# Chapter 23
# ArcGIS Python GUIs

**Abstract** The graphical user interfaces (GUIs) discussed in the previous chapter create file browsing interfaces with simple Python calls to the `tkFileDialog` module. Some applications may require additional input (other than files/directories). Or you may want to create one GUI dialog that accepts multiple input parameters. If the application is meant to be run in an environment where ArcGIS is installed, *Script Tools* and *Python toolboxes* provide a solution for building GUIs with these characteristics. 'Script Tool' is an ESRI term for an ArcGIS construct that resides within a custom toolbox and points to an underlying Python script. By using a Script Tool, you can create a custom GUI that looks similar to the built-in ArcGIS tool GUIs. You can also add a button to one of the ArcGIS menus, so that users can launch the tool with one button click. Python toolboxes are another way to create GUIs that look just like Script Tool GUIs. A Python toolbox is a text file containing Python classes to define the toolbox and tools. Script Tools are a good way to learn about the GUI options available; Python toolboxes are an efficient way to develop tools, once you understand these options. This chapter introduces Script Tools and then steps through the various customization techniques. Last, Python toolboxes are discussed.

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

- Create an ArcGIS graphical user interface for a Python script.
- Specify GIS data type interface parameters.
- Set types, default values, direction, multiplicity, and other properties of parameters.
- Enable a user to input hand-digitized points, lines, or polygons features.
- Create a toolbar button to launch the GUI.
- Add output to a map.
- Set the symbology of an output parameter.
- Display progress as processes run.
- Implement dynamic behavior in a GUI.
- Build a Python toolbox GUI.
- Explain the difference between a Script Tool and a Python toolbox tool.

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_23

## 23.1 Creating a Script Tool

A Script Tool can be thought of as a wrapper for running a Python script; It points to a Python script, passes user input into the script, runs the script, and receives output from the script. The workflow for building a Script Tool involves creating a Python script and a toolbox first, then using the Script Tool wizard to set up the Script Tool. Script Tools reside within custom ArcGIS toolboxes. A custom toolbox is one created by a user, not one of the built-in ArcGIS toolboxes (In Figure 23.1, 'practice.tbx' is a custom toolbox). You create a toolbox, then add a Script Tool to it. This launches a Script Tool wizard. The wizard steps you through setting up the Script Tool properties, including a list of parameters which are used to pass arguments to the Python script. Once created, the Script Tool appears in ArcCatalog in the table of contents under its toolbox. The icon for a Script Tool looks like a scroll (In Figure 23.2, 'printTextFiles' is a Script Tool).

**Note** To create a Script Tool, you need three things:

1. A Python script.
2. A custom toolbox.
3. A list of desired script parameters and their types.

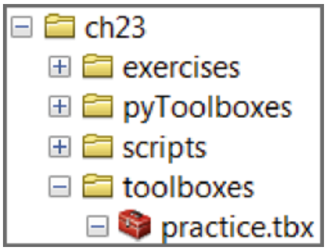


**Figure 23.1** A custom toolbox, 'practice.tbx' viewed in ArcCatalog.

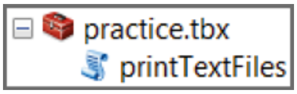


**Figure 23.2** A ScriptTool, 'printTextFiles' viewed in ArcCatalog.

Let's step through an example. Since a Script Tool is merely a pointer to a script, we first need an existing script. We'll use a sample script which lists the text files with '.txt' extensions within a directory (see 'textLister.py' in Example 23.1) and we'll create a custom toolbox to house a new Script Tool. This sample script doesn't take any parameters. Upcoming examples will show how to use parameters. This example simply focuses on creating and running a Script Tool. Get some hands-on experience by following these steps:

1. Open and run 'C:\gispy\sample_scripts\ch23\scripts\textLister.py' in an IDE such as PythonWin or PyScripter. It should print text file names:

```
>>> cfactors.txt
crop_yield.txt
poem.txt
RDUforest.txt
report.txt
wheatYield.txt
xyData2.txt
```

2. Create a toolbox by browsing to 'C:\gispy\sample_scripts\ch23\toolboxes' in ArcCatalog.
3. Add a Script Tool to the new toolbox by right-clicking on 'practice.tbx' in ArcCatalog and choosing Add>Script… This launches a Script Tool wizard to customize the Script Tool step by step.
4. In the first pane of the wizard, set the script 'Name' and 'Label' to 'printText-Files'. The label appears in the ArcCatalog table of contents; It can have spaces, but the name can't. For simplicity, you can use the same value for both. Check 'Store relative path names'. This is an important choice, as we'll discuss shortly. Click 'Next >' to go to the next wizard pane.
5. We want this Script Tool to point to 'textLister.py'. To set the 'Script File', browse to 'textLister.py' in 'C:\gispy\sample_scripts\ch23\scripts'. Leave the default values for the check boxes. Running a Script Tool in-process improves efficiency, so this is usually the preferred approach. Click 'Next >' to proceed.
6. The final pane pertains to script parameters. Since 'textLister.py' needs no arguments, we'll leave this page as-is. Finally, click 'Finish' to exit the wizard. You can always come back and change any of the choices made during the initial setup by right-clicking on the Script Tool and selecting 'Properties…'.
7. A Script Tool named 'printTextFiles' appears in ArcCatalog as in Figure 23.2.

Run the tool by double-clicking on it. A GUI with several buttons and a help panel appears as in Figure 23.3. The left panel states 'This tool has no parameters'. This is what we expect to see, since we didn't set up any parameters. Click 'OK' to run the Script Tool. When the Script Tool run has completed, it launches a window to report success or failure. We refer to this as the 'Geoprocessing Window'.

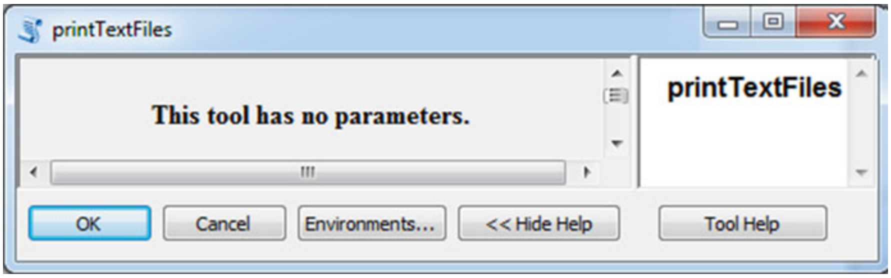


**Figure 23.3** A Script Tool GUI with no parameters.

**Example 23.1: Simple script for illustrating Script Tools.**

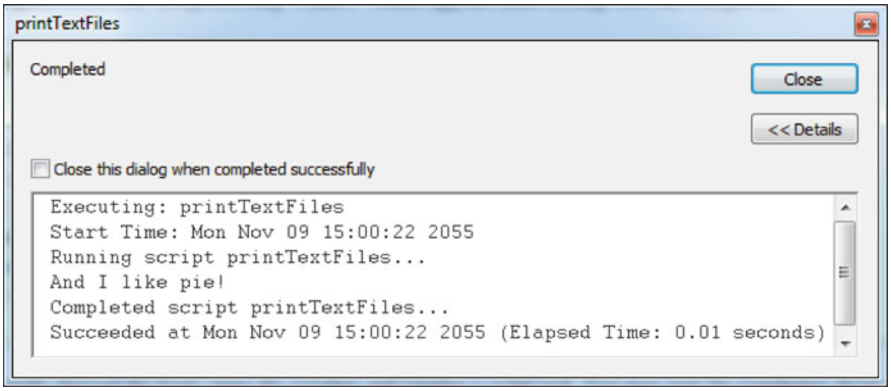
```
# textLister.py
# Purpose: Print the text file (.txt) names in the directory.
import arcpy, os
myDir = r'C:\gispy\data\ch23\smallDir'
fileList = os.listdir(myDir)
for f in fileList:
    if f.endswith(".txt"):
        print f

arcpy.AddMessage('And I like pie!')
```

This example showed the basic steps for creating and running a Script Tool. You can check your work against the 'printTextFiles' tool in 'practiceExamples.tbx'. Right-click on the Script Tool and select 'Properties…' to view the properties of an existing Script Tool. Meanwhile, did you notice that the Geoprocessing Window reported that the script ran successfully and mentioned something about pie, but did not print any text file names? This is because standard Python print statements do not appear in the Geoprocessing Window. The next section discusses two techniques for communicating with the user within Script Tools. Also, this GUI didn't take any input from the user. We'll be getting to that shortly.

**Note** The ArcToolbox 'Add toolbox' commands puts a toolbox in the default toolbox location. To control the location of the toolbox, right-click on the target directory in ArcCatalog and select New>Toolbox.

### *23.1.1 Printing from a Script Tool*

Until now, we've used the Python `print` keyword to communicate with the user. The print statement takes a string and displays it in the Interactive Window of an IDE. But printed messages don't appear in the Geoprocessing Window. To get



```
Executing: printTextFiles
Start Time: Mon Nov 09 15:00:22 2055
Running script printTextFiles...
And I like pie!
Completed script printTextFiles...
Succeeded at Mon Nov 09 15:00:22 2055 (Elapsed Time: 0.01 seconds)
```

**Figure 23.4** The 'Geoprocessing Window' which appears after a Script Tool has completed.

around this, you can use the `arcpy` `AddMessage` method. The `AddMessage` method takes one argument, a string message, and prints it in the Geoprocessing Window. In the Geoprocessing Window in Figure 23.4, as you may have already guessed, the following line generates the message (`'And I like pie!'`) wedged between the standard messages 'Running script…' and 'Completed script…':

```
arcpy.AddMessage('And I like pie!')
```

Python's print statement doesn't display in the Geoprocessing Window and `AddMessage` doesn't print in the Interactive Window:

```
>>> import arcpy
>>> message = 'And I like pie!'
>>> print message
And I like pie!
>>> arcpy.AddMessage(message)
>>>
```

This leads to a dichotomy: If you only use print statements, they will not appear in the Geoprocessing Window when you run a Script Tool. However, if you only use `AddMessage`, the message will not appear in the Interactive Window when you run the underlying script in an IDE. Since neither will print in both environments, the best solution is to use both types of statements to print output messages. For convenience, you can define a function to take a message and report it both ways:

```
def printArc(message):
    '''Print message for Script Tool and standard output.'''
    print message
    arcpy.AddMessage(message)
```

Example 23.2 calls `printArc` to print the number of files in a directory. Output from both the Python print and `AddMessage` statements looks like this:

```
Directory C:/gispy/data/ch23/smallDir contains 27 files.
```

Unlike the built-in `print` function, the `AddMessage` method does not accept comma separated values as input. To prepare messages for `AddMessage`, you must use concatenation or string formatting.

```
>>> print 5, 'miles'
5 miles

>>> arcpy.AddMessage(5,'miles')
TypeError: AddMessage() takes exactly 1 argument (2 given)
```

**Example 23.2: Print and AddMessage.**

```
# print4ScriptTools.py
# Purpose: Prints a directory's file count using
#          both 'print' and 'AddMessage'

import arcpy, os

def printArc(message):
    '''Print message for Script Tool and standard output.'''
    print message
    arcpy.AddMessage(message)

myDir = r'C:\gispy\data\ch23\smallDir'
# Lists all the files in the given directory.
fileList = os.listdir(myDir)
myMessage = 'Directory {0} contains {1} files.'.format(myDir,
                                                        len(fileList))

printArc(myMessage)
```

### *23.1.2 Making a Script Tool Button*

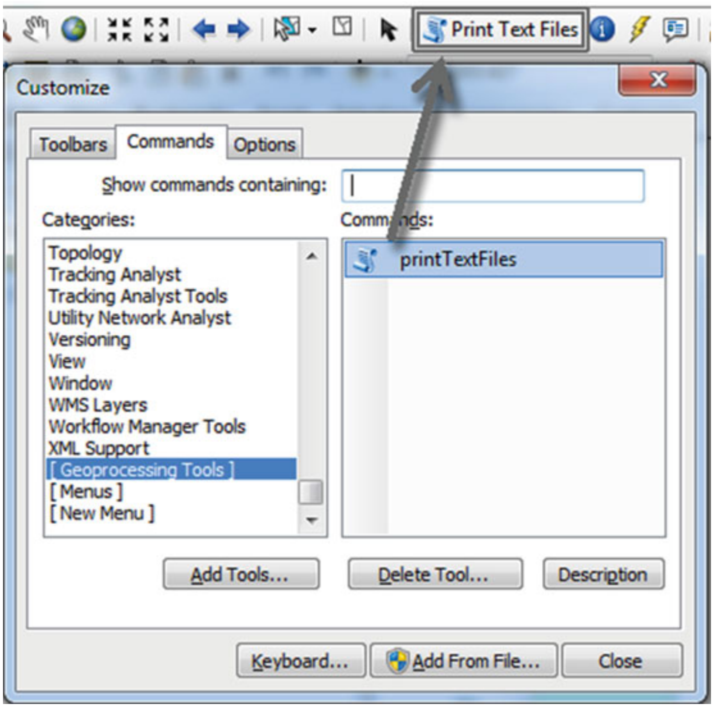
So far, we've been running Script Tools by double-clicking on them in ArcToolbox. You can also set up a button shortcut for a Script Tool. You can customize ArcMap (or ArcCatalog) to include this button on a toolbar for quick access to the tool. The button icon and text can be customized to suit the application.

To create a button on an ArcMap toolbar to launch a Script Tool, use the following steps:

1. Select ‘Customize’ > ‘Customize mode…’
2. Select the ‘Commands’ tab
3. Under ‘Categories’, scroll down to select ‘[Geoprocessing tool]’
4. Select ‘Add Tools…’
5. Browse to the custom toolbox > Select the Script Tool in toolbox > Select ‘Add’. The Script Tool appears under ‘Commands’.
6. Click on the Script Tool under ‘Commands’ and while holding the left mouse key down, drag the Script Tool between any existing buttons on any ArcMap toolbar. A black vertical bar appears when you’re between buttons in a position where the new button can be placed. Release the mouse button to drop it into position.
7. Before closing the ‘Customize’ dialog, right-click on the tool in the toolbar to modify the button text and image, as desired.
8. Click ‘Close’ on the ‘Customize’ dialog to save changes.

Changes to the location, icon image, and text can be made by reopening the ‘Customize’ dialog (‘Customize’ > ‘Customize mode…’). When not in customize mode, the button is locked. To remove the button, open the ‘Customize’ dialog and drag it back into the commands window.

Clicking the button launches the same GUI as double-clicking on the tool in ArcToolbox. This can be useful for sharing custom applications with novice ArcGIS users. All irrelevant toolbars can be hidden to reduce confusion.



### *23.1.3 Pointing to a Script*

Before we create GUIs with Script Tools, it's important to understand a few things about Script Tools and their Python scripts. Figure 23.2 shows the Script Tool in ArcCatalog, but if you browse to the same directory ('C:\gispy\sample_scripts\ch23\toolboxes') in Windows Explorer, you see toolbox files, but no Script Tool file. Script Tools do not appear in Windows Explorer; They are stored as part of a '.tbx' file. To see evidence of this, check the current file size of the 'practice.tbx' file (~6 KB). Add a dummy Script Tool to the toolbox (right-click on 'practice.tbx' in ArcCatalog and choose Add>Script…). Accept all the defaults (Just click 'Next', 'Next', 'Finish' without browsing to a script, etc.). Now check the size of the 'practice.tbx' file again (~7 KB). It's larger because it is now storing the additional Script Tool.

A Script Tool is part of the toolbox. To share a Script Tool with someone else, you need to give them the toolbox; the Script Tool will be visible to them when they view the toolbox in ArcCatalog. Since the Script Tool is just a pointer to a script, you must also give them the script. You may be wondering, what it means to say the Script Tool is a 'pointer to a script'? And will the Script Tool still point to the script when the toolbox is moved? To explore the relationship between Script Tools and scripts try some experiments, as described in the following steps:

1. Right click on the 'printTextFiles' Script Tool, select 'Properties…', and then select the 'Source' tab. Note the Python script specified by the 'Script file' path on this tab ('C:\gispy\sample_scripts\ch23\scripts\textLister.py'). Close the Script Tool properties by clicking 'OK'.
2. Browse to 'textLister.py' and open it. If you change the code in the underlying script, the Script Tool will adopt these changes immediately. Add another message to 'textLister.py' and save the script:

```
arcpy.AddMessage('***I like kale***!')
```

3. Double-click on the Script Tool to run it again and you'll see the additional message in the Geoprocessing Window. No updates need to be made to the Script Tool to see this change, because no information from the Python script is stored in the Script Tool. Only the path to the script is stored. This is what meant by 'pointing at the script'.
4. Since the path is the only link from the Script Tool to the script, if you rename or move the underlying script, the Script Tool will no longer be able to access the script. To see this, rename 'textLister.py' to 'textLister2.py' and run the Script Tool again. An exception will be thrown and reported in the Geoprocessing Window:

```
ERROR 000576: Script associated with this tool does not exist.
Failed to execute (printTextFiles).
```

5. Rename the script back to its original name, 'textLister.py', and run the Script Tool again. It's back to a working state.

6. Now move 'textLister.py' up one directory to 'C:\gispy\sample_scripts\ch23\'. Run the Script Tool again. You get the same error as when it was renamed. Though the script does exist, the Script Tool can't find it. Move the script back into the 'C:\gispy\sample_scripts\ch23\scripts' directory.

Take care when you share a Script Tool, so that the user doesn't encounter Error 000576. Maintain a relative path between the Script Tool and the script and set the Script Tool to use relative paths. Open the 'printTextFiles' Script Tool properties and select the 'General' tab. The 'Store relative path names' checkbox should be checked. This means we can move the Script Tool as long as we move the script to the same relative location. Relative to 'example.tbx', 'textPrinter.py' is one directory up out of 'toolboxes' and then one step down into a 'scripts' directory. Remember that moving the Script Tool is accomplished by moving the toolbox. To see what happens when you don't maintain a relative path with the script try the following:

1. Make a copy of the 'practice' toolbox and place it under 'C:\gispy\sample_scripts\ch23\sandbox\toolboxes\'.
2. Run the new copy of the Script Tool inside; It reports Error 000576.
3. To correct the error, you need to recreate the original relative path to the script. How can you achieve this?
4. Make a copy of the 'scripts' directory under 'C:\gispy\sample_scripts\ch23\sandbox'.
5. Test the new Script Tool again to see that it works. For this example, a relative path is maintained by copying both the 'toolboxes' directory and the 'scripts' directory into the same parent directory.

**Script Tool Concepts to Remember**

- Script Tools do not appear in Windows Explorer. They are part of a '.tbx' file.
- Since a Script Tool points to a script, you can change the script and see the updates immediately when you run the tool.
- If you rename, move, or remove the underlying script, the Script Tool will be broken.
- For portability, check 'Store relative path names'. Then copy the toolbox and the script maintaining the relative path.
- Script Tools don't have a debugger, so it's important to test Python scripts thoroughly outside the Script Tool.
- A simple way to maintain relative paths is to set up the Script Tools and scripts in the same directory and share that directory.

## 23.2 Creating a GUI

Now that you know how to create Script Tools and how the tools relate to underlying scripts, you're ready to use Script Tools to create GUIs. When you set up a Script Tool, the last pane of the Script Tool wizard contains a table for parameters. For the 'printTextFiles' example, we left this table empty. This is why this Script Tool says 'This tool has no parameters' when it is launched. In this section, we'll use the Script Tool wizard parameter pane to set up some parameters. This pane contains two boxes, one for the parameters and one for the parameter properties (Figure 23.5). A parameter is added by specifying a display name and data type in the parameter table. Parameter properties can be adjusted for each parameter. This section steps through an example that adds some simple parameters. Sections 23.2.1 and 23.2.2 contain details on parameter data types and properties.

When items are added to the parameter table, the Script Tool generates a widget for the parameters in the list. User interface elements (e.g., text boxes, buttons, check boxes, combo boxes, and list boxes) are commonly referred to as *widgets*. Widgets help the user make input choices by constraining the way input is accepted. For example, they may help the user browse to a file or select amongst several mutually exclusive choices.

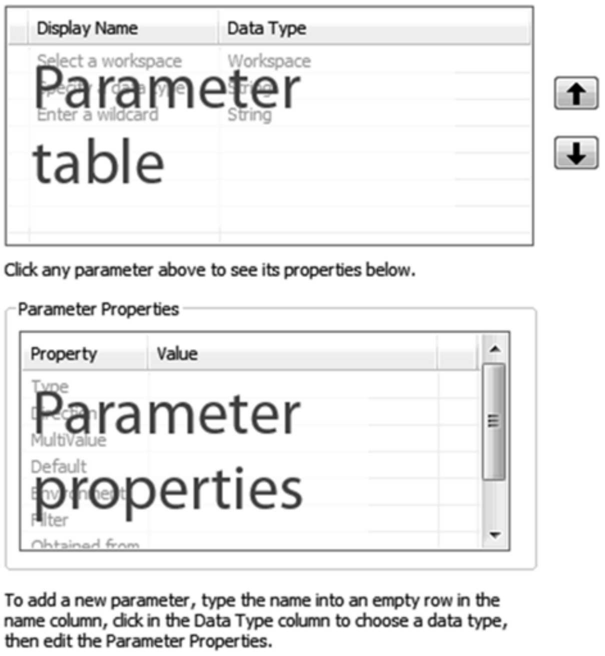

Display Name
Data Type
Select a workspace
Workspace
Enter a wildcard
String
Parameter table
Click any parameter above to see its properties below.
Parameter Properties
Property
Value
Type
MultiValue
Default
Filter
Obtained from
Parameter properties
To add a new parameter, type the name into an empty row in the name column, click in the Data Type column to choose a data type, then edit the Parameter Properties.

**Figure 23.5** The Script Tool wizard parameters page.

The Script Tool automatically adds a widget to the GUI for each input parameter. To see this, we'll create a Script Tool with three parameters. The sample script named 'deleter.py' (Example 23.2) deletes files in a workspace based on file type and name. For input, the script needs a workspace, a file type, and a wild card. If the data type is 'raster' or 'feature class', it deletes files of that type within the workspace that contain the wild card string in their names. If any other data type is given, it deletes files which have a file extension matching the wild card. Create a GUI for this script by building a Script Tool that lists three parameters with the following steps:

1. View 'C:/gispy/data/ch23/rastTester.gdb' in ArcCatalog and observe that there are over a dozen rasters in this workspace labeled '_out'. The file geodatabase 'rastTester_1.gdb' and 'rastTester_2.gdb' are identical copies of this workspace. Open and run 'deleter.py' in PythonWin with the following arguments to see how it works:

   C:/gispy/data/ch23/rastTester_1.gdb raster _out

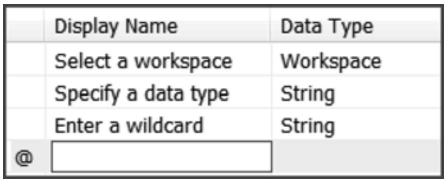
   When the script has stopped running, view 'rastTester_1.gdb' in ArcCatalog to confirm that all of the rasters with names containing '_out' have been deleted. No debugging is available within Script Tools; Hence, it is essential to test Python scripts before building Script Tools and then test modifications to the script outside of the Script Tool.

2. Add a second Script Tool to the 'practice.tbx' toolbox (Right-click on the toolbox in ArcCatalog and choose 'Add>Script…').

3. Name and label it 'deleteFiles'. Add a description of the Script Tool's purpose ('Delete files from a workspace based on their type and name.'). The description appears in the tool help box when the tool is launched (Figure 23.7). Also, check 'Store relative path names'. Click 'Next >' to proceed.

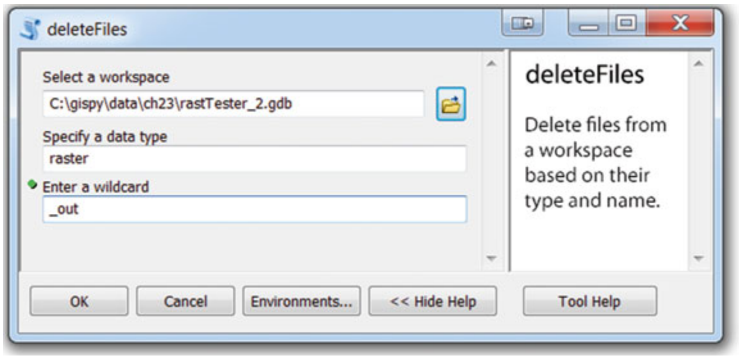
4. For the 'Script File', browse to 'deleter.py' in 'C:\gispy\sample_scripts\ch23\scripts'. Click 'Next >' to proceed.

5. Next, we'll add three script parameters, as needed by 'deleter.py'. Click in the left column in the parameter table and add display names as shown in Figure 23.6. Click in the right column and select the data types, Workspace, String, and String again (see Figure 23.6). You can use the mouse to scroll through the data types,



| | Display Name | Data Type |
|---|---|---|
| | Select a workspace | Workspace |
| | Specify a data type | String |
| | Enter a wildcard | String |
| @ | | |

**Figure 23.6** Parameter table for 'deleteFiles' Script Tool.


deleteFiles

Select a workspace
C:\gispy\data\ch23\rastTester_2.gdb
Specify a data type
raster
Enter a wildcard
_out

deleteFiles

Delete files from a workspace based on their type and name.

**Figure 23.7** A Script Tool GUI with three parameters.

but for greater efficiently, type the first letter of the data type (e.g., ‘w’ for ‘Workspace’) and then use the ‘down arrow’ key to scroll to the desired data type.

6. For now, we’ll use the default parameter properties, so click ‘Finish’ to exit the wizard.
7. Run the new Script Tool, ‘deleteFiles’, by double-clicking on it. The GUI has three widgets (a file browsing text box and two basic text boxes, in this case), as in Figure 23.7, one for each parameter. The three display names appear above the boxes to prompt the user.
8. Specify the input as shown in Figure 23.7 (Browse to ‘rastTester_2.gdb’ in the first box and specify ‘raster’ and ‘_out’ in the second and third boxes). Click ‘OK’ to run the tool. Then check the results in ‘rastTester_2.gdb’ by viewing it in ArcCatalog.

Adding three parameters in the ‘deleteFiles’ example created three boxes in its GUI. The first box has a browsing button, because the data type for this parameter was set to ‘Workspace’. This filters the workspace selection to directory, file geodatabase, and any other formats accepted by ArcGIS as a workspace. The browsing mechanism helps the user select a valid existing workspace. The other two widgets are simply text boxes that accept text without restrictions. The type of widget created for each parameter depends on the parameter data type and properties, as discussed next.

**Note** Script Tools can’t be stepped through with a debugger. Test Python scripts outside of the Script Tool in a Python IDE.

## Example 23.3

```
# deleter.py
# Purpose: Delete files from a workspace based on
#            their type and name.
# Usage: workspace datatype (raster, feature class,
#           or other) wildcard
# Sample input: C:/gispy/data/ch23/rastTester.gdb raster _out

import arcpy, os, sys

def printArc(message):
    '''Print message for Script Tool and standard output.'''
    print message
    arcpy.AddMessage(message)

arcpy.env.workspace = sys.argv[1]
fType = sys.argv[2]
wildcard = sys.argv[3]
substring = '*{0}*'.format(wildcard)

if fType == 'raster':
    data = arcpy.ListRasters(substring)
elif fType == 'feature class':
    data = arcpy.ListFeatureClasses(substring)
else:
    entireDir = os.listdir(arcpy.env.workspace)
    data = []
    for d in entireDir:
        if d.endswith(wildcard):
            data.append(d)
for d in data:
    try:
        arcpy.Delete_management(d)
        printArc( '{0}/{1} deleted.'.format(arcpy.env.workspace, d))
    except arcpy.ExecuteError:
        printArc( arcpy.GetMessages())
```

### *23.2.1 Using Parameter Data Types*

Script Tool parameter data types are Esri data types, not built-in Python data types; They are specialized structures related to ArcGIS data, like the data types that are listed on the geoprocessing tool help parameter tables. A few of the Esri data types have Python built-in data type equivalents as shown in Table 23.1.

**Table 23.1** Esri and Python equivalent data types.

| Esri data type | Built-in Python data type |
|---|---|
| Boolean | bool |
| Double | float |
| Long | int |
| String | unicode (string) |

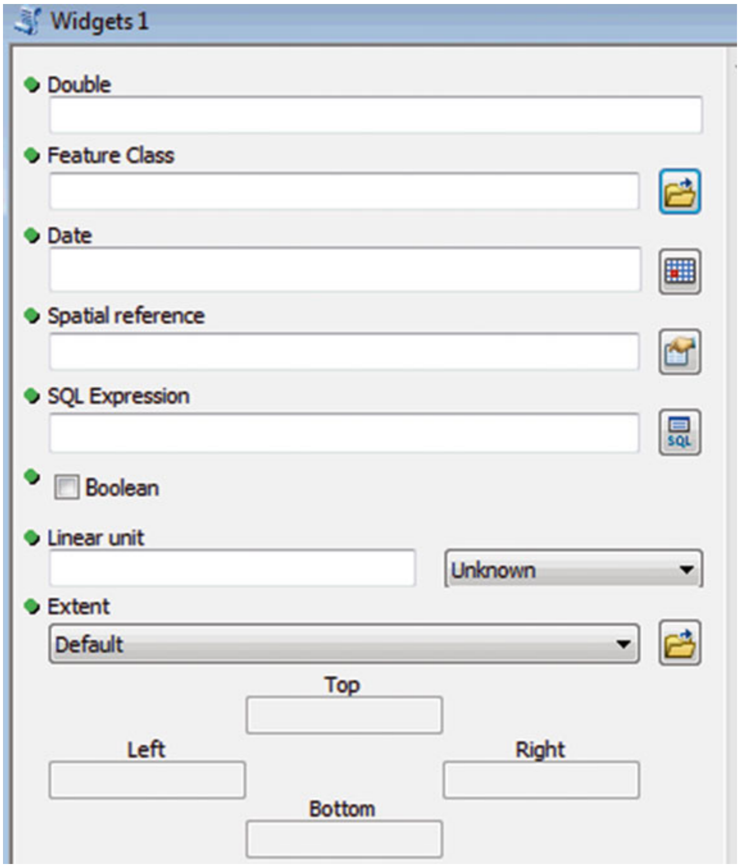


**Figure 23.8** The parameter in the 'widgets1' Script Tool GUI are labeled to match their parameter type to demonstrate how the widgets appear.

Script Tool widgets generated by Esri data types are the same widgets used in standard ArcGIS tools to collect these types of data. GUI Figure 23.8 shows a Script Tool GUI example, 'widgets1' (found in 'C:\gispy\sample_scripts\ch23\toolboxes\widgetExamples.tbx'). To demonstrate the widgets for various data types, the display name and data type are set to the same value for each parameter in 'widgets1'. The appearance of the widget created for each parameter depends partially on the parameter data type (and partially on the parameter properties, which we'll get to in a moment). Simple data types like Double, Float, or String generate a box where text or numbers can be entered. Input file (or path) data types, such as Feature Class, Raster Dataset, and Workspace, generate a text box with a file browsing button on the right end. Other types, such as 'Date', 'Spatial Reference', and 'SQL Expression', generate a text box with a distinct button that launches a specialized interface, such as a calendar, a spatial reference browser, or a SQL expression generator. Still others, such as ‘Boolean’, ‘Linear unit’, and ‘Extent’, tailor GUIs to guide user interaction. The ‘Boolean’ check box constrains the user to only two choices (checked or unchecked). The ‘Linear unit’ generates a text box on the left, plus a drop-down menu, also know as a ‘combo box’, on the right. The text box allows the user to specify a numerical value and the combo box constrains the unit to valid linear distance measures. The ‘Extent’ GUI allows the user to select the ‘Default’ value or to specify four values in the boxes below the combo box. The ‘widgets1’ Script Tool pictured in Figure 23.8 is pointing to the ‘reportSTargs.py’ from Example 23.4. The ‘reportSTargs.py’ script prints the parameter values received from a Script Tool. When ‘widgets1’ is run with a set of (arbitrarily chosen) input values, the output in the Geoprocessing Window looks like this:

```
Number of arguments = 9
Argument 0: C:\gispy\sample_scripts\ch23\scripts\reportSTargs.py
Argument 1: 0.55
Argument 2: C:\gispy\data\ch23\smallDir\trails.shp
Argument 3: 3/12/2014 3:14:41 PM
Argument 4: GEOGCS['GCS_Chatham_Islands_1979',DATUM['D_Chatham_
   Islands_1979',SPHEROID['International_1924',6378388.0,297.0]],PRIMEM
   ['Greenwich',0.0],UNIT['Degree',0.0174532925199433]];-400 -400 1000000000;
   -100000 10000;-100000 10000;8.98279933943848E-09;0.001;0.001;IsHigh
   Precision
Argument 5: value = 5
Argument 6: true
Argument 7: 3 Kilometers
Argument 8: 0 0 10 10
```

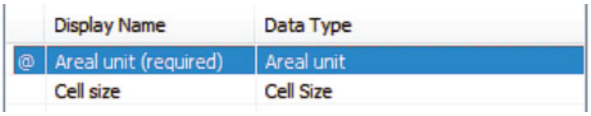
In the early stages of constructing GUIs, importing `reportSTargs` and calling `printArgs` may be helpful for trouble shooting and understanding the format of incoming values. To do so, place the `reportSTargs` module in the same directory as the script and add the following code to the script:

```
import reportSTargs
reportSTargs.printArgs()
```

The ‘widgets1’ Script Tool in Figure 23.8 shows the widgets for a few of the data types discussed here. The Script Tool named ‘widgets2’, also in ‘widgetExamples.tbx’, points to ‘reportSTargs.py’ and has a longer parameter list with matching names and types. This Script Tool can be used to preview the widgets that various data types generate. Of course, you can append additional parameters in this tool or build your own Script Tools to see how the widgets work. A Script Tool doesn’t even need to be pointed to a Python script to merely experiment with parameter widget appearance.

### 23.2.2 *Using Parameter Properties*

Parameter properties provide additional tailoring for the input widgets. The box at the bottom of the parameters input pane (see Figure 23.5) controls these properties. Take a look at the parameter list for '01_optionalParam' in the 'propertyExamples' toolbox (Right-click on '01_optionalParam' in ArcCatalog, select 'Properties…' and then select the parameters tab). Double click on one of the parameter names in the top box. An @ symbol appears to the left of its name to show that it is selected:



| | Display Name | Data Type |
|---|---|---|
| @ | Areal unit (required) | Areal unit |
| | Cell size | Cell Size |

When you select a parameter in the top box, the parameter properties box updates to display the properties for that particular parameter. You may not have noticed this, since many data types start out with the same property values. A few data types do have unique default values auto-populated. To see this, click on 'Cell Size' and then 'Compression' in the '01_optionalParam' parameter list and watch the parameter 'Default' value change from 'MAXOF' to 'LZ77' (these are names of the default algorithms for these data types). Table 23.2 lists the parameter properties with descriptions. This section discusses the parameter properties examples in the 'propertyExamples.tbx' toolbox.

#### 23.2.2.1 Type

The 'Type' property designates a parameter as 'Required', 'Optional', or 'Derived'. By default, parameters are 'Required'. A dot appears next to required parameters on the GUI until they are filled in. The tool will not run unless required parameters are specified. Optional parameters can be left blank. Script Tool '01_optionalParam' takes one required argument (an areal unit) and four optional arguments ('Cell

**Table 23.2**

| Property | Description |
|---|---|
| Type | Required, optional, or derived |
| Direction | Input or output |
| Multivalue | Accept a list of values or just one value |
| Default or schema | A schema for feature set or record set data types; a default value for all other data types |
| Environment | Set the default value based on an environment setting |
| Filter | Restrict the input values |
| Obtained from | Set a values on information from another parameter |
| Symbology | Set symbology to display output |

Size', 'Compression', 'Double', 'Feature Class' types). It points to 'reportSTargs.py'. When the tool is run with the areal unit set to '8 SquareKilometers' without specifying any of the optional arguments, it reports the following:

```
Argument 0: C:\gispy\sample_scripts\ch23\scripts\reportSTargs.py
Argument 1: 8 SquareKilometers
Argument 2: MAXOF
Argument 3: LZ77
Argument 4: #
Argument 5: #
```

When a tool is run with an optional parameter left blank, the default value for that data type is used. Some data types have a special default value (e.g., MAXOF and LZ77 for cell size and compression data types). Other data types, such as Double, Feature class, and String, have a generic default value, a hash sign (#). The user doesn't see the hash sign, but this is what the underlying script receives. The Python script needs to check for the hash sign when handling these types of optional parameters. Script Tool '02_optionalParam' takes one required argument (a base) and one optional argument (a power). This tool points to 'exponentiator.py' which calculates the base number raised to a power (e.g., if the input is 5 and 2, the tool prints '5.0 raised to the 2.0 is 25.0'). If the optional argument is omitted, the script uses a power of 1. To handle the optional argument, the underlying script checks for a hash sign:

```
if sys.argv[2] == '#':
    power = 1
    reportSTargs.printArc('No exponent provided. Using \
                           default power of 1.')
else:
    power = float(sys.argv[2])
```

With this approach, the user must provide hash signs for the 'optional arguments' when running the script outside of the Script Tool, else the script will raise an `IndexError` exception. Required arguments should be placed at the beginning of the parameter list (with optional parameters at the end).

An example of the third parameter type, 'Derived', will be discussed in the next section, as this is closely related to the 'Direction' property.

#### 23.2.2.2 Direction

The 'Direction' property designates a parameter as either 'Input' or 'Output'. By default, parameters are 'Input'. Input values are information the script needs to perform its tasks, such as a dataset or workspace to use. The Script Tool examples so far in this chapter have only used input parameters. Input parameters can be required or optional, but not derived.

The output direction is used for Script Tool parameters that represent output generated by the tool. This may be one or more datasets created by the tool. It may be a modified preexisting dataset (e.g., a dataset with a new field added, as shown in an upcoming example, '14_derivedObtainedFrom'). It may be a Boolean. It may be numerical values resulting from Script Tool calculations. Any types of results generated by standard ArcToolbox tools could be output from a custom Script Tool. Like standard tools, custom Script Tools can be used as tools in ModelBuilder models. The output (Boolean, numerical results, dataset, and so forth) would then be passed along to the output ovals in the models.

Output parameters can be required, optional, or derived. Output parameters with a 'Required' or 'Optional' type allow the user to set the name of new output datasets that will be created by the Script Tool. The '03_requiredOutput' Script Tool has two parameters, one required input feature class and one required output feature class. It points to a script named 'copier.py' that makes a copy of the first argument and names it as specified by the second argument, as in the following code:

```
arcpy.Copy_management(sys.argv[1], sys.argv[2])
```

When a tool with output data parameters is run in ArcMap, the geoprocessing output can be automatically added to the table of contents. Go to Geoprocessing menu>Geoprocessing options and check 'Add results of geoprocessing output to the display'. Run the '03_requiredOutput' on any input feature class. When the tool run is completed, the output copy of the input feature class should be automatically displayed on the map.

The combination of 'Required' type with 'output' direction can only be used for output that will be created by the script, not for modifications to existing data (if you try to select an existing dataset, the GUI will raise an error or warning). Also, it should only be used when you want to allow the user to select the location and name of the output. If the output is a modification of existing data or if the script itself determines the output name and location, output should be a 'Derived' type parameter. Input can't be derived, when you select 'Derived' in the 'Type' property, the 'Direction' property is automatically set to 'Output'. To display derived output results, you need to use an `arcpy` method named `SetParameterAsText`. Script Tool '04_derivedOutput1' has only one parameter, a derived output. This tool points to script 'buffer1.py' (Example 23.4) which buffers a hard-coded shapefile and passes the output buffer file name back to the Script Tool using the following statement:

```
arcpy.SetParameterAsText(0, outputFile)
```

The `SetParameterAsText` method takes two arguments, a number and a string. The number specifies the Script Tool parameter index. The second argument is a string representing the name of the output. In this case, it's the output file name. The `SetParameterAsText` method does not count the script path name, rather it only counts the parameters in the Script Tool list (using zero-based indexing). In this example, the Script Tool has only one parameter, so the index for this derived output parameter is zero.

**Example 23.4**

```
# buffer1.py
# Purpose:    Buffer a hard-coded file and send the result
#              to a Script Tool.

import arcpy
arcpy.env.overwriteOutput = True

fileToBuffer = 'C:/gispy/data/ch23/smallDir/randpts.shp'
distance = '500 meters'
outputFile = 'C:/gispy/scratch/randptsBuffer.shp'

arcpy.Buffer_analysis(fileToBuffer, outputFile, distance)

arcpy.SetParameterAsText(0, outputFile)
```

When the Script Tool has multiple parameters, care must be taken to index the output correctly. Script Tool '05_derivedOutput2' points to 'buffer2.py' (Example 23.5) and demonstrates how this works. Example 23.5, like Example 23.4, buffers a file, but in this case, the file to be buffered and the buffer distance are garnered from user input. The Script Tool, '05_derivedOutput1', points to this script and lists three parameters, a file to buffer, a buffer distance, and an output derived feature class, the buffer output. Example 23.5 calls `SetParameterAsText` with an index of 2 because the output file is the third one in the Script Tool parameter list. The first entry in the `sys.argv` list is the script name, so indexing for `sys.argv` user arguments starts with 1; Whereas, indexing for the `SetParameterAsText` method starts with zero. As long as the derived output parameters are listed last, you can simply start with the same index number as the last `sys.argv` index.

**Example 23.5**

```
# buffer2.py
# Purpose:    Buffer an input file by an input distance
#             and send the result to a Script Tool.

import arcpy, os, sys
arcpy.env.overwriteOutput = True

fileToBuffer = sys.argv[1]
distance = sys.argv[2]
arcpy.env.workspace = os.path.dirname(fileToBuffer)
outputFile ='C:/gispy/scratch/Buff'

arcpy.Buffer_analysis(fileToBuffer, outputFile, distance)

arcpy.SetParameterAsText(2, outputFile)
```

#### 23.2.2.3 Multivalue

The multivalue property can be 'Yes' or 'No'; the default is 'No'. When this property is set to 'Yes', the parameter will accept a list of values. Not all data types can have multiple values (e.g., Boolean parameters can not handle multiple values). A multivalue input parameter allows the user to select multiple input files and a multivalue derived output parameter allows a Script Tool to output multiple files and add them to a map automatically. When multiple values are passed into one parameter, the script receives these values as a semi-colon delimited string. Script Tool '06_ multValueIn' has a multivalue raster input parameter. Run this tool and select several rasters. The output will look something like this:

```
Input string:
C:\gispy\data\ch23\rastTester.gdb\aspect;C:\gispy\data\ch23\
rastTester.gdb\elev;C:\gispy\data\ch23\rastTester.gdb\landcov
Input file: C:\gispy\data\ch23\rastTester.gdb\aspect
Input file: C:\gispy\data\ch23\rastTester.gdb\elev
Input file: C:\gispy\data\ch23\rastTester.gdb\landcov
```

To consume the individual input rasters, the script, 'multiIn.py' (Example 23.6), splits the multivalue input string on the semicolon. This returns a list of items that can be processed within a loop.

**Example 23.6**

```
# multiIn.py
# Purpose: Parse a semicolon delimited input string.
# Usage: semicolon_delimited_string
import reportSTargs, sys

inputString = sys.argv[1]

reportSTargs.printArc('Input string: {0}'.format(inputString))

inputList = inputString.split(';')

for i in inputList:
    reportSTargs.printArc ('Input file: {0}'.format(i))
```

For multivalue derived output, the set of output names needs to be compiled into a semi-colon separated string to be passed back to the script. This can then be accomplished using the string `join` method. Script Tool '07_ multiValOut' has three parameters, an input folder, an input linear unit, and a derived multivalue output with a shapefile data type. It points to the 'bufferAll.py' script shown in Example 23.7. The script buffers every shapefile in the input folder, creating one output for each input shapefile. The names of the output files that are successfully created are collected in a list. Append the list only inside the `try` block to avoid adding files that failed to be created. After all files are created, the list is joined with a semicolon and the Script Tool output is set to the resulting string with the `SetParameterAsText` method.

**Example 23.7**

```
# bufferAll.py
# Purpose: Buffer all the feature classes in an input folder by
#           the input distance and send the output file names to
#           the Script Tool.
# Usage: working_directory linear_unit
# Sample input: C:/gispy/data/ch23/smallDir "0.2 miles"

import arcpy, reportSTargs, sys

arcpy.env.overwriteOutput = True
arcpy.env.workspace = sys.argv[1]
distance = sys.argv[2]

fcs = arcpy.ListFeatureClasses()
outList =[]
for fc in fcs:
    reportSTargs.printArc('Processing: {0}'.format(fc))
    outputFile = fc[:-4] + 'Out.shp'
    try:
        arcpy.Buffer_analysis(fc, outputFile, distance)
        reportSTargs.printArc('Created {0}'.format(outputFile))
        outList.append(outputFile)
    except arcpy.ExecuteError:
        reportSTargs.printArc(arcpy.GetMessages())

results = ";".join(outList)
reportSTargs.printArc(results)

arcpy.SetParameterAsText(2, results)
```

Both the input and output from a script can have multivalue set to true; Just use the string `split` and `join` methods as needed. A script can also have a combinations of non-multivalue and multi-value input or output. The script just needs to treat the multi-value input or output as one entity when receiving or returning it.

#### 23.2.2.4 Default or Schema

For ‘Feature Set’ or ‘Record Set’ data types, this property supplies a training set, a ‘Schema’. For all other data types this provides a default value for the parameter. The default value is intuitive, so we’ll first look at just one example with some tips on how to use the default. Sample script ‘08_defaultValues’ is identical to sample script '01_optionalParam', except the 'Default' property has been used to specify a value for each parameter. If you run the tool without modifying any of the arguments, you'll get the following arguments:

```
Argument 0: C:\gispy\sample_scripts\ch23\scripts\reportSTargs.py
Argument 1: 5 SquareKilometers
Argument 2: MINOF
Argument 3: 'JPEG' 75
Argument 4: 1.5
Argument 5: C:\gispy\data\ch23\smallDir\trails.shp
```

The defaults must be specified in a way that the tool recognizes. This is simple for well known numerical and string data types, but for many attributes, you may need to run the tool to see the choices for default values and how they are specified. For example, try to modify the default for the first parameter, the areal unit, to 200 $m^2$. To find out how to specify this, run the tool with this value selected. The output prints 'Argument 1: 200 SquareMeters', that is, 200 followed by a space, followed by Square and Meters (which are not space separated). Similarly, you can also run the tool and select various values for cell size or compression to see the sets of valid values.

When default data values are specified, relative paths should be used whenever possible. Paths should be specified relative to the position of the toolbox. The default for the last parameter in '08_defaultValues' was set using a relative path. The toolbox examples for this chapter are in 'C:\gispy\sample_scripts\ch23\toolboxes'. The relative path to the 'trails.shp' dataset is up three directories and down two; Hence, the relative path is specified as '..\..\..\data\ch23\smallDir\trails.shp'. When this default value was set, it looked like this:

```
    Default          ..\..\..\data\ch23\smallDir\trails.shp
```

But when the tool is opened again it shows the derived full path file name ('C:\gispy\data\ch23\smallDir\trails.shp'). This means that when you open the tool, it will not appear to be a relative path. To determine if a relative path has been used, move the toolbox to another directory, and the default path will be changed if a relative path was used.

When a parameter's data type is set to 'Feature Set' (or 'Record Set'), the fourth property listed in the property box says 'Schema' instead of 'Default'. For these parameter data types, this property supplies a template or training set (a 'Schema'). A feature set Script Tool parameter can be used to digitize new features or to select existing features. The feature can then be used as input for a Script Tool. The feature type (polygon, polyline, etc.) depends on the 'Schema'. The 'Schema' can be a template that you create with symbology tailor-made for a specific application or you can use a more generic feature class or feature layer.

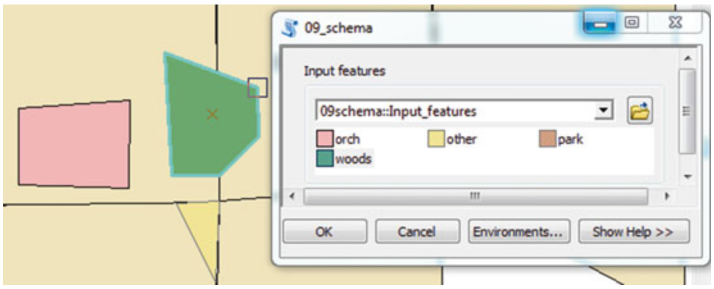


**Figure 23.9** With the '09_shema' toolbox, the user can select a land cover type for digitizing polygons of those types.

Script Tool '09_schema' lists two parameters, a 'Required', 'Input' Feature Set and a 'Derived', 'Output' Shapefile. The first parameter has the 'Schema' set to a polygon layer file, 'C:\gispy\data\ch23\training\COVER4.lyr' which uses color categories for land cover types. When the tool is launched in ArcMap, the feature set parameter shows the symbology options from this layer file (Depending on the computer's speed, it may take a moment for the symbology display to populate). The user can select a land cover type and digitize a polygon with this type. In Figure 23.9 three polygons were digitized, one with 'orch' cover, one with 'other' cover and one with 'woods' cover. The Script Tool points to 'getFeature.py' (Example 23.8) which copies the features to a shapefile and sets the derived output to the result. The output is added to the map.

**Example 23.8**

```
# getFeatures.py
# Purpose: Copy the digitized feature set input into a shapefile
#          and send this to the Script Tool as output.

import arcpy, sys
arcpy.env.overwriteOutput = True
fs = sys.argv[1]
outputFeat = 'C:/gispy/scratch/getFeaturesOutput.shp'
arcpy.CopyFeatures_management(fs, outputFeat)
arcpy.SetParameterAsText(1, outputFeat)
```

To use the tool to select existing features, click on the pull-down menu and select a layer map, then select one feature or hold down the shift key to select more than one feature from the selected map layer. You can try this by running the tool in 'featureSetExample.mxd' and selecting 'workzones' features in this way.

**Table 23.3** Parameter filters and the data types that provide them.

| Filter type | Description | Data types |
|---|---|---|
| Value list | A list of string or numeric values | String, long, double, boolean |
| Range | Numeric values between a min. and max | Long, double |
| Feature class | A list of feature class types | Feature class |
| Field | A list of field types | Field |
| File | A list of file extensions | File |
| Workspace | A list of workspace types | Workspace |

#### 23.2.2.5 Environment

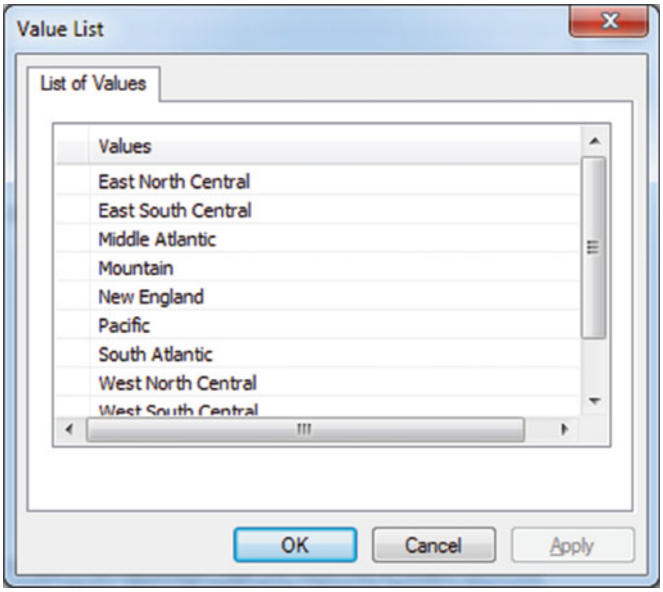
This property can be used to set the default value of a parameter to an environment setting. For example, you can use the current workspace, the scratch workspace, the output coordinate system, and so forth, of the map document to set the default value for a parameter. To use this, leave the default value blank and instead select an environment setting from the drop-down list in the 'environment' property. The '10_environ' Script Tool sets the environment property of its only parameter to the scratch workspace. If this tool is run from ArcCatalog inside of ArcMap within the 'featureSetExample' map document, the default value for the 'Workspace' parameter is set to 'C:\gispy\data\ch23', because this is the value of the scratch workspace environment setting for this map.

#### 23.2.2.6 Filter

The filter property restricts the values that can be entered. There are six types of filters and the type of filter that can be used depends on the parameter's data type as listed in Table 23.3. Other data types only give 'None' as a choice for the 'Filter' property. To use this property, click on the parameter name, then click in the filter box in the parameter property list. When you select the filter type, a GUI is launched to guide you in specifying the constraints. Script Tools '11_filterValueList' and '12_filterRangeList' use value list and range filters.

Script Tool '11_filterValueList' takes a string argument, a United States region. To help the user select a valid table, the regions are given in a value list. Double-click on the parameter and then on its filter property to see the 'List of Values' (Figure 23.10). These values are simply typed into the list; Add another item to the list and you'll see it when you run the tool. The value list creates a combo box (Figure 23.11). This tool points to 'regional.py' (Example 23.9) which prints the states in the input region. If 'New England' is selected, the output is:

```
--States in New England--
Maine
Vermont
```



Values
East North Central
East South Central
Middle Atlantic
Mountain
New England
Pacific
South Atlantic
West North Central
West South Central

**Figure 23.10** A list of values in the value list filter.

```
New Hampshire
Massachusetts
Connecticut
Rhode Island
```

## Example 23.9

```
# regional.py
# Purpose: Print the names of states in the input region.
import arcpy,reportSTargs, sys
region = sys.argv[1]
inf = 'C:/gispy/data/ch23/USA/USA_States_Generalized.shp'
fields = ['SUB_REGION', 'STATE_NAME']
sc = arcpy.da.SearchCursor(inf, fields)
reportSTargs.printArc('\n--States in {0}--'.format(region))
for row in sc:
    if row[0] == region:
        reportSTargs.printArc(row[1])

reportSTargs.printArc('\n')
del sc
```

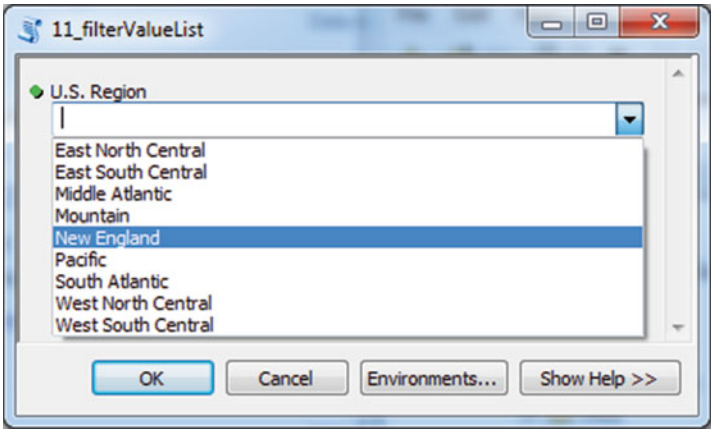


**Figure 23.11** A combo-box menu created by the value list in Figure 23.4.

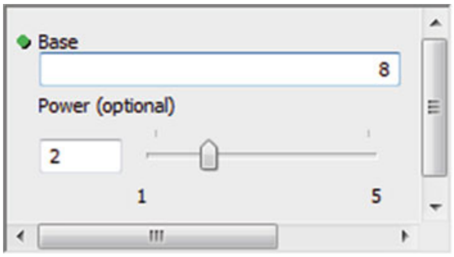
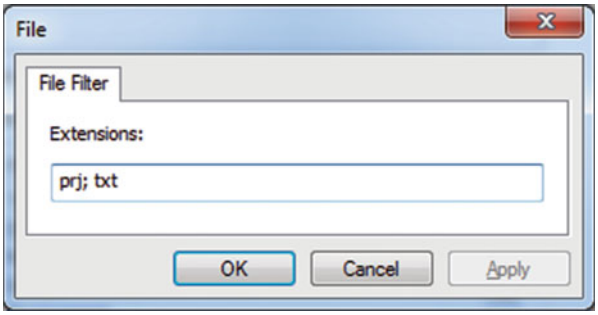


**Figure 23.12** The second parameter, ‘Power’, is a long filtered to have a range of 1 to 5.

‘12_filterRangeList’ points to the ‘exponentiator.py’ script used in a previous example, but in this case, the exponent is restricted to the integers from 1 through 5 by using the range filter with the minimum set to 1 and the maximum set to 5. If the range filter is used for a long (integer) data type, the widget is a slider bar like the one for the parameter labeled ‘Power’ in Figure 23.12. For Double data types, it displays a text box, but no slider.

The remaining filters are only available for the data types by the same name. In other words, ‘Feature Class’ filters can only be used on ‘Feature Class’ type parameters, ‘Field’ filters can only be used on ‘Field’ type parameters, and so forth. The ‘Feature Class’ filter can be used to restrict a feature class input to point, multipoint, polygon, polyline, annotation, or dimension feature classes. ‘15_symbology’ described in an upcoming section uses the feature class filter to limit the input feature class to polygon type files, so that the script can perform a feature to point conversion on the user input. Likewise, the ‘Field’ type filter can be used to restrict a field input based on a field data type (Short, Long, Float, Double, Text, and so



Extensions:

prj; txt

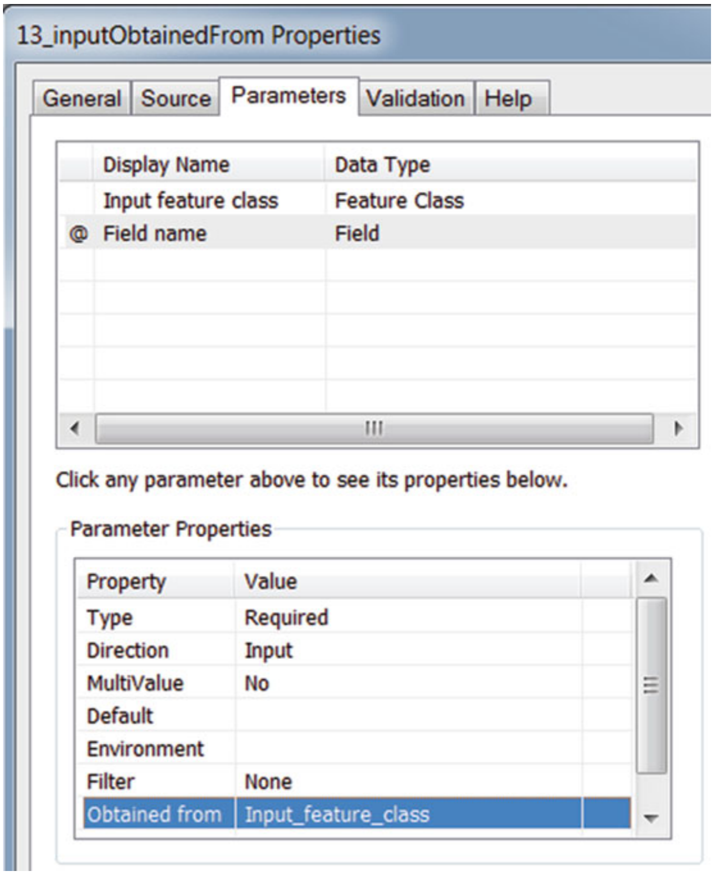
**Figure 23.13** Setting a parameter file filter to only accept projection (prj) and text (txt) files.

forth). The 'File' filter serves the same purpose as the 'filetypes' option in the `tkFileDialog` module; For 'File' type parameters, it filters the type of files the user can view and select in the file browser. When you select the 'File' filter, a dialog (like the one in Figure 23.13) opens to prompt you for extensions. Enter semicolon separated extensions. Similarly, for 'Workspace' type parameters, the 'Workspace' filters the selection to file systems, local databases, or remote databases.

#### 23.2.2.7 Obtained from

The essential idea of the 'Obtained from' property is to use information from one parameter to generate or constrain the value of another parameter. It can be used for input or derived type parameters. In Script Tool '13_inputObtainedFrom', the 'Field' values in the second parameter are obtained from the input feature class (Figure 23.14). When you run the tool the 'Field name' combo box choices change based on the 'Input feature class' selection (Figure 23.15). This property can only be set for certain parameter data types, 'Field', 'SQL expression', 'Linear Units', 'Coordinate System', and a few others.

You can also use this 'Obtained from' property to handle output data that is derived from modifying an input file. For example, input data might be modified by an update cursor or a field may be added, as mentioned in the discussion on the parameter 'Type' property. Script Tool '14_derivedObtainedFrom' points to 'combineFields.py' (Example 23.10) which adds a new field to the input dataset by combining two fields. The tool lists five parameters: an input file, two fields to combine, and a new field name (Figure 23.16). The fifth parameter is a derived output parameter obtained from the input file; It corresponds to the updated version of the input dataset. If this tool is run in ArcMap, the updated dataset is added to the map (or if that dataset is already a map layer, the layer is updated).



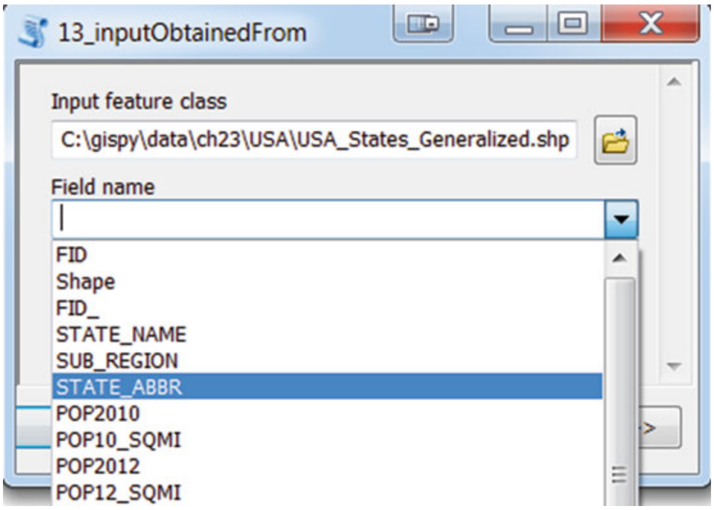
| | Display Name | Data Type |
|---|---|---|
| | Input feature class | Feature Class |
| @ | Field name | Field |

Click any parameter above to see its properties below.

Parameter Properties

| Property | Value |
|---|---|
| Type | Required |
| Direction | Input |
| MultiValue | No |
| Default | |
| Environment | |
| Filter | None |
| Obtained from | Input_feature_class |

**Figure 23.14** Using an input feature class to automatically populate a list of field names.

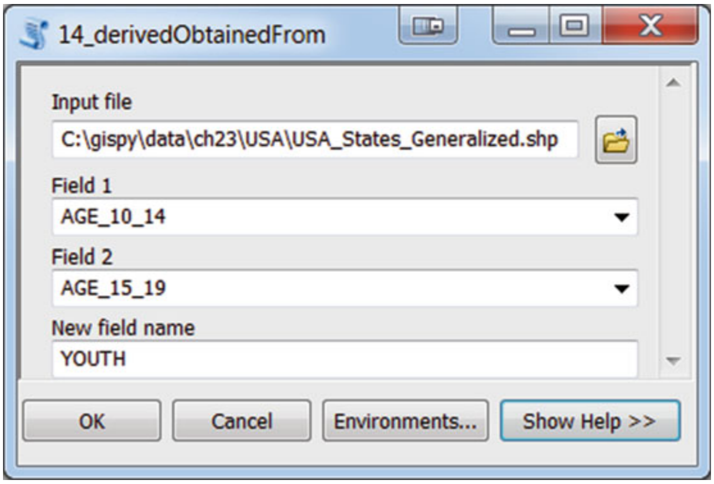


Input feature class
C:\gispy\data\ch23\USA\USA_States_Generalized.shp

Field name

FID
Shape
FID_
STATE_NAME
SUB_REGION
STATE_ABBR
POP2010
POP10_SQMI
POP2012
POP12_SQMI

**Figure 23.15** The ‘Field name’ values dynamically update depending on the selected ‘Input feature class’.



Input file
C:\gispy\data\ch23\USA\USA_States_Generalized.shp
Field 1
AGE_10_14
Field 2
AGE_15_19
New field name
YOUTH

**Figure 23.16** Four input parameters appear in the GUI, but the fifth one does not, because it is a derived output parameter.

### Example 23.10

```
# combineFields.py
# Purpose: Create a new field that is the sum of two existing fields.
import arcpy, sys
dataset = sys.argv[1]
field1 = sys.argv[2]
field2 = sys.argv[3]
newfield = sys.argv[4]

arcpy.AddField_management(dataset, newfield)
expression = '!{0}!+!{1}!'.format(field1, field2)
arcpy.CalculateField_management(dataset, newfield,
                                expression, 'PYTHON')

arcpy.SetParameterAsText(4,dataset)
```

#### 23.2.2.8 Symbology

The symbology property can be used to set the visual representation of output parameters. To do so, you need to use ArcMap to predefine the desired symbology and export the layer as a layer file (with an '.lyr' extension). Then you can set the symbology property of the output parameter to the location of the layer file. When the output is created, it will be added to the map and displayed with the same symbology as the layer file. Script Tool '15_symbology' points to a script called 'feature-2point.py' (Example 23.11) which takes an input polygon layer and calls the 'FeatureToPoint' tool to find the centroid of each layer. The derived output feature

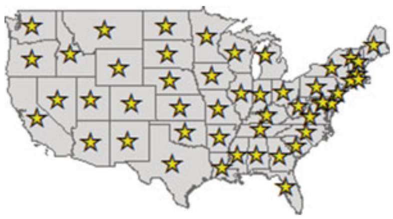


**Figure 23.17** The tool assigns star-shaped symbols to the point output based on the layer file in the symbology property.

class parameter is set to the resulting output. Its 'Symbology' property is set to a layer file with star point symbols ('C:\gispy\data\ch23\training\starPoints.lyr'). Figure 23.17 shows the star symbology output when the tool is run on the United States layer in 'symbologyPropExample.mxd'. This is one of several ways in which symbology can be applied via Python. The 'ApplySymbologyFromLayer' (Data Management) tool can also be called from Python. There are also methods within the `arcpy mapping` module, discussed in the upcoming chapter. All of these methods require a training dataset that has the desired symbology pre-configured.

**Example 23.11**

```
# feature2point.py
# Purpose: Find the centroids of the input polygons.

import arcpy, sys

arcpy.env.overwriteOutput = True
inputFile = sys.argv[1]
outputFile = 'C:/gispy/scratch/Points.shp'

# Find points based on the input.
arcpy.FeatureToPoint_management(inputFile, outputFile)

# Return the results to the tool.
arcpy.SetParameterAsText(1, outputFile)
```

## 23.3 Showing Progress

While a Script Tool performs lengthy processes, the user may be uncertain about the progress. To avoid this, the tool can communicate updates as progress occurs. There are several `arcpy` progress bar methods designed to provide custom feedback as the tool works. When a Script Tool is run, the geoprocessing window shows an oscillating progress bar (a trail of boxes moving to and fro) to let the user know that some progress is being made (Figure 23.18). The geoprocessing dialog title shows the name of the Script Tool (e.g., 'deleteFiles') and below that a label tells the user what the tool is doing. The default label states that the Script Tool is being executed (e.g., 'Executing deleteFiles'). With the `arcpy SetProgressor`

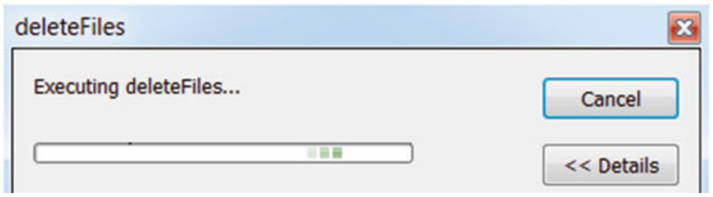


**Figure 23.18** The standard progress bar and label.

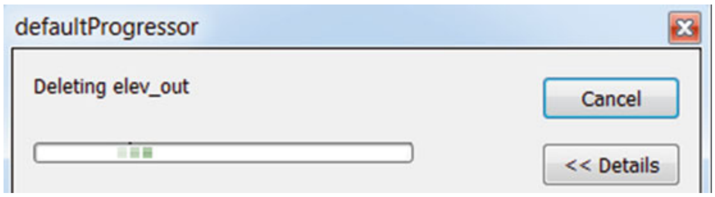


**Figure 23.19** The 'default' progress bar with a custom label.

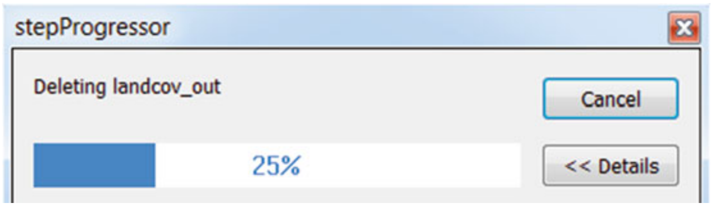


**Figure 23.20** The 'step' progress bar with a custom label.

method, you can modify the behavior of this progress indicator. There are two progress modes:

1. The 'default' mode shows the same oscillating bar, but you can change the label. The following code initializes the default progress bar with a `'Hello'` message and then updates the message to `'Deleting elev_out'`:

```
arcpy.SetProgressor( 'default', 'Hello' )
arcpy.SetProgressorLabel('Deleting elev_out')
```

    Figure 23.19 shows the geoprocessing window after the second line of code is run.

2. The 'step' mode displays a percentage bar. The following code initializes the default progress bar with a `'Hello'` message and then updates the message to 'Deleting Int_rand1':

```
arcpy.SetProgressor('step', 'Hello', 0, 4)
arcpy.SetProgressorLabel('Deleting landcov_out')
arcpy.SetProgressorPosition()
```

Figure 23.20 shows the geoprocessing window after these three lines of code are run. The third and fourth argument in the `SetProgressor` method specify a minimum and maximum value for the progressor position. An optional fifth argument can be used to specify an interval other than one for each step. The label is updated when 'SetProgressorLabel' is called. The percentage shown on the bar is updated when `SetProgressorPosition` is called. The percentage is calculated as ($s$ * interval) * 100%)/maximum (or 100%, whichever is smaller) where $s$ starts at the minimum value and gets incremented by the interval each time `SetProgressorPosition` is called. Usually when `SetProgressor` is called, the minimum is set to zero and the maximum is set to the total number of steps being tracked. In this example, there are four files being deleted, so maximum is set to four. In Figure 23.20, the bar shows 25% progress because the `SetProgressorPosition` has been called once and the maximum is four (1/4 = 25%).

The 'defaultProgressor' Script Tool in the 'progressorExamples' toolbox points to 'defaultProgressor.py' (Example 23.12), which uses the default progress bar and updates the labels. This tool is based on the 'deleteFiles' tool (discussed earlier in the chapter) that deletes files as specified by the user. A few lines of code have been added for the progress commands. The script initializes the default progressor label (`arcpy.SetProgressor('default', message)`) and then updates the label inside the deletion loop (`arcpy.SetProgressorLabel('Deleting {0}'.format(d))`). For demonstration purposes, the built-in `time` module is used to stall the script progress so that the labels persist long enough to be read. The `sleep` method suspends code execution for the given number of seconds.

**Example 23.12**

```
# Excerpt from defaultProgressor.py
ws = arcpy.env.workspace
message = """Delete '{0}' files from {1}""".format(wildcard,ws)
arcpy.SetProgressor('default', message)
time.sleep(3)
printArc(message)
for d in data:
    try:
        arcpy.SetProgressorLabel('Deleting {0}'.format(d))
        arcpy.Delete_management(d)
        printArc('{0}/{1} deleted'.format(ws, d))
        time.sleep(3)
    except arcpy.ExecuteError:
        printArc(arcpy.GetMessages())
```

If the number of steps are known at the outset, as in our 'deleteFiles' example, which gathers a list of files to delete, each accomplishment can be reported as a percentage of the overall progress. The number of files to be processed (e.g., deleted, buffered, copied, etc.) can be used as the total number of steps. The 'stepProgressor' Script Tool points to 'stepProgressor.py' (Example 23.13), which is identical to 'defaultProgressor.py', except it demonstrates using the 'step' progress mode. The script initializes the step progressor using four arguments, the mode ('step'), the label message, the minimum step value (`0`), and the maximum step value (`len(data)`). The script deletes each file in a list, so the total number of files, the length of this list, is used as the maximum step value. Inside the deletion loop, the progress label is updated (`arcpy.SetProgressorLabel('Deleting {0}'.format(d))`) and the bar position is updated (`arcpy.SetProgressorPosition()`). Again, calls to the sleep method are only for demonstration purposes and can be removed for practical applications.

**Example 23.13**

```
# Excerpt from stepProgressor.py

# Initialize the progressor.
message = """Preparing to delete '{0}' files \
           from {1}""".format(wildcard, arcpy.env.workspace)
arcpy.SetProgressor('step', message, 0, len(data))
time.sleep(3)
printArc(message)
for d in data:
    try:
        # Update progress label
        arcpy.SetProgressorLabel('Deleting {0}'.format(d))
        arcpy.Delete_management(d)
        printArc('{0}/{1} deleted'.format(arcpy.env.workspace, d))
        time.sleep(3)
    except arcpy.ExecuteError:
        printArc(arcpy.GetMessages())
    # Update progress bar percent.
    arcpy.SetProgressorPosition()
```

## 23.4 Validating Input

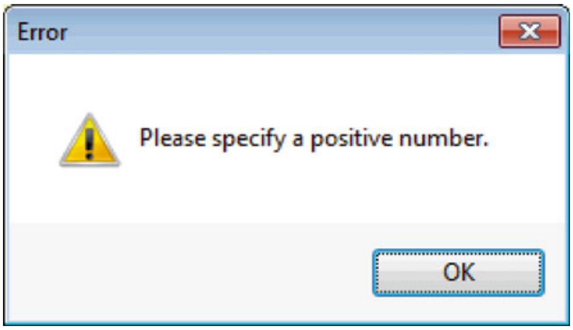
Script Tool parameters properties and tool progress methods allow you to control tool behavior in many ways. An important advantage of providing a Script Tool graphical user interface with a script is that it constrains the values users can specify; The GUI verifies and confirms that the values are valid or it posts error messages when they are not. This is referred to as *validation*. The validation process occurs while the user is interacting with the GUI. The Script Tool automatically runs internal validation based on the parameter specifications. *Internal validation* is the set of basic checks the tool performs on parameters (E.g., Are all required inputs filled in? Does the input dataset exist? Is the input a raster when it's supposed to be raster? And so forth). This protects the user from problems that can occur due to invalid input.

Some applications may require input verification that is not included in the internal validation. A 'ToolValidator' Python class embedded within the Script Tool allows you to exact even more control over the tool. Suppose, for example, you want to update a user's choices for one parameter based on another parameter in a way that the 'Obtained From' property doesn't support, or that you want to restrict the range of a non-numeric parameter, such as a date. This kind of control can be programmed by modifying the `ToolValidator` class.

The ToolValidator has to be edited using a prescribed procedure that launches an editor from within ArcDesktop. To view the `ToolValidator` class for a Script Tool, you must open the tool properties and select the 'Validation' tab. This tab displays the `ToolValidator` class. The code can not be edited in place in the 'Validation' tab; instead, you must click the 'Edit' button on this tab to launch an editor. The geoprocessing options specify which editor is used. To modify the editor, close the Script Tool properties and select the ArcCatalog (or ArcMap) 'Geoprocessing' menu>'Geoprocessing Options…'>'Script Tool Editor/Debugger'. For the 'Editor', select the full path file name of the executable for your preferred IDE (e.g., pythonwin.exe or pyscripter.exe). You may need to search Windows Explorer for the executable location.

When you edit the 'ToolValidator' script, you need to follow a particular workflow for updates to take effect. We'll use the '01_favorites_um' tool to demonstrate the steps for editing 'ToolValidator' code. This tool prompts the user for a favorite positive number. Run the tool with a positive number and then a negative number to see that it accepts either input and merely prints a message either way. 'ToolValidator' code can be added to throw an error message when input is negative. Open the '01_favorites_um' tool properties and use the following steps to edit the tool:

1. Open the tool properties (Right-click on the tool>'Properties')
2. Select the 'Validation' tab.
3. Click 'Edit…'. The script will open in your IDE.
4. Modify the script by adding three lines of code to the `updateMessages` method as shown in Example 23.14. (We'll return to the details of this code shortly.)
5. Save the script in the IDE and close it.
6. Click OK on Validation tab.
7. Run the tool to test the results. Run the tool with a negative number to see the error message the new code created.



Please specify a positive number.

**Example 23.14: ToolValidator code from Script Tool, '01_favorites_um', in validatorExamples.tbx.**

```
# Setting an error message
def updateMessages(self):
    """Modify the messages created by internal validation for
    each tool parameter.    This method is called after
    internal validation."""
    if self.params[0].altered:
        if self.params[0].value <= 0:
            self.params[0].setErrorMessage("Please specify \
                                                a positive number.")
    return
```

This example demonstrated how to access and modify the `ToolValidator` class. Next we'll discuss its methods and attributes.

### *23.4.1 The ToolValidator Class*

**Example 23.15**

```
class ToolValidator(object):
    """Class for validating a tool's parameter values and
    controlling the behavior of the tool's dialog."""

    def __init__(self):
        """Setup arcpy and the list of tool parameters."""
        self.params = arcpy.GetParameterInfo()

    def initializeParameters(self):
        """Refine the properties of a tool's parameters. This method is
        called when the tool is opened."""
        return

    def updateParameters(self):
        """Modify the values and properties of parameters before
        internal validation is performed. This method is called
        whenever a parameter has been changed."""
        return

    def updateMessages(self):
        """Modify the messages created by internal validation
        for each tool parameter. This method is called after
        internal validation."""
        return
```

The `ToolValidator` class has four methods ( `__init__` , `initializeParameters`, `updateParameters`, and `updateMessages`) and a class property named `params`. By default, three of the methods are empty except for docstrings and the `return` keyword. These methods can be modified but should not be deleted or renamed and the `return` keywords should not be deleted. The structure of the 'ToolValidator' class (shown in Example 23.15) should look familiar from the chapter on user-defined classes. The `__init__` method, is activated when an instance of the `ToolValidator` class is created. The Script Tool performs this instantiation behind the scenes when the tool is launched. The other three methods are also triggered automatically by the Script Tool, based on various events. Most code that you would want to add involves using the `self.params` list, which contains a list of `Parameter` objects. To understand the modifications that can be made to the `ToolValidator` methods, you need to be familiar with the `Parameter` object properties and methods.

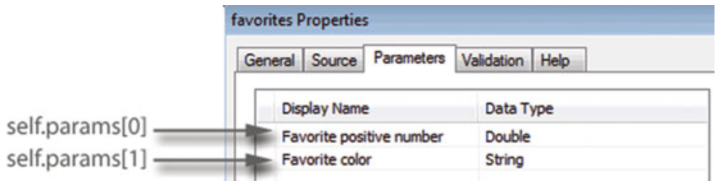
### 23.4.1.1 Initializing Parameter Objects

The modifications that you can make usually involve the class attribute named `params` which is defined in the `__init__` method. The `getParameterInfo` `arcpy` method returns a list of `Parameter` objects and this is stored in `self.params`:

```
def __init__ (self):
    self.params = arcpy.GetParameterInfo()
```

This list of `Parameter` objects corresponds to the parameters in the list on the Script Tool properties 'Parameters' tab, with zero-based indexing starting at the top of the list. For example, in the '01_favorites_um' Script Tool, self.params[0] refers to the 'Favorite positive number' parameter and self.params[1] refers to the 'Favorite color' parameter (Figure 23.21).

`Parameter` objects have some of the properties that are specified on the 'Parameters' tab, such as, `name` (Display name with underscores substituted for spaces), `direction`, `datatype`, and `symbology`. These are read-only properties. Other properties, such as, `category`, `value`, `enabled`, `altered`, and `filter.list` can be modified. Search for 'ToolValidator Parameter object' on the



| Display Name | Data Type |
|---|---|
| Favorite positive number | Double |
| Favorite color | String |

**Figure 23.21** The `self.params` Python list corresponds to the tool parameter list.

ArcGIS Resources site for documentation of the `Parameter` object properties and methods. The upcoming examples demonstrate ways to use these properties and methods.

### 23.4.1.2 An `updateMessages` Example

Let's return to Example 23.14. This code forces the user to enter a positive favorite number into the '01_favorites_um' Script Tool, using the `updateMessages` method. As the docstring states, the `updateMessages` method is called just after internal validation (the set of checks automatically performed by the tool). If an invalid input is used, internal validation detects this problem and throws an error when you update the parameter or try to run the tool. For some applications, you may want to add your own checks for conditions that can't be constrained by other means. For example, though you can use a filter to force the user to select a number between 1 and 5, you can't use the filter to specify an open ended range, but `updateMessages` can enforce this. The purpose of `updateMessages` is to enable custom checks, errors, and warning messages. Example 23.14 uses `altered` and `value` Parameter properties and the `setErrorMessage` Parameter method. The `params` variable is a `ToolValidator` class property, so when you refer to it inside of class methods, you must use `self.params`. The first line of code inside `updateMessages` in Example 23.14 checks if the first parameter in the list has been `altered` by the user:

```
if self.params[0].altered:
```

The `altered Parameter` property is Boolean (`True` or `False`). When the tool is first launched, the `altered` property for each parameter is `False`. This property is set to `True` when the tool detects a change (and back to `False` when appropriate). The next line of code compares the first parameter's `value` to zero:

```
    if self.params[0].value <= 0:
```

The Parameter property named `value` has the value passed in by the user. In the Python `sys.argv` variable, all arguments are received as strings. By contrast, the data type of the `value` property is a `Value` object and the behavior depends on the data type of the input. For parameters with an Esri data type in Table 23.1, the Python equivalent data types are used. Many data types are received as strings. However, the `value` of the user's favorite number does not need to be cast to 'float' before it is compared to zero because `self.params[0].value` is derived from a 'Double' type parameter. If the value of this number is not positive, `setErrorMessage` is called:

```
self.params[0].setErrorMessage('Please specify a positive number.')
```

The `setErrorMessage` method is one of several Parameter methods related to messaging (Other methods include `setWarningMessage`, `clearMessage`, `hasError`, and so forth). The `setErrorMessage` method opens a window containing the message and the tool can not be run until the input is changed to a positive value. Putting it all together, the code checks if the parameter has been altered, if so, it checks if its value is positive. If not, it displays a message associated with that parameter:

```
if self.params[0].altered:
    if self.params[0].value <= 0:
        self.params[0].setErrorMessage('Please specify a positive \
                                                  number.')
```

Generally, the code in `updateMessages` should check a parameter's `altered` condition before checking the parameter's `value`. If no default is set for a parameter and the script doesn't check if it has been altered, an error may appear before the user even has an opportunity to specify a value. To see this occur, remove the statement `if self.params[0].altered`: from the '01_favorites_um' ToolValidator code (dedent the code below it as needed) and then run the tool.

### 23.4.1.3 An `initializeParameters` Example

The `initializeParameters` method is called just one time. The call occurs when the tool is first launched (after the `ToolValidator` object has been created). This method is only ever called once during a single tool run, it can be used as an alternative means to set default values. Default values can either be set on the 'Parameters' tab using the 'Default or schema' property or they can be set in the `ToolValidator` class using the `Parameter` object `value` attribute. If the default is set in both ways, the value used in the `ToolValidator` code will override whatever value was set on the 'Parameters' tab.

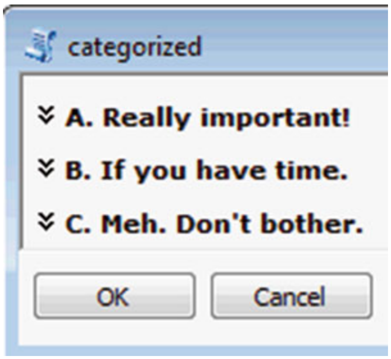


**Figure 23.22** The 'categorized' Script Tool has three parameter categories, shown here collapsed.

The initializeParameters method can also be used to set up parameter categories. Figure 23.22 shows a tool with three categories. Parameters grouped in categories can be collapsed and expanded. The user can click on the arrows next to the category name to expand or collapse a category. This can help to keep the size of the GUI manageable and organize the parameters logically. Example 23.16 sets up three categories. The code loops through the parameters and sets the category property of each Parameter object. By looping through the indices and checking the parameter index, the category for the first third of the parameters is set to 'A. Really important!'. The category for the middle third is set to 'B. If you have time.', and for the last third, it's set to 'C. Meh. Don't bother.' for the last third. Tools with numerous parameters sometimes group parameters into 'Required' and 'Optional' categories by checking their parameterType. This is posed as an exercise at the end of the chapter.

**Example 23.16: Code in the 'validatorExamples.tbx/02_categories_ip' ToolValidator.**

```
def initializeParameters(self):
    """Assign parameter categories."""
    numParams = len(self.params)
    for index in range(numParams):
        if index < numParams/3.0:
            self.params[index].category = 'A. Really important!'
        elif index < (2*numParams)/3.0:
            self.params[index].category = 'B. If you have time.'
        else:
            self.params[index].category = "C. Meh. Don't bother."
    return
```

#### 23.4.1.4 An updateParameters Example

The updateParameters method is called each time a parameter is changed. This method can be used to dynamically update one parameter based on another parameter. The '03_rasters_up' Script Tool is designed to allow the user to select a workspace and then select rasters within the workspace. The tool has two parameters, a workspace parameter and a multivalue string parameter. The string data type is used for the second parameter because the string data type allows a value list filter. The code shown in Example 23.17 uses the filter.list Parameter property to set the list of items in the filter. The code in updateParameters updates the value list of the second parameter based on the first. The code gets a list of rasters in the workspace specified by the user and sets the filter.list property to the list of raster names. The updateMessages method posts an error if the list is empty (in other words, the selected workspace contains no rasters).

**Example 23.17: ToolValidator code from Script Tool, 'validatorExamples.tbx/03_rasters_up'.**

```
def updateParameters(self):
    '''Initialize raster list.'''
    if self.params[0].altered:
        arcpy.env.workspace = self.params[0].value
        rasts = arcpy.ListRasters()
        if rasts:
            self.params[1].filter.list = rasts
        else:
            self.params[1].filter.list = []

    return

def updateMessages(self):
    '''Check for rasters.'''
    if self.params[0].altered:
        if not self.params[1].filter.list:
            self.params[0].setErrorMessage('This directory \
                                  does not contain any rasters.')
    return
```

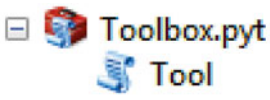
When you save the ToolValidator script and select ‘Apply’, the script is checked internally. If there is a problem with the modified code, the process may hang until you cancel the update (by clicking the ‘Cancel’ button). Some common mistakes, such as omitting `self` or `value` when needed obstruct the script from being applied. For example, the following code exhibits these two mistakes:

```
# Incorrect...
self.params[0] = 5
params[1].filter.list = [1,2,3]

# Correct...
self.params[0].value = 5
self.params[1].filter.list = [1,2,3]
```

The `ToolValidator` class can be debugged by creating a stand-alone script that imports the toolbox, specifies the tool, sets the input parameters, instantiates a `ToolValidator` object, and invokes the validation methods. An alternative to the Script Tool referred to as a ‘Python toolbox’ makes debugging easier by holding all the information within a text file.

## 23.5 Python Toolboxes

A *Python toolbox* is an ASCII text file with a '.pyt' extension that contains Python code to define a toolbox and one or more tools. Create one in a directory in ArcCatalog (10.1 or higher) with right-click>New>Python toolbox. In ArcCatalog, the Python toolbox (named 'Toolbox.pyt' by default) is displayed with a toolbox icon and expands to show the tools it contains (by default, it contains one named 'Tool'). The Python toolbox icon shows a scroll by the toolbox and the tool uses the same icon as Script Tools:



A tool in a Python toolbox can do everything that a Script Tool can do, but it streamlines the workflow of creating a GUI. With the Script Tool, each parameter must be specified by clicking and selecting the properties on the 'Parameters' tab of the Script Tool wizard. The Python code associated with a Script Tool is contained in a separate file; Whereas, the code for a Python toolbox and one or more tools can all be contained in a single '.pyt' file (It can also import user-defined modules to invoke code in other Python scripts). Script Tools are a convenient way to learn about the GUIs that can be made with ArcGIS, but Python programmers may prefer Python toolboxes, since they are entirely Python based.

Initially, the code in a Python toolbox defines two classes, `Toolbox` and `Tool`. Example 23.18 shows the default Python code generated by creating a Python toolbox. To view or edit this code, right click on the toolbox in ArcCatalog and select 'Edit…'. (Or to open it directly from an IDE, select Open>change the file type to 'All files'>browse to the '.pyt' file). The `Toolbox` class sets the toolbox label and alias. It also sets `self.tools`, to a list containing one tool class. You can add more tools by making a copy of the `Tool` class, pasting it at the bottom of the file, renaming it (e.g., `Tools2`), and then adding the new tool class to the tools list (e.g., `self.tools = [Tool, Tool2]`). Each tool class must contain six methods, `__init__`, `getParameterInfo`, `isLicenced`, `updateParameters`, `updateMessages`, and `execute`. The `__init__` method sets the tool label and description. The label can differ from the name of the class, but the tool list in the Toolbox class must use the class name, not the label. The `getParameterInfo` method initializes the parameters. The `isLicensed` method can be used to keep a tool from being run if the required geoprocessing tools are not available. The `updateParameters` and `updateMessages` method are for validation (like their Script Tool `ToolValidator` equivalents). Lastly, the code that you want to run when the tool is executed can be placed in the `execute` method.

**Example 23.18: Python toolbox template.**

```
import arcpy

class Toolbox(object):
    def __init__(self):
        '''Define the toolbox (name of the toolbox is the
        name of the '.pyt' file).'''
        self.label = 'Toolbox'
        self.alias = ''

        # List of tool classes associated with this toolbox
        self.tools = [Tool]
class Tool(object):
    def __init__(self):
        '''Define the tool (tool name is the name of the class).'''
        self.label = 'Tool'
        self.description = ''
        self.canRunInBackground = False

    def getParameterInfo(self):
        '''Define parameter definitions'''
        params = None
        return params

    def isLicensed(self):
        '''Set whether tool is licensed to execute.'''
        return True

    def updateParameters(self, parameters):
        '''Modify parameters before internal validation. Called
        whenever a parameter has been changed.'''
        return

    def updateMessages(self, parameters):
        '''Modify messages created by internal validation. Called
        after internal validation.'''
        return

    def execute(self, parameters, messages):
        '''The source code of the tool.'''
        return
```

The Python toolbox, ‘rasterToolbox.pyt’ in the ‘pyToolboxes’ directory contains a single tool class, `RastersExample`. This tool calculates the Sine of the input rasters. The first parameter gets the user workspace. When this is selected, the rasters within this workspace are listed (using code similar to Example 23.17). Then when the tool is run, it calculates the Sin of each of the rasters selected in the second parameter. We’ll use the `RastersExample` class to compare Script Tools to the tools in Python toolboxes.

**Note** To edit a Python toolbox, right-click on the toolbox, not the tool(s) it contains.

### *23.5.1 Setting Up Parameters (`getParameterInfo`)*

The biggest difference between the Script Tool and the Python toolbox is that you set up the parameters in the code inside the `Tool` class in the `getParameter-Info` method. This replaces the 'Parameters' tab on the Script Tool wizard. For each parameter, you create a `Parameter` object and set its properties. For example, the following code creates `myParam`, a `Parameter` object:

```
myParam = arcpy.Parameter()
```

Then the following code sets its `name` property:

```
myParam.name = 'My_precious'
```

The parameter name can not contain spaces or special characters. The parameter name is the only required property; However, the `displayName`, `parameter-Type`, `direction`, and `datatype` are conventionally set as well (by default, the display name is set to `None`, the parameter type is set to `'Required'`, the direction is set to `'Input'`, and the data type is set to `'String'`). There is an equivalent Python toolbox Parameter property for each one displayed in the Script Tool property list. The parameters are created and initialized inside the `getParame-terInfo` method and then this function returns a list of the `Parameter` objects. Examples 23.19 shows the `getParameterInfo` method from the 'RastersExample' tool. Both parameters are required input. The first one is a workspace, the second is a string. A workspace filter list is used to restrict the workspace selection to local databases (file or personal geodatabases). In the Script Tool, this would be done by clicking on filter>workspace>Local Database. The second parameter allows multiple values by setting the `multiValue` property to `True`. The last line of the method returns the two `Parameter` objects in a Python list as follows:

```
return [param1, param2]
```

This list is passed in as an argument in the `updateParameters`, `updateMes-sages`, and `execute` methods as discussed shortly.

**Example 23.19**

```
def getParameterInfo(self):
    '''Set up the parameters and return
    the list of Parameter objects.'''
    # 1_Select_a_workspace_containing_rasters
    param1 = arcpy.Parameter()
    param1.name = '1_Select_a_workspace_containing_rasters'
    param1.displayName = '1. Select a workspace \
                          containing rasters:'
    param1.parameterType = 'Required'
    param1.direction = 'Input'
    param1.datatype = 'Workspace'
    param1.filter.list = ["Local Database"]

    # 2_Select_rasters_within_the_workspace
    param2 = arcpy.Parameter()
    param2.name = '2_Select_rasters_within_the_workspace'
    param2.displayName = '2. Select rasters within \
                          the workspace:'
    param2.parameterType = 'Required'
    param2.direction = 'Input'
    param2.datatype = 'String'
    param2.multiValue = True
    param2.filter.list = []

    return [param1, param2]
```

## 23.5.2 *Checking for Licenses (`isLicensed`)*

The `isLicensed` method can be used to check if an extension license is available and prevent the tool from running. The 'RastersExample' tool uses the Spatial Analyst extension to calculate the Sine. This is not available in all ArcGIS desktop products. The code in Example 23.20 prevents the user from running the tool if the Spatial Analyst extension is not licensed.

**Example 23.20**

```
def isLicensed(self):
    """Prevent the tool from running if the Spatial Analyst extension
       is not available."""
    if arcpy.CheckExtension('Spatial') == 'Available':
        return True # tool can be executed
    else:
        return False # tool can not be executed
```

### *23.5.3 Validation (`updateParameters` and `updateMessages`)*

Like Script Tools, Python toolbox tools perform internal validation and the code can implement custom evaluation. The `updateParameters` and `updateMessages` method have the same functionality as their Script Tool ToolValidator equivalents. Example 23.21 shows the validation included in the `RastersExample` tool class. Comparing this to the code in Example 23.17, they are almost identical. The only difference is how the parameters are referenced in the code. In Example 23.17 the first parameter is `self.params[0]`. In Example 23.21, the first parameter is `parameters[0]`. The Script Tool `ToolValidator` class stores the list of `Parameter` objects as a class property. The `self.params` list is set in the `__init__` method as follows:

```
self.params = arcpy.GetParameterInfo()
```

The Python toolbox `Tool` class handles the parameter list differently. Instead of using a property, the `Tool` class passes the parameter list into the methods that use it. For example, it is the second item in the argument lists for both `updateParameters` and `updateMessages`:

```
def updateParameters(self, parameters):
def updateMessages(self, parameters):
```

This means, you simply index into this ‘parameters’ list without using the self-dot notation. The `Parameter` objects are the same data type as those used in the Script Tool ToolValidator class, so all of the same methods and properties apply (e.g., `value`, `filter.list`, `altered`, `setErrorMessage`, `enabled`, and so forth…) To show some additional examples comparing Script Tool validation to Python toolbox validation, Python toolbox ‘validatorExConverted.pyt’ (in the ‘pyToolboxes’ directory) is equivalent to the ‘validatorExamples.tbx’ (in the ‘toolboxes’ directory). For the most part, the validation code is the same, other than the `Parameter` object handling. Unlike the ToolValidator, the Python toolbox has no `initializeParameters` method. This functionality can usually be placed inside the `getParameterInfo` method for a tool in a Python toolbox. Example 23.16 used `initializeParameters` to categorize the parameters. The ‘Categories_ip’ tool sets these categories in its `getParameterInfo` method before returning the parameter list.

**Example 23.21**

```
def updateParameters(self, parameters):
    '''Initialize raster list.'''
    if parameters[0].altered:
        arcpy.env.workspace = parameters[0].value
        rasts = arcpy.ListRasters()
```

```
            if rasts:
                parameters[1].filter.list = rasts
            else:
                parameters[1].filter.list = []
        return

    def updateMessages(self, parameters):
        '''Check for rasters.'''
        if parameters[0].altered:
            if not parameters[1].filter.list:
                parameters[0].setErrorMessage('This directory does not \
                                               contain any rasters.')
        return
```

### *23.5.4 Running the Code (`execute`)*

A Python toolbox tool is launched by double-clicking on the tool in ArcCatalog. Then values for the parameters are specified in the GUI. Once valid values are selected, the tool can be run by selecting the 'OK' button. When the 'OK' button is selected, the tool class `execute` method is called. This means that when you create a tool for Python toolbox, the code that you want it to run should be placed in the `execute` method. Script Tools point to a stand-alone scripts; By contrast, in a Python toolbox, the main script code can be placed directly in this text file. As in the `updateParameters` and `updateMessages` methods, the list of `Parameter` objects is passed into this method as the variable named `parameters`. Example 23.22 shows the `execute` method for the 'RastersExample' tool. This code gets the list of user-selected rasters and computes the trigonometric Sine of each raster. The code checks out the Spatial Analyst extension, turns on output overwriting, and sets the workspace based on the value of the first parameter. Next, it loops through the list of raster names selected by the user. Instead of using the `value` property, this line of code uses the `values` property:

```
for rast in parameters[1].values:
```

The `values` property of a multivalue string parameter returns a list of strings. The code inside the loop computes and saves an output raster for each name in the list.

**Example 23.22**

```
def execute(self, parameters, messages):
    '''Calculate the Sine of each input raster.'''
    arcpy.CheckOutExtension('Spatial')
    arcpy.env.overwriteOutput = True
    arcpy.env.workspace = parameters[0].value # Set the workspace
```

```
        for rast in parameters[1].values:
            try:
                outSin = arcpy.sa.Sin(rast)
                outSin.save(rast + '_Sin')
                message = '{0}_Sin created in {1}.'.format(rast,
                                                arcpy.env.workspace)
                arcpy.AddMessage(message)
            except:
                message = '{0}_Sin could not be created.'.format(rast)
                arcpy.AddMessage(message)
```

The main code for the `execute` method can also be placed in a stand-alone user-defined module within one or more functions. Then the module can be imported and its functions called from within the `execute` method. For example, 'RastersToolboxVersion2.pyt' imports `rastModule` and calls `batchSine`. The following concise code is the resulting `execute` module:

```
def execute(self, parameters, messages):
    '''Calculate the Sine of each input raster.'''
    import rastModule
    wkspace = parameters[0].value
    rasters = parameters[1].values
    rastModule.batchSine(wkspace, rasters)
```

The user-defined module 'rastModule' is shown in Example 23.23. This approach of modularizing `execute` code improves organization and maintainability. If the GUI is added last, this approach simplifies integration with existing code.

**Example 23.23**

```
# rastModule.py
import arcpy
def batchSine(workspace, rastList):
    '''Calculate the Sine of each raster in the list.'''
    arcpy.CheckOutExtension('Spatial')
    arcpy.env.overwriteOutput = True
    arcpy.env.workspace = workspace # Set the workspace
    for rast in rastList:
        try:
            outSin = arcpy.sa.Sin(rast)
            outSin.save(rast+'_Sin')
            message = '{0}_Sin created in {1}.'.format(rast,
                                            arcpy.env.workspace)
            arcpy.AddMessage(message)
        except:
            message = '{0}_Sin could not be created.'.format(rast)
            arcpy.AddMessage(message)
```

### 23.5.5 Comparing Tools

To see a few additional differences between Script Tools and Python toolbox tools, compare standard toolbox, ‘propertyExamples.tbx’ (in the ‘toolboxes’ directory) with Python toolbox ‘propertyExConverted.pyt’ (in the ‘pyToolboxes’ directory). The tools in the ‘.pyt’ files are equivalent to those in the ‘tbx’ files. The Python toolbox contains one Python class for each tool. The tool class of ‘propertyExConverted.pyt’ sets a long list of class names (one for each of the 15 tools):

```
self.tools = [ToolOptionalParam1,ToolOptionalParam2,ToolRequiredOutput,
              ToolDerivedOutput1,ToolDerivedOutput2,ToolmultiValueIn,
              ToolMultiValueOut, ToolDefaultVals, ToolSchema,
              ToolEnvironments, ToolFilterRange, ToolFilterValueList,
              ToolInputObtainedFrom, ToolDerivedObtainedFrom,
              ToolSymbology]
```

Tool names and labels differ slightly in this example since Python class names can not start with a number. For example, the `ToolDerivedOutput2` class creates the ‘05_derivedOutput2’ tool. The code in the `execute` method for this tool (line 249) illustrates one of the things that you should be aware of when writing code for Python toolboxes. The value property of a parameter is a `Value` object. Functions that take strings as input, such as, `os.path.dirname` can not use the `Value` object directly. Instead, you need to use its `value` property to get a string. The following code from the `execute` method for ‘ToolDerivedOutput2’ sets the ‘fileToBuffer’ variable in this way:

```
fileToBuffer = parameters[0].value.value
arcpy.env.workspace = os.path.dirname(fileToBuffer)
```

Using `value.value` returns the text string that names the object (instead of the `Value` object). In this case, it is the full path file name of the feature class selected by the user. The `os.path.dirname` function can derive a directory name from this string; whereas, it can not derive a directory name from a `Value` object. The following code throws an error:

```
fileToBuffer = parameters[0].value
arcpy.env.workspace = os.path.dirname(fileToBuffer)
```

To work correctly, some ArcGIS tools require the text string for input, instead of the object. For example, the Calculate Field (Data Management) tool is used in the `ToolDerivedObtainedFrom` class (Example 23.24). If the text name is not used, the tool throws an error saying that the field that was just created does not exist. Not all `Value` objects have a `value` property, so it can’t be used all the time, but you can keep in mind that unexpected errors may be resolved by using the text name (as `value.value`) instead of the `Value` object.

**Example 23.24**

```
def execute(self, parameters, messages):
    # From combineFields.py
    # Purpose: Create a new field that is the sum
    #          of two existing fields.
    dataset = parameters[0].value.value
    field1 = parameters[1].value
    field2 = parameters[2].value
    newfield = parameters[3].value
    arcpy.AddField_management(dataset, newfield)
    expression = '!{0}!+!{1}!'.format(field1, field2)
    arcpy.CalculateField_management(dataset, newfield,
                                    expression, 'PYTHON')

    arcpy.SetParameterAsText(4,dataset)
```

The `ToolDerivedObtainedFrom` class demonstrates another difference between Script Tools and Python toolbox tools. The Python toolbox equivalent of the ‘Obtained from’ property is `parameterDependencies`. Example 23.25 shows an excerpt from the `getParameterInfo` method. The first parameter is an input ‘Feature Class’ and the second one is a ‘Field’. The tool allows the user to select an input feature class and then a field from that dataset. The fields in the second parameter are obtained from the first parameter by setting the `parameter-Dependencies`. This property takes a Python list of `Parameter` object names. The last line of code in Example 23.25 sets the dependencies of the second parameter to a list containing the first `Parameter` object name (`[param1.name]`). When the user selects the input dataset, the second parameter populates a combo box with the dataset’s field names. The fifth parameters in this tool, the derived output dataset, is set up the same way:

```
param5.parameterDependencies = [param1.name]
```

**Example 23.25: The `parametersDependencies` `Parameter` property is equivalent to the 'Obtained from' Script Tool property.**

```
def getParameterInfo(self):
    '''Create parameters and set their properties'''
    # Input_file
    param1 = arcpy.Parameter()
    param1.name = 'Input_file'
    param1.displayName = 'Input file'
    param1.parameterType = 'Required'
    param1.direction = 'Input'
    param1.datatype = 'Feature Class'

    # Field1
    param2 = arcpy.Parameter()
    param2.name = 'Field1'
    param2.displayName = 'Field 1'
    param2.parameterType = 'Required'
    param2.direction = 'Input'
    param2.datatype = 'Field'
    param2.parameterDependencies = [param1.name]
```

In Python toolboxes, filter lists are still available and they are accessed with the same notation as in the Script Tool `ToolValidator`, with the `filter.list` `Parameter` property. The 'ToolFilterValueList' tool hard codes the region names with a Python list of strings as in the following code:

```
param1.filter.list = ['East North Central', 'East South Central',
'Middle Atlantic', 'Mountain',
'New England', 'Pacific', 'South Atlantic',
'West North Central', 'West South Central']
```

The range filter for a 'Long' type parameter can be set by specifying the filter type as `'Range'` and setting the filter list as a Python list of two values. For example, the 'ToolFilterRange' tool sets the values of the second parameter to range between one and five with the following code:

```
param2.filter.type = 'Range'
param2.filter.list = [1,5]
```

Python toolboxes can still use relative paths for default datasets, workspaces, schema files, and so forth, but to use a relative path, the code must calculate the absolute path dynamically. For example, to set a default shapefile within the 'ToolDefaultVals' tool, the code uses a custom function named `getAbsPath`. As shown in Example 23.26, `getAbsPath` derives an absolute path from a relative path by using the path of the Python toolbox. Custom functions can be added to a '.pyt' file, just as they can to any other Python script. To make sure they are defined before they are used, functions should be placed just after the import statements, before the class definitions. The following code, from the `ToolDefaultVals` class `getParameterInfo` method, sets a default value for an input dataset using a relative path and dynamically deriving its absolute path by calling the `getAbsPath` function:

```
relativePath = '../../../data/ch23/smallDir/trails.shp'
param5.value = getAbsPath(relativePath)
```

**Example 23.26**

```
def getAbsPath(relativePath):
    '''Return the absolute path given a relative path to this file'''
    tbxPath = os.path.abspath(file
    tbxDir = os.path.dirname(tbxPath)
    fullPath = os.path.join(tbxDir, relativePath)
    return os.path.abspath(fullPath)
```

## 23.6 Discussion

Script Tools and Python toolboxes provide a convenient means for creating graphical user interfaces for ArcGIS scripts. Though they have some limitations, they are well-suited for making GUIs to collect ArcGIS data types. Script Tools are a good starting point for learning to use parameter data types and their properties. Python toolboxes facilitate a more streamlined workflow for creating tools by using Python code, instead of a tool wizard, to specify parameter properties. Setting up validation is also more efficient in a Python toolbox tool. When you modify the code, you can save it, refresh the toolbox, and see the results. You can leave the saved '.pyt' file open while testing the tool and you can simply edit it again as needed. Each time you edit Script Tool validation code, you need to open the tool properties, select the 'Edit' button on the validation tab, edit and save the code, apply the changes, close the tool properties, and then run the tool. When you make changes to the code in Python toolboxes, you do need to refresh the file in ArcCatalog to see the changed behavior.

A red X on a tool icon in ArcCatalog means the tool won't run. When you see an X on a Script Tool, it often means that the path to a script was not set as relative or the script has been moved to a different relative position. If this is the case, you can update the 'Script' path to fix the tool (and check 'Use relative pathnames, if desired). Python toolboxes help to avoid this common mistake since they define the toolbox and tools in one file. If the code in a Python toolbox contains an error, a red X will appear as well. For certain kinds of errors, a 'Why' tip will appear on the tool. You can select this tip to learn more about the error (see 'zzz.pyt' for an example).

ArcGIS hosts a downloadable tool for exporting Script Tools to Python toolboxes, but similar to exporting ModelBuilder models to Python scripts, the user needs to have a good understanding of Python to refine the resulting '.pyt' file for full functionality. The template along with sample Python toolboxes provide a good starting point for creating new Python toolboxes.

Graphical user interfaces should usually be added near the end of a GIS project, once the programming functionality has been developed and tested. GUIs are convenient for users, not for developers.

## 23.7 Key Terms

| | |
|---|---|
| Script Tool | Validation |
| Geoprocessing Window | Internal validation |
| Widget | Python toolbox |
| Combo box | `Parameter` object |
| `ToolValidator` class | `Value` object |

## 23.8 Exercises

**General instructions:** Exercises 1–11 involve creating new Script Tools or copying and modifying existing Script Tools. The toolbox, 'exercises.tbx' in the 'C:\gispy\sample_scripts\ch23\exercises' directory contains sample Script Tools used in the exercise ('exerciseCopy.tbx' is a copy of this one, in case you need to revert to the original state). Place Python scripts in the same directory as the toolboxes and set the Script Tool to use relative path names for scripts. Exercises 12 and 13 involve creating Python toolboxes. Either write all code for these inside the '.pyt' file or import files in the same directory. Display all printed output for exercises 1–13 in both the Interactive Window and the Geoprocessing Window.

1. **triArea** and **triangleArea.py** Write a script, 'triangleArea.py', to calculate the area of the triangle based on the user input of a triangle base and width. Create a Script Tool, 'triArea', that has two float parameters, the base and width of a triangle. Point the tool to the 'triangleArea.py' script.

   Example input: 5.6 9

   Example output:
   ```
   >>> The area of the triangle is 25.2.
   ```

2. **bridgeOfDeath** and **answers.py** Create a script, 'answers.py', to print the user's name, quest, and favorite color, as shown in the example. Create a Script Tool, 'bridgeOfDeath', that poses three questions: 'What is your name?', 'What is your quest?', and 'What is your favorite color?' The tool should have three string parameters to collect the answers. Point the tool to the 'answers.py' script.

   Example input: "Sir Lancelot" "to seek the Holy Grail" blue

   Example output:
   ```
   >>> Your name is Sir Lancelot.
   Your quest is to seek the Holy Grail.
   Your favorite color is blue.
   ```

3. **filterPractice** and **filterFun.py** Write a script, 'filterFun.py' that prints the index and value of the arguments, as shown in the example. Then create a Script Tool, 'filterPractice', that points to the script and use filters to allow the user to:

(a) Choose 'hehe', 'hoho', or 'woot' from a combo box.
(b) Choose an integer in the range of −5 to 5.
(c) Choose one or more Point feature classes.
(d) Choose a text file with a '.csv' or '.txt' extension.
(e) Choose a workspace that is either a file geodatabase or a personal geodatabase.

Example input:
woot -3 C:/gispy/data/ch23/tester.gdb/c1;C:/gispy/data/ch23/tester.gdb/c2 C:/gispy/data/ch23/smallDir/poem.txt C:/gispy/data/ch23/rastTester.gdb/

Example output:

```
>>> Argument 0: C:\gispy\sample_scripts\ch23\exercises\filterFun.py
Argument 1: woot
Argument 2: -3
Argument 3:
C:/gispy/data/ch23/tester.gdb/c1;C:/gispy/data/ch23/tester.gdb/c2
Argument 4: C:/gispy/data/ch23/smallDir/poem.txt
Argument 5: C:/gispy/data/ch23/rastTester.gdb/
```

4. **check4Value** Sample script 'C:\gispy\sample_scripts\ch23\exercises\freqCount.py' counts the number of occurrences of a particular value of a given text field in a file. It takes three argument: a file name, a field name, and a field value. Modify the script so that it prints the count in both the Interactive Window and the Geoprocessing Window. Next, create a Script Tool, 'check4Value' that points to 'freqCount.py' and has three parameters:

   (a) The first parameter (File name) should be a required table. Set the default value to C:/gispy/data/ch23/tester.gdb/park.
   (b) The second parameter (Field name) should be required input with choices obtained from the first parameter. Set the filter so that only 'text' type fields are displayed.
   (c) The third parameter (Field value) should be a required string. Set the default value to 'woods'.

   Example input:
   C:\gispy\data\ch23\tester.gdb\park COVER woods

   Example output:

```
>>> Arg 0 = C:\gispy\sample_scripts\ch23\exercises\freqCount.py
Arg 1 = C:\gispy\data\ch23\tester.gdb\park
Arg 2 = COVER
Arg 3 = woods
The COVER field in C:\gispy\data\ch23\tester.gdb\park contains
213 occurrences of woods.
```

5. **batchCount** and **countEach.py** Write a script, 'countEach.py' that uses the Get Count (Data Management) tool and reports the number of records in each input table, as shown in the example. Then create a Script Tool, 'batchCount', that points to the script and has a multivalue input table parameter.

Example input:
C:\gispy\data\ch23\smallDir\trails.shp;
C:\gispy\data\ch23\smallDir\wheatYield.txt;
C:\gis py\data\ch23\training\xyData2.shp

Example output:

```
>>> C:\gispy\data\ch23\smallDir\trails.shp contains 4 records.
C:\gispy\data\ch23\smallDir\wheatYield.txt contains 124 records.
C:\gispy\data\ch23\training\xyData2.shp contains 8 records.
```

6. **smoother** and **smoothLine.py** Write a script 'smoothLine.py' to smooth the input linear feature class using the 'PAEK' smoothing algorithm and the smoothing tolerance given by the user. Then create a Script Tool, 'smoother', that points to the script and has three parameters:

    - The first parameter (Input feature class) should be a required Polyline feature class. Set the default value to C:/gispy/data/ch23/smallDir/trails.shp.
    - The second parameter (Smoothing tolerance) should be a required input linear unit. Set the default to 2000 meters.
    - The third parameter (Output feature class) should be a derived output feature. Set the symbology to 'C:\gispy\data\ch23\training\wideLines.lyr'. Modify the Python script so that the output feature class is named 'smoothLine' and is added automatically to the map. Test the tool in ArcMap to confirm that the yellow highway line symbology is applied when the output is added to the map.

7. Script Tool, 'bCopy' in 'C:\gispy\sample_scripts\ch23\exercises\exercises.tbx', copies a batch of raster or feature class datasets from one workspace to another. The tool takes three parameters, source and destination workspaces, and a dataset type. The tool points to script 'bCopy.py' in the 'C:\gispy\sample_scripts\ch23\exercises' directory. Script Tools 'bCopyProgress1', 'bCopyProgress2', and 'bCopyToMap' are identical to Script Tool 'bCopy'. Also, 'bCopyProgress1.py', 'bCopyProgress2.py', and 'bCopyToMap.py' are identical to script 'bCopy.py'. For each part below, modify the Script Tool to point to the script with the same base name, then modify the scripts as specified.

    (a) **bCopyProgress1** Modify 'bCopyProgress1.py' to report progress using the default progressor. Update the label before each file is copied, to report which file is being copied next.
    (b) **bCopyProgress2** Modify 'bCopyProgress2.py' to report progress using the step progressor. Update the label before each file is copied, to report which file is being copied next. Update the progress bar percent (the progressor position) after each file is processed.
    (c) **bCopyToMap** Modify 'bCopyToMap.py' to collect a list of the output files created. Use the `SetParameterAsText` method to return a semicolon delimited string of output file names. Then add a fourth parameter to the Script Tool. Make this parameter a multivalue Dataset derived output type,

so that the output will be added to the map. Test the tool in ArcMap to confirm that each copied file is automatically added to the map.

8. **slopeComputer** and **slope.py** Write a script, 'slope.py', to report the slope of the line that passes through points, (x1,y1) and (x2,y2). Create a Script Tool, 'slopeCompute', that has four float input parameters, x1, y1, x2, and y2. Use the ToolValidator `updateMessages` method to throw an error, if the user enters a vertical line.

   Example input1: 1 2 1 4

   Example output1: Script Tool posts an error message…
   ```
   The slope of this line is undefined (change x1 or x2).
   ```

   Example input2: 1 2 3 4

   Example output2:
   ```
   >>> The slope of the line passing through (1.0, 2.0) and (3.0, 4.0)
   is 1.0.
   ```

9. **getDate** and **printDate.py** Write a script, 'printDate.py', to report the month, day, and year of a date passed as a string in the format MM/DD/YYYY. Create a Script Tool, 'getDate', that has one input parameter, a date. Label the parameter 'Select a date in 2013'. Use the ToolValidator `updateMessages` method to throw an error, if the user selects a date that is not in the year 2013. The date is passed in as a `datetime` object so you can create two `datetime` objects, one for the beginning of 2013 and one for the end:

   ```
   dstart = datetime.datetime(2013, 1, 1, 0, 0, 0, 0)
   dend = datetime.datetime(2013, 12, 31, 23, 59, 59)
   ```

   Then use comparison operators to check the data input by the user.

   Example input1: 11/30/2013 4:00:46 PM

   Example output1:
   ```
   Month: 11 Day: 30 Year: 2013
   ```

   Example input2: 5/17/2014 12:08:51 PM

   Example output1: Script Tool posts an error message. . .
   ```
   No data available for 2014. Select a date in year 2013.
   ```

10. **twoCat** Copy '02_categories_ip' from 'validatorExamples.tbx' into 'exercises.tbx' and rename it as 'twoCat'. Modify the `initializeParameters` method to organize the parameters into two categories instead of three. Name the categories "I. Required" and "II. Optional". Set the category of each parameter based on its `parameterType` (if the type is 'Required' set it to the first category; Otherwise, set it to the second category).

11. **pointORpoly** Script 'pointORpoly.py' calls one of four geoprocessing tools based on a tool name input by the user. The 'pointORpoly' Script Tool in

‘C:\gispy\sample_scripts\ch23\exercises\exercises.tbx’ points to this script. Run the ‘pointORpoly’ Script Tool five times on a polygon shapefile to test the five tool name options. The first Script Tool parameter is an input point or polygon feature class. The second parameter is a value list of names to specify which tool to call. Currently, any tool can be called for any input. For example, the ‘Points to line’ tool can be selected for point or polygon input features, even though polygons are not appropriate input. Modify the `updateParameters` method in the Script Tool `ToolValidator` class to change this behavior. Specifically, if the shape type of the first parameter is ‘Polygon’, change the filter list for the second parameter to `['Feature to point', 'Minimum bounding box']` and if the shape type is ‘Point’, change the filter list to `['Thiessen polygon', 'Points to line']`.

12. **convertExer.pyt** Create a Python toolbox named ‘convertExer.pyt’. For each part of this exercise, copy and paste the `Tool` class template within the ‘.pyt’ file. Rename it as shown in bold. Modify the class, as specified. Add the tool name to the `tools` list in the `Toolbox` class and check the tools.

    (a) **computeTriArea** This Python toolbox tool should have the same parameters and functionality as ‘triArea’ in #1 above.
    (b) **montyPyBridge** This Python toolbox tool should have the same parameters and functionality as ‘bridgeOfDeath’ in #2 above.
    (c) **filtersGalore** This Python toolbox tool should have the same parameters and functionality as ‘filterPractice’ in #3 above.
    (d) **countValues** This Python toolbox tool should have the same parameters and functionality as ‘check4value’ in #4 above. In the `execute` method, the search cursor will need the field name, not the `Value` object. Set the field name using the `value` property of the `Value` object, as in the following example:

    ```
    fieldname = parameters[1].value.value
    ```

    (e) **bCount** This Python toolbox tool should have the same parameters and functionality as ‘batchCount’ in #5 above.

13. **enableEx.pyt** The Script Tool ‘enableExample’ in ‘exercises.tbx’ has three parameters, an input shapefile, a tool name (‘buffer’ or ‘get count’), and a buffer distance. The third parameter is only applicable when the user chooses to run the buffer analysis. The `updateParameter` in the `ToolValidator` is used to enable/disable the third parameter as needed. Run the tool to see how it works. Then create a Python toolbox, ‘enableEx.pyt’ that contains a tool class named `enableExample` that has the same functionality as the ‘enableExample’ Script Tool.