting the workspace is simply a string assignment. No checking is done at this point to ensure that the workspace exists. This only occurs when a tool attempts to use the workspace to read or write data. The overwrite output status, either `True` or `False`, controls whether or not existing files are allowed to be overwritten by output from tools. The built-in constants `True` or `False` must be capitalized. The overwrite output status can also be set to 1 or 0 (`1` for `True` and `0` for `False`). The default value for the `overwriteOutput` properties is `False`.

```
>>> arcpy.env.overwriteOutput
False
```

This protects the user from unintentionally overwriting a file, but is inconvenient during software development when scripts need to be run more than once for testing. For complex scripts, it's useful to set overwriteOutput to `True` placing this statement near the beginning of the script, after `arcpy` is imported, but before any tool calls are made:

```
>>> arcpy.env.overwriteOutput = True
```

Since Python is case-sensitive, be sure to use lower camel case for `overwriteOutput`. For environment settings, no error will appear if the capitalization is incorrect; it will simply not work as expected. In the following example, we use the wrong capitalization and no error is reported, but the value of the `overwriteOutput` property is not changed to `False`:

```
>>> arcpy.env.overwriteoutput = False
>>> arcpy.env.overwriteOutput
True
```

Other environment variables may be useful for specific problems. For example, when you are creating raster data sets, you may want to set the tile size property which specifies the height and width of data stored in blocks. The default size is 128 by 128 pixels:

```
>>> arcpy.env.tileSize
u'128 128'
```

Type the following line of code to print a complete list of available environment properties:

```
>>> arcpy.ListEnvironments()
```

5.8 Key Terms

ModelBuilder models
`import` keyword
`arcpy` package
Python package
Python module
Environment settings
ArcToolbox
ArcCatalog tool search
ArcGIS Resources site
ModelBuilder
Model parameter
ArcCatalog geography and table previews
Data locks

5.9 Exercises

1. Use the given steps to create and test 'aggregate.py'. The steps walk through creating a model, exporting it, and running it as a script. The script will call the Aggregate Polygons (Cartography) tool, a data summary technique which groups features that are very close together. If input features are within a specified aggregation distance, this script will combine them into a single unit. The figures below show an original shapefile in Figure 5.13a and an aggregated version in Figure 5.13b.
   (a) Step 1: *Create a new toolbox*. Browse to C:\gispy\sample_scripts\ch05 in ArcCatalog and create the new toolbox there.
   (b) Step 2: *Create the model*. The model should use the tool named 'Aggregate Polygons'. Browse to the tool (ArcToolbox > Cartography Tools > Generalization > Aggregate Polygons), then drag and drop it onto the model. Give the tool two input parameters: input features and aggregation distance.

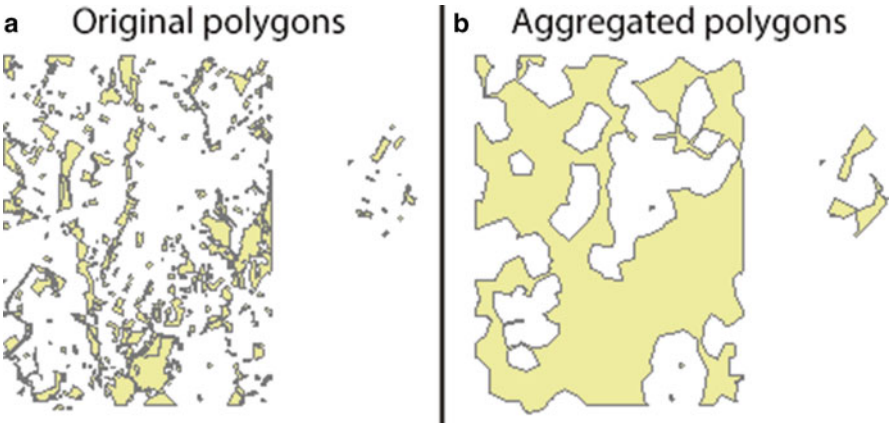
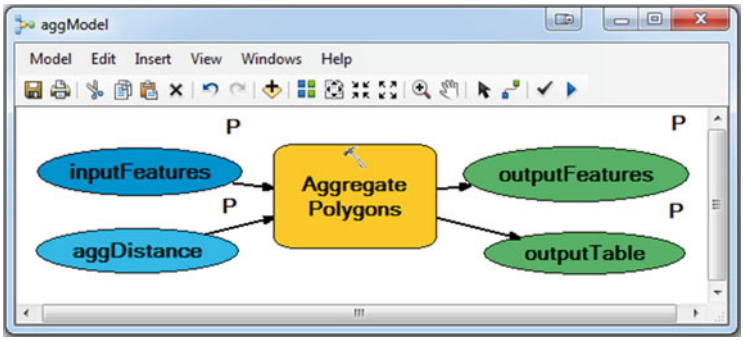


Figure 5.13 park.shp with no aggregation (**a**) and after 1500 foot aggregation (**b**).

Double-click on the input variables and give them default values of 'C:\\gispy\\data\\ch05\\park.shp' and 100 feet. The model will become colored once these values are set. Rename the variables to meaningful, succinct names: inputFeatures, aggDistance, outputFeatures, and outputTable. Make all four variables model parameters (right-click on the oval>Model Parameter and a 'P' appears by the oval). Set the `inputFeatures` 'P' first and `aggDistance` 'P' second so that your script has the input features as the first parameter and the aggregation distance as the second parameter.



(c) Step 3: *Export the model*. Call it aggregateExport.py and save it in 'C:\gispy\sample_scripts\ch05'.

(d) Step 4: *Run the model in PythonWin*. Open the script in PythonWin and run the script with arguments: In the 'Run Script' window 'Arguments' text box, insert the arguments the script needs, a shapefile and an aggregation distance. Try the following input example:

C:/gispy/data/ch05/park.shp "1500 feet"

Be sure to separate the two arguments by a space. Click 'OK' to run the script. Next, in the script, change the names of the output features and table to "C:\\gispy\\data\\ch05\\park_Agg500.shp" and "C:\\gispy\\data\\ch05\\park_Agg500_Tbl", respectively, and run the script again with an aggregation distance of 500 feet, by modifying the aggregation distance value in the 'Arguments' text box.

(e) Step 5: *Check the output*. View the two output shapefiles in ArcCatalog by selecting each one in turn in the 'Catalog Tree' and selecting the 'Preview' tab. Select the 'Table' view from the drop-down menu at the bottom of the 'Preview' tab and observe the number of records in the output files as compared to the input file.

2. Try the following steps to experiment with environment settings and answer the questions that follow:

   1. Launch ArcCatalog.
   2. Search for the Buffer (Analysis) tool in the search window  .

3. Double-click on the tool link to launch the tool dialog.
4. Drag C:/gispy/data/ch05/park.shp into the 'Input Features' parameter box.
5. Observe the 'Output Feature Class' path.
6. Cancel the tool.
7. Open the environment settings dialog (Geoprocessing>Environments…)
8. Click on Workspaces to expand this category.
9. Set 'Scratch Workspace' to C:/gispy/scratch
10. Click OK to accept the changes and close the dialog box.
11. Repeat Steps 2–5

    Follow-up questions:
    - Explain the effect of changing 'Scratch Workspace' Environment Settings in this example.
    - Write a line of Python code that, in a script, would achieve the same affect as step 9.

3. Follow the steps in Section 5.3 to create a module like the one shown in Figure 5.7. Then export the model and open the exported script. It should look like the one in Figure 5.8. Run the script and check the output. Run the script again. It should throw an error. Add a line of code to the script so that you can run it twice without getting an error. What line of code did you add? Where did you place it?