Chapter 4

Python toolboxes

Introduction

The scripts and script tools you have written so far can be powerful and useful, but they are not easily portable or shared. Often, you need to either share multiple files or know that the recipient is able to open and run Python scripts in an IDE. These options are not user-friendly and would be prohibitive if you were sharing your custom tools with hundreds of users.

Two new features were added in ArcGIS 10.1 to help you write better applications and share them easily with others—Python toolboxes and Python add-ins. This chapter covers Python toolboxes, showing you the benefits of this new feature and how to create and share these toolboxes. Python add-ins are covered in chapter five.

Tutorial 4-1 Creating a Python toolbox

Python toolboxes were introduced in ArcGIS 10.1 with the express purpose of making your script tools easier to program and to share.

Learning objectives

- Understanding Python toolbox components
- Creating and sharing toolboxes
- Defining multiple tools

Preparation

Research the following topics in ArcGIS for Desktop Help:

- "What is a Python toolbox?"
- "Comparing custom and Python toolboxes"
- "Creating a new Python toolbox"
- "Editing a Python toolbox"

Introduction

The Python toolbox has two main advantages over a custom toolbox when it comes to building your own tools. The first advantage is that tools in Python toolboxes are easier to develop because all the code is stored in a single file. A script tool requires a Python script (.py) file to store executable code and a custom toolbox (.tbx) file to store the tool's interface. The user must manually coordinate the code with the toolbox properties for inputs, filters, value lists, and output. A Python toolbox contains the Python code for all the tools and interfaces in a single Python toolbox (.pyt) file, making the scripting simpler to write and share.

The Python toolbox code structure contains different classes. Each class handles a different tool, so your toolbox can contain multiple tools. Within the classes are modules that contain an aspect of the code that may deal with the start-up environment, the input and output parameters, and the business logic for the tools. There are also separate modules to handle validation messages and to check the license and extension availability when any of the tools in the toolbox are used. With all these processes handled in a single file, these applications are easier to code and troubleshoot.

The second advantage is portability. If you are sharing an application that is written as a standard script tool, multiple files must be moved to transfer the application to another machine. If there are issues in the code after the move, both files would need to be scrutinized to find the errors. However, a Python toolbox application can be moved as a single file, and it presents a single set of code for troubleshooting.

There is, however, a downside to the Python toolbox. It cannot contain regular script tools or models—these remain the exclusive domain of custom toolboxes.

Scenario

The application you wrote in tutorial 2-4 is working fine, and you would like to share it with all the members of the library. If you recall, this application took a previously selected site and determined how far out you would have to go to find 200 registered library patrons. Sharing the application will involve putting it on 15 to 25 different machines, so you want to make the tool as portable as possible. To accomplish this task, take the code written for tutorial 2-4, and migrate it to a Python toolbox.

Data

A completed copy of the code is in text file Tutorial 4-1 in the Data folder. This code will access the same data as tutorial 2-4.

SCRIPTING TECHNIQUES

The code for the Python toolbox is separated into several parts, each of which controls a different aspect of the tools you add to the toolbox. The first part is the Toolbox class. The code for the Toolbox class looks like this:

```
import arcpy


class Toolbox(object):
    def __init__(self):
        """Define the toolbox (the 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]
```

Import the ArcPy module.

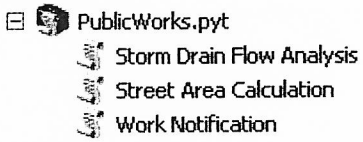
The class Toolbox(object) contains information about the toolbox and the names of all the tools the toolbox will contain. The class name Toolbox is protected and should not be changed.

Within this class, define a toolbox label and a toolbox alias—the same things you would set for a regular toolbox. Make a list of all the class names for the tools this toolbox will contain. For instance, a toolbox for public works that had tool classes named StreetArea, WorkNotify, and StormFlow would have the code shown:

```
import arcpy


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

        # List of tool classes associated with this toolbox
        self.tools = [StreetArea,WorkNotify,StormFlow]
```

Note the list object self.tools that contains a list of all the tool class names within the toolbox.

Your public works Python toolbox would display the tool labels for each class, such as Storm Drain Flow Analysis, Street Area Calculation, and Work Notification, and look like this:



For each tool listed, there is a class containing a set of code to define and control it. The three main components of this class are the initialize (_init_) module, the parameter module, and the code module.

The _init_ module allows you to set a tool label and description and control the availability of background processing. The code for the StreetArea class in the public works example looks like this:

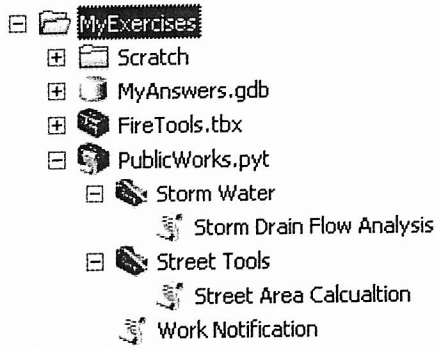
```
class StreetArea(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Street Area Calculation"
        self.description = "The user selects a street and the area of pavement is calculated." + \
                           "The user can specify the measurements in square feet or square yards"
        self.canRunInBackground = False
```

Note that the class defines a label and a description that will be displayed in the toolbox.

Each tool in a Python toolbox is placed either at the root level of the toolbox or into a subcategory called a *toolset* and is always displayed in alphabetical order. Toolsets can be used to help organize tools within a toolbox, making them easier for the user to find. Toolsets are defined in the _init_ module of a tool's class by setting the self.category parameter, as shown in the following graphic. You can have as many tools as you like in a toolset by adding the same category statement to the tool's _init_ module.

```
class StreetArea(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Street Area Calculation"
        self.description = "The user selects a street and the area of pavement is calculated." + \
                           "The user can specify the measurements in square feet or square yards"
        self.canRunInBackground = False
        self.category = "Street Tools"
```

In the following graphic, toolsets were created for Storm Water and Street Tools. Each toolset can contain multiple tools. The tool Work Notification was left outside of a toolset and appears at the root level of the main toolbox, as shown:



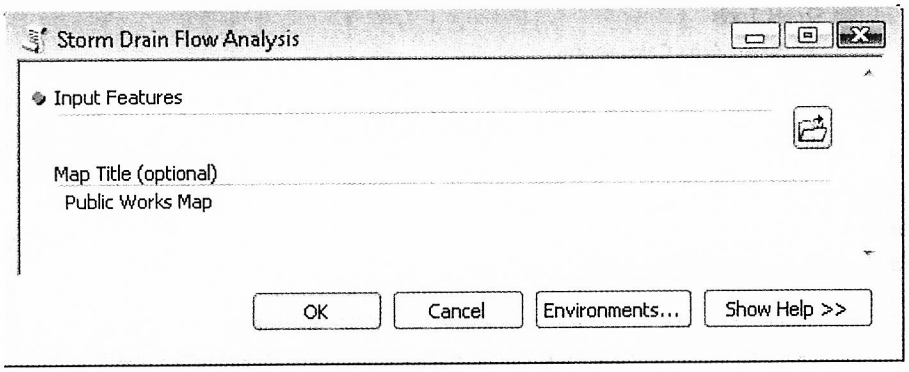
The next component of the tool's class is the parameters module. In this module, all the input and output parameters are set in much the same way as creating a script tool. For each parameter, many properties can be set, including the input name and label, data type, parameter type, and direction. A complete list of the parameter properties and methods can be found in ArcGIS for Desktop Help by searching for "Defining parameters in a Python toolbox" or "Defining parameter data types in a Python toolbox." As an example, the code to accept two parameters for the public works tool would look like this:

```
    def getParameterInfo(self):
        """Define parameter definitions"""
        # First parameter
        param0 = arcpy.Parameter(
            displayName="Input Features",
            name="in_features",
            datatype="DEFeatureClass",
            parameterType="Required",
            direction="Input")

        # Second parameter
        param1 = arcpy.Parameter(
            displayName="Map Title",
            name="mapitle",
            datatype="GPString",
            parameterType="Optional",
            direction="Input")
        # Set a default value
        param1.value = "Public Works Map"

        params = [param0, param1]
        return params
```

Index numbers are used with the parameters the same as in a script tool, and you can set many of the same properties. The list of valid input data types is basically the same as that used for creating a script tool, but the keywords are different for

Python toolboxes. In this example, the feature class data type is DEFeatureClass, and the string data type is GPString. Refer to the ArcGIS for Desktop Help topic "Defining parameter data types in a Python toolbox" for the complete list. The getParameterInfo method returns a list object you define containing the tool's parameters, which can be referenced in the code section of the tool's class.

When the tool runs, each parameter listed appears in the input dialog box and allows user interaction, as shown, which basically replaces the arcpy.GetParameterAsText() command used to get input from the user in a script tool.



One of the most useful properties that can be set in the parameters module is a filter. In the same way that you controlled the input in a script tool, a filter can be set to control the input in a Python toolbox tool. A simple filter might be to set up a value list. The code for setting a value list filter identifies the filter type and defines the list of values. The code shown in the following graphic is setting the filter type to ValueList and then creating a list object with the valid values. A comment in this code lists all the valid filter types, which must be entered as shown:

```
class WorkNotify(object):

    def getParameterInfo(self):
        """Define parameter definitions"""
        param0 = arcpy.Parameter(
            displayName="Material",
            name="pipeMat",
            datatype="GPString",
            parameterType="Required",
            direction="Input")
        # Set the filter type
        # Valid filter types are 'ValueList', 'Range', 'FeatureClass', 'File', 'Field', and 'Workspace'
        param0.filter.type = 'ValueList'
        # Define a list object with the choices
        param0.filter.list = ['PVC', 'Clay', 'Steel', 'AC Conc']
        params = [param0]
```

Another interesting filter might be to set a field name selection that is dependent on a selected feature class. For instance, the user would select a feature class in a tool's first parameter, and the second parameter would present a value list of field names from that feature class. In the example code shown in the following graphic, the first parameter has a data type set to feature class and a filter set to Polygon.

This filter allows the selection of only polygon-type feature classes. The second parameter has a data type of Field, a filter of Text, and a dependency set to the value of the first parameter, which would allow only string fields from the selected feature class to appear in the value list.

```
    def getParameterInfo(self):
        """Define parameter definitions"""
        param0 = arcpy.Parameter(
            displayName="Input Features",
            name="inFeat",
            datatype="DEFeatureClass",
            parameterType="Required",
            direction="Input")
        # Limit the choice of feature class type to polygons
        param0.filter.list = ['Polygon']

        param1 = arcpy.Parameter(
            displayName="Destination Field",
            name="destField",
            datatype="Field",
            parameterType="Required",
            direction="Input")
        # List a specific field type
        param1.filter.list = ['Text']
        # Define a dependency
        param1.parameterDependencies = [param0.name]
        # Only string fields from the selected feature class will be shown

        params = [param0, param1]
        return params
```

Many more filter settings are available for establishing data integrity rules. These settings are found in the ArcGIS for Desktop Help topic "Defining parameters in a Python toolbox." The Help topic lists valid filter types, keywords for data types and field types, and sample code for their use. Remember that the more that you use data integrity rules to control data entry, the fewer errors users will have when using your tools.

The final component of a tool's class is the source code, or execute, module. The execute module contains the code that the tool will use to carry out whatever processes your application requires. In the example shown in the following graphic, two parameters were defined in the parameters module and were then brought into variables in the execute module by referencing their index numbers. The variables can then be used in the code.

```
def execute(self, parameters, messages):
    """The source code of the tool."""
    inputFC = parameters[0].valueAsText
    destField = parameters[1].valueAsText
    # Your executable code goes here. #
    return
```

Notice that the name references for the parameters have changed. When these parameters are defined in the getParameterInfo method, you might use param0 and param1. These parameters are then returned to the script as a list object named parameters[], and they keep the same order and thus the same index numbers. Note that the definition line of the execute module calls the parameters list object, so, for example, param0 becomes parameters[0], and param1 becomes parameters[1].

Each Python toolbox includes a class to define the characteristics of the toolbox itself and a class for each tool it contains. Each tool class, at a minimum, has an _init_ module to define the tool and an execute module to contain the source code. It is also common to have a parameters module to define input and output parameters for the tool, if it has any. Other modules can be used to define validation rules and messages.

EDITING PYTHON TOOLBOXES

Most IDEs do not recognize a Python toolbox .pyt file as a script file, and therefore do not perform code completion or syntax-checking functions. Refer to appendix A for information on how to configure your particular IDE to handle .pyt files.

Also remember to set the Geoprocessing Options in ArcMap to whichever IDE you wish to use. If no preference is set, your operating system may open a text editor that is not suitable for editing Python code.

Create a Python toolbox

1. Open the map document Tutorial 4-1, and pin the Catalog window to the desktop interface.

Shown are the parcels for Oleander and the suggested bookmobile sites that the head librarian suggested.

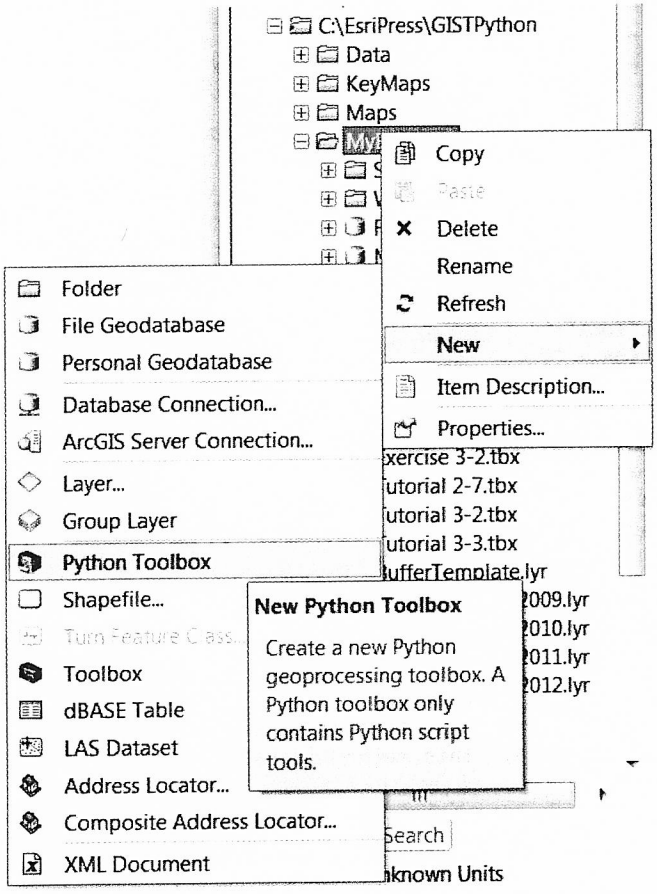
2. In a text browser, open the text file Tutorial 4-1 in the Data folder.

This text file, shown in the graphic, is a completed version of the script you created in tutorial 2-4 to perform the library patron analysis.

```
# Import the modules.
import arcpy

# Set up the environment.
arcpy.env.workspace = r"C:\EsriPress\GISTPython\Data\City of Oleander.gdb\\"
arcpy.env.overwriteOutput = True

# Set up cursor for the bookmobile sites.
arcpy.MakeFeatureLayer_management("Parcels","Parcels_Lyr",'"DU"= 1')
arcpy.MakeFeatureLayer_management("BookmobileLocations","Locations_lyr")

siteCursor = arcpy.da.SearchCursor("Locations_lyr","Marker")
for row in siteCursor:
    siteName = row[0]

    # Select parcels within 150 feet and store as a new selection.
    arcpy.Select_analysis("Locations_lyr",\
    r"C:\EsriPress\GISTPython\MyExercises\Scratch\TemporaryStorage.gdb\SiteTemp",\
    '"Marker" = \'' + siteName + "'")

    arcpy.SelectLayerByLocation_management("Parcels_lyr","WITHIN_A_DISTANCE",\
    r"C:\EsriPress\GISTPython\MyExercises\Scratch\TemporaryStorage.gdb\SiteTemp","150","NEW_SELECTION")

    # Start a while statement until number of dwelling units exceeds 200.
    parcelCount = int(arcpy.GetCount_management("Parcels_lyr").getOutput(0))
    print parcelCount

    myCount = 1
    while parcelCount < 200:
        # All statements at this indent level are part of the while loop.

        # Add to the selected set all property within 150 feet and redo count.
        # SelectLayerByLocation_management (in_layer, overlap_type, select_features, search_distance, selection_type)
        arcpy.SelectLayerByLocation_management("Parcels_lyr","WITHIN_A_DISTANCE","Parcels_lyr","150","ADD_TO_SELECTION")

        parcelCount = int(arcpy.GetCount_management("Parcels_lyr").getOutput(0))
        print parcelCount
        if myCount == 8:
            parcelCount = 200
        else:
            myCount = myCount + 1

        # Exit the while statement when count exceeds 200.

    # Export the selected features to a new feature class using the Marker name.
    arcpy.CopyFeatures_management("Parcels_lyr", r"C:\EsriPress\GISTPython\MyExercises\MyAnswers.gdb\\" + siteName.replace(" ","_"))
    print siteName + " output OK!"
    # Move to the next site and repeat.
```

To create a Python toolbox application from this script, create a new Python toolbox, and configure it to have the source code from the provided file. Creating the new toolbox is simple because it requires only a name.

3. **In the Catalog window, right-click your MyExercises folder and click New > Python Toolbox, as shown. Name the toolbox** Proximity Tools.pyt.



Next, edit the code for the Python toolbox. In a standard custom toolbox, right-click the script tool and click Edit because you will be editing each script separately. In a Python toolbox, right-click the toolbox and click Edit because all the code for all the tools this toolbox contains is in the single .pyt file.

4. Right-click Proximity Tools.pyt and click Edit. The code is displayed as shown:

```
import arcpy


class Toolbox(object):
    def __init__(self):
        """Define the toolbox (the 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 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, parameters):
        """Modify the messages created by internal validation for each tool
        parameter.  This method is called after internal validation."""
        return

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

The default template that was created with the Python toolbox contains the classes and modules you need for the tools. Configure the toolbox name, description, and the list of tools it will contain. This configuration is done in the Toolbox class.

5. **In the Toolbox class, set the label to** Proximity Tools **and the alias to** prox**. Then enter the tool name** PatronSelect **in the self.tools list object, replacing the generic "tool," as shown:**

```
class Toolbox(object):
    def __init__(self):
        """Define the toolbox (the name of the toolbox is the name of the .pyt file)."""
        self.label = "Proximity Tools"
        self.alias = "prox"

        # List of tool classes associated with this toolbox
        self.tools = [PatronSelect]
```

Now you can change the Tool class to accept the parameters for your library tool.

6. **Change the name of class Tool(object) to include your new tool name,** class PatronSelect(object).

7. **As shown, in the _init_ module, set the label to** Bookmobile Patron Selection **and the description to** "The user selects a possible bookmobile location, and the distance is measured to see how far the first 200 patrons will travel to visit it."

```
class PatronSelect(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Bookmobile Patron Selection"
        self.description = "The user selects a possible bookmobile location, and the distance is measured" + \
        " to see how far the first 200 patrons will travel to visit it."
        self.canRunInBackground = False
```

There are no parameters for this tool because the input is a shapefile that the head librarian already created. Move directly to adding the code to the execute module.

8. **Copy the code from the text file starting with the line "# Set up the environment" down to "# Move to the next site and repeat." Paste the code to the execute module right after the comment "The source code of the tool." Be careful to match the indent levels of the existing tool with the new code. Confirm that the indent levels for the for, while, and if statements have been maintained.**

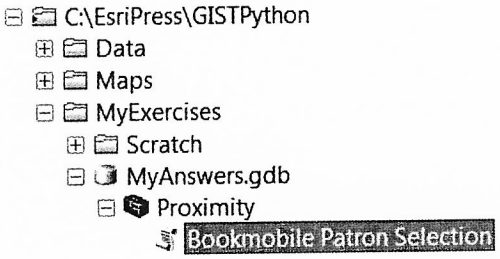
```
    def execute(self, parameters, messages):
        """The source code of the tool."""
        # Set up the environment.
        arcpy.env.workspace = r"C:\EsriPress\GISTPython\Data\City of Oleander.gdb\\"
        arcpy.env.overwriteOutput = True

        # Set up cursor for the bookmobile sites.
        arcpy.MakeFeatureLayer_management("Parcels","Parcels_Lyr",'"DU"= 1')
        arcpy.MakeFeatureLayer_management("BookmobileLocations","Locations_lyr")

        siteCursor = arcpy.da.SearchCursor("Locations_lyr","Marker")
        for row in siteCursor:
            siteName = row[0]

            # Select parcels within 150 feet of the new selection.
            arcpy.Select_analysis("Locations_lyr",\
            r"C:\EsriPress\GISTPython\MyExercises\Scratch\Temporary Storage.gdb\SiteTemp",\
            '"Marker" = \'' + siteName + "\'")

            arcpy.SelectLayerByLocation_management("Parcels_lyr","WITHIN_A_DISTANCE",\
            r"C:\EsriPress\GISTPython\MyExercises\Scratch\Temporary Storage.gdb\SiteTemp",\
            "150","NEW_SELECTION")

            # Start a while statement until number of dwelling units exceeds 200.
            parcelCount = int(arcpy.GetCount_management("Parcels_lyr").getOutput(0))
            # print parcelCount - replace in Python toolbox.
            arcpy.AddWarning(parcelCount)

            myCount = 1
            while parcelCount < 200:
                # All statements at this indent level are part of the while loop.

                # Add to the selected set all property within 150 feet, and redo count.
                # SelectLayerByLocation_management (in_layer, overlap_type, select_features,
                # search_distance, selection_type).
                arcpy.SelectLayerByLocation_management("Parcels_lyr","WITHIN_A_DISTANCE",\
                "Parcels_lyr","150","ADD_TO_SELECTION")

                parcelCount = int(arcpy.GetCount_management("Parcels_lyr").getOutput(0))
                # print parcelCount - replace in Python toolbox.
                arcpy.AddWarning(parcelCount)
                if myCount == 8:
                    parcelCount = 200
                else:
                    myCount = myCount + 1

                # Exit the while statement when count exceeds 200.

            # Export the selected features to a new feature class using the Marker name.
            arcpy.CopyFeatures_management("Parcels_lyr", r"C:\EsriPress\GISTPython\MyExercises\MyAnswers.gdb\\"\
             + siteName.replace(" ","_"))
            # print siteName + " Output OK!" - replace in Python toolbox.
            arcpy.AddWarning(siteName + " Output OK!")
            # Move to the next site, and repeat.
        return
```

The code will work as written, except for the print statements. These are fine for stand-alone Python scripts, but once you move into the ArcGIS environment, you must change these to an ArcPy message command.

9. **Scroll through the code, and make these three replacements of the print command, as shown:**

```
# print parcelCount
arcpy.AddWarning(parcelCount)
#print parcelCount
arcpy.AddWarning(parcelCount)
#print siteName + " Output OK!"
arcpy.AddWarning(siteName + " Output OK!")
```

10. **Save and close the script. Right-click MyExercises and click Refresh to update the code. Your Python toolbox should now look like this:**



11. **Double-click the Bookmobile Patron Selection tool to run it. There are no input parameters, so click OK at the prompt.**

The tool should run exactly as it did as a stand-alone script and produce the patron counts as expected.

Exercise 4-1

Open the map document Exercise 4-1. Edit the code file for the Proximity Tools toolbox. Turn the stand-alone script you wrote for exercise 2-4 into a Python toolbox tool in the existing toolbox. You will need to add the tool to the Toolbox class, then duplicate the PatronSelect tool class, and modify the tool to create the Exercise 2-4 tool class. Be careful with the indentation and variable types, and be sure to replace any Python-only commands, such as print, with ArcPy commands.

Tutorial 4-1 review

The Python toolbox contains and runs code in exactly the same way as a Python script tool, but the interface is much easier to set up. A single .pyt file will hold all the necessary parameters for all the tools that will appear in the toolbox. These parameters can be controlled in exactly the same way as a script tool, but because everything is in a single file, .pyt files are much easier to share with others. A user can receive a .pyt file and place it in any folder. When the file is accessed in the Catalog window, it will appear exactly like any other toolbox without any further action from the user.

Each tool that you add to the toolbox gets its own class, and each class contains separate functions. The various functions create and manage the self object, which is used to store values associated with the function. Within the class, these values can be passed freely from one function to another, allowing one function to interact with or control other functions.

Study questions

1. What programs are able to edit a .pyt file?
2. Can tools be moved from the Python toolbox to custom toolbars? Try this maneuver, or research its possibility in ArcGIS for Desktop Help.
3. What are some of the drawbacks of Python toolboxes?

Tutorial 4-2 Setting up value validation

One of the best things you can have your script do is to validate user input, which may keep users from typing incorrect values that may cause your scripts to crash.

Learning objectives

- Accepting user input
- Setting up data validation
- Formatting Python toolbox messages

Preparation

Research the following topics in ArcGIS for Desktop Help:

- "The Python toolbox template"
- "Defining a tool in a Python toolbox"
- "Accessing parameters within a Python toolbox"
- "Defining parameters in a Python toolbox"

Introduction

Turning a Python script tool into a Python toolbox can get a little more complicated when the script contains user-defined parameters. To define parameters in the stand-alone script, you first have to write the code to accept values into a variable, which involves the use of the command arcpy.GetParameterAsText(), or for outputs, use arcpy.SetParameterAsText(). Then for each