Study questions

1. Look over the list of data types for script tool inputs in the script tool parameters Data Type drop-down list, and give three examples of where specific data types should be specified. Describe what the input dialog box should look like for each data type example you have chosen.
2. The filter option was used to restrict the input for the building types. Give examples of other uses of the filter.
3. Explain why the context-sensitive Help is important.

Tutorial 2-7 Using cursors

Cursors are a programming technique used to step through feature classes and tables, item by item. Each item the cursor finds can be altered independently or used in a geoprocessing task.

Learning objectives

- Working with table properties
- Using cursors with tables
- Using input validation in script tools

Preparation

Research the following topics in ArcGIS for Desktop Help:

- "Make Table View (Data Management)"
- "Understanding validation in script tools"

Introduction

The lists and cursors that you have used up until now have worked mostly with geodatabases and feature classes, but these techniques can also be used on tables. You may want to get a list of tables or use a cursor to go through the rows of a table, one by one.

As with the cursors you have used on features, cursors applied to rows in a table also allow individual processing. You can access the fields and their values and then use decision-making and condition statements to perform a variety of tasks.

Scenario

On your trip to the Fire Department to show off the building count application, you discover another complex process that the department is trying to perform manually (you really need to stop going over there). In any given month, the Oleander Fire Department responds to several hundred calls around the region. Since Oleander has mutual-aid agreements with 16 neighboring cities, the call location may be in any one of these locations. The department needs to split the fire run data that represents each call for service into multiple files with one table for Oleander calls and another table for each of the other cities in which they responded.

The department would also like to geocode the calls, but in the current format, the addresses are parsed into separate fields, so the addresses will need to be combined. You could reformat the address data in the main file, but that would mess up the access to the historical data in the automated dispatch software (the field schema cannot change). This change can only appear in the output files, and not in the source data.

You also cannot use a simple query to split this data because you do not know what cities to use in a selection. The list of cities may be different each month. However, you should be able to make a list of all the city names and use that list to do your table selections.

The solution is to create a temporary copy of the table in memory, make any field changes you need, and calculate the new address field. Use the text slicing technique and a file creation tool to create a new geodatabase that includes the name of the month and the year. Then use a cursor to run through the data to develop a list of unique city names. Use that list to do selections on the database, and create new tables for each unique city name that is found.

You can run this script from ArcCatalog since there are no graphics involved. The tools necessary are listed here so that you can do some research before writing your pseudo code:

MakeTableView

SelectLayerByAttribute

CreateGeodatabase

CopyRows

CalculateField

FieldInfo()

.replace()

The pseudo code will be complex, so think through the process as if you were doing only one record and document it. Then check to see what other events might occur to change the process, and accommodate for these events in your pseudo code. None of the tasks by themselves are that difficult, but combining them will take planning and precision.

Data

The only data used for this tutorial is the run data for each time frame. A couple of examples are given, but once the script is written, it could be used for years to come to perform the same function on any new dataset created from the dispatch software.

SCRIPTING TECHNIQUES

This tutorial combines many of the techniques you have learned in other tutorials. This tutorial uses cursors, table views, for statements, and several different ArcGIS tools. A new technique to try is the use of a with block to combine the cursor and the for statements into a single subset of the code. The format is as follows:

```
with {cursor setup} as {cursor name}:
    for {name to track current row} in {cursor name}:
        # Add your code here to process cursor items
```

There are two major advantages of using a with block. The first advantage is that the cursor and the for statements are defined in two lines of code. The second advantage is that references to any files or map documents in the with block are automatically removed at the end of the code block, even if the code fails, which prevents a crashed script from locking those datasets or map documents.

Another new technique is the use of input validation code. By adding this extra code in the input properties, the programmer can greatly control what the user is allowed to use for input into the script. This technique is another type of data integrity rule that can range from checking for the correct file name to examining the field structure before allowing the selection. Its function is dependent on the validation code you write.

Use cursors

1. **Start ArcCatalog. Research the tools you may need, and write your pseudo code. A set of pseudo code is shown at the end of this tutorial. If your pseudo code differs substantially, try using it for writing the code, and reference the tools as presented in these steps.**

2. **Start your IDE, and set up the template lines for a new script named ProcessFireData. The workspace environment for outputting new files is C:\EsriPress\GISTPython\MyExercises, as shown:**

```
try:
    # Import the modules
    import arcpy
    from arcpy import env

    # Set up the environment
    env.workspace = r"C:\EsriPress\GISTPython\MyExercises\\"
    env.overwriteOutput = True
```

The process starts by asking the user which file they want to work on. In the code, you can use the generic GetParameterAsText() function, and when you create the script tool, set up the data type.

3. **Add the line of code to accept user input into a variable named inTable, as shown:**

```
    # Prompt user for the input table
    inTable = arcpy.GetParameterAsText(0)
    # When this is set up as a cursor tool, set the input to tables only
```

Next, make a copy of the input table in memory with the MakeTableView command. This copy allows you to make selections and queries without affecting the source data, unless you actively save the table view object. You do not need to show all the fields in the output, but you must add a field that can be used to store the concatenated address.

ArcGIS for Desktop Help has a great example of how to do this. Note that you cannot add a field to a table view, but the sample code in the tool reference for MakeTableView shows a way to change the name of a field that is not being used.

First, get a list of all the fields in the input table, and then create an object that will store the information about the fields. This object will be modified and used in the MakeTableView command later on.

4. **Get a list of all the fields in inTable, and store the list in a list object named fields. Then use the FieldInfo() function to create a field object, as shown:**

```
    # Get the fields from the input
    fields = arcpy.ListFields(inTable)

    # Create a fieldinfo object
    fieldinfo = arcpy.FieldInfo()
```

Before you create the table view, alter the field information for the output. This alteration will make certain fields visible in the output and will change the name of addr_2 to GeoAddress, which will later accept the concatenated address string. At the end of the elif statements is an else statement that hides all unidentified fields in the output. The tool reference for MakeTableView has a good example of this type of statement.

5. **Add condition statements to make the list of desired fields visible in the table view. Also, change the name of addr_2 to GeoAddress, as shown:**

```
# Define a fieldinfo object to bring only certain fields into the view
#  inci_no, alm_date, alm_time, arv_date, arv_time, inci_type
#  descript, station, shift, city
#  number, st_prefix, street, st_type, st_suffix
#  (you can't add new fields to a table view, so reuse a discarded one)
#  Change the name of addr_2 to GeoAddress in the output table
# Code was copied and modified from the Help screen

# Iterate through the fields, and set them to fieldinfo
for field in fields:
    if field.name == "inci_no":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "alm_date":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "alm_time":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "arv_date":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "arv_time":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "inci_type":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "descript":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "station":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "shift":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "city":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "number":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "st_prefix":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "street":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "st_type":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "st_suffix":
        fieldinfo.addField(field.name, field.name , "VISIBLE", "")
    elif field.name == "addr_2":
        fieldinfo.addField(field.name, "GeoAddress" , "VISIBLE", "")
    else:
        fieldinfo.addField(field.name, field.name, "HIDDEN", "")
```

When you create the table view, it will contain only the fields you want.

6. **Create a new table view using the user's input table and the field object you defined, and name it** fire_view, **as shown:**

```
# Create a table view of the input table
# The created fire_view table view will have fields as set in fieldinfo object
arcpy.MakeTableView_management(inTable, "fire_view", "", "", fieldinfo)
```

7. **Write the code to reformat the address fields into a single field named** GeoAddress, **as shown. This code is similar to the code you wrote in chapter one.**

```
# Do the address formatting into GeoAddress for the whole table
# Concatenate number + st_prefix + street + st_type + st_suffix and remove spaces
arcpy.CalculateField_management("fire_view","GeoAddress","str(!number!) + ' ' +\
!st_prefix!.strip() + ' ' + !street!.strip() + ' ' + !st_type!.strip() + ' ' +\
!st_suffix!.strip()", "PYTHON")
```

Note that this reformatting only makes changes to the table view and does not change the source data unless that change is specifically intended.

The output of the process must be stored in a new table in a new geodatabase. To get the year of the data's collection, slice the last four digits from the input file name. Then add the year as a suffix to a new geodatabase name, and store the geodatabase in your MyExercises folder.

8. **Use string slicing to get the year for the input data. Then create a new geodatabase with the year as the suffix for the file, as shown:**

```
# Create new geodatabase to store results for year
# ("Fire Files for " + last 4 digits of file name)
gdbName = "Fire_Files_For_" + inTable[-8:]
arcpy.CreateFileGDB_management("C:\\EsriPress\\GISTPython\\MyExercises",gdbName)
```

The eventual goal is to create a separate database for each of the different city names containing only the records for each city, but you do not know what those city names will be. In some months, there may be only four or five cities with responses, but in a busy month, the number can grow to as many as 14. You can find all the values by having the cursor iterate through the records to look for unique names, and then store the values in a list variable. Remember that a list variable can store multiple values, which are retrieved using an index number.

You can initialize an empty list variable by making it equal to a set of empty brackets ([]). As you find the unique values, they can be determined to be in the list and, if necessary, appended to the list. Check your Python reference for the syntax of these functions. A with statement can set up the cursor and the framework for the iteration. Then a for statement can run through all the records, and an if statement can determine whether the found value of cityName is already in the list.

9. **Add the code to set up the with block, which will scroll through the rows looking for city names, and create a list variable to store the unique results. If you like, add an ArcPy message to show that the list was created successfully, as shown:**

```
# Use cursor to find each unique city name and add it to a list.
# City names included may differ from file to file.
# Set up a list to hold unique city names.
cityList = []

# Start cursor iteration
for row in fireCursor:
    cityName = row[23]
    if cityName not in cityList:
        cityList.append(cityName)
# Result is a list object with all the unique values of the CITY field
del row
arcpy.AddWarning("Made the list of city names.")
```

You are ready to create the new tables. By going through the new list of city names, you can get both the value to use in a search of the table view and the name of the new output table.

10. **Use a for statement to go through the city names list, select all the records from that city, and write a new table to the geodatabase you created in step 8, as shown:**

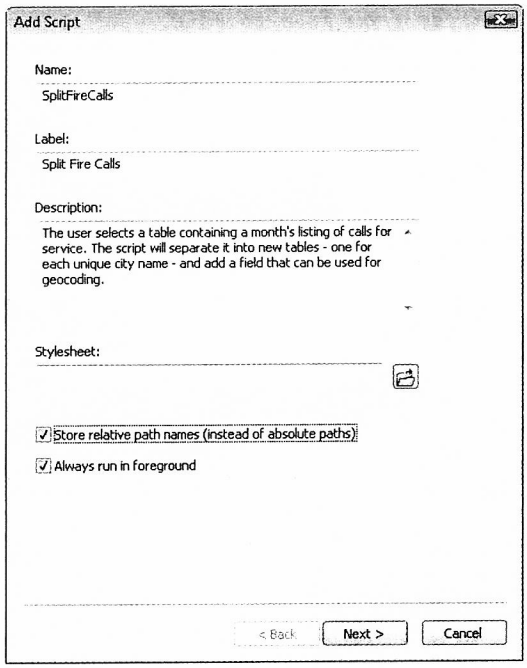
```
# Use the names in the list object to select records
for name in cityList:
    cityQuery = '"city" = \'' + name + '\''

    arcpy.SelectLayerByAttribute_management ("fire_view", "NEW_SELECTION",cityQuery)

    newTable = "C:\\EsriPress\\GISTPython\\MyExercises\\" + gdbName + ".gdb\\" + \
    name.replace(" ","_")

    arcpy.CopyRows_management ("fire_view", newTable)

    itemCount =int(arcpy.GetCount_management("fire_view").getOutput(0))

    arcpy.AddWarning("A table called " + newTable + " was created with " + \
    str(itemCount) + " rows.")
```

Note that when the new table is created, the spaces in the city names are replaced with underscores because ArcMap does not accept a space or a special character in the name of a table. Notice also that code was added to count the number of features and to report the names and record counts of the new tables back to the Results window as the tables are created. Although this action is not required, it is helpful to let the user know what the script is doing.

11. **Save the script, and close your IDE.**

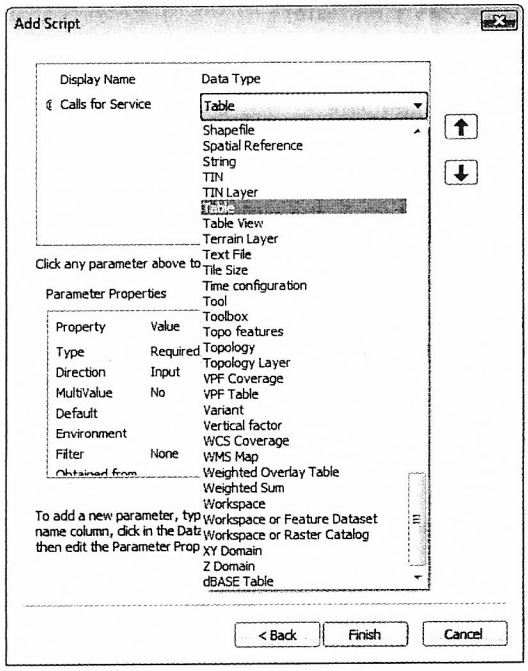
The script takes only one input—the fire department calls for service—so your script tool will need only one input, which you can restrict to tables.

12. Start ArcCatalog. In the Catalog window, create a new toolbox in your MyExercises folder named Tutorial 2-7.tbx. Right-click the toolbox and click Add > Script. As shown, name the tool SplitFireCalls, and add an appropriate label and description. Select the "Store relative path names" check box and click Next.

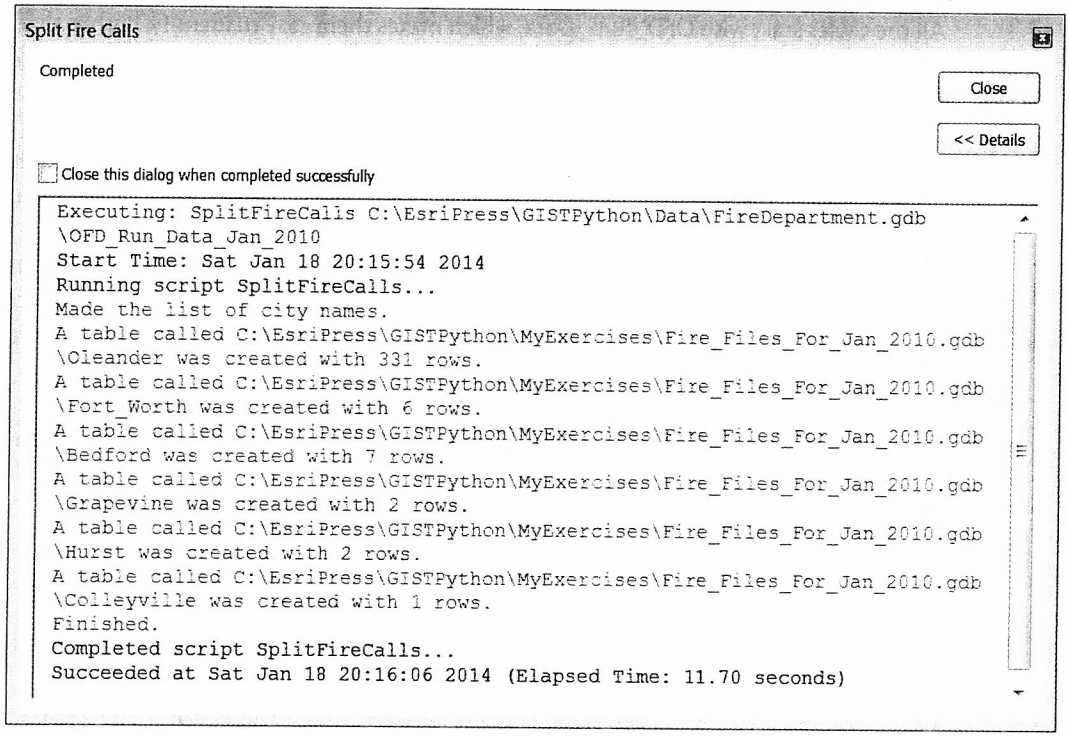


13. Navigate to the folder where you stored the script and select it. Click Next.

14. Set the display name to Calls for Service and the data type to Table, as shown. Click Finish.



15. **Run the script tool, and test it with the January run data in C:\EsriPress\GISTPython\Data\FireDepartment.gdb, as shown. Debug the script tool, if necessary.**



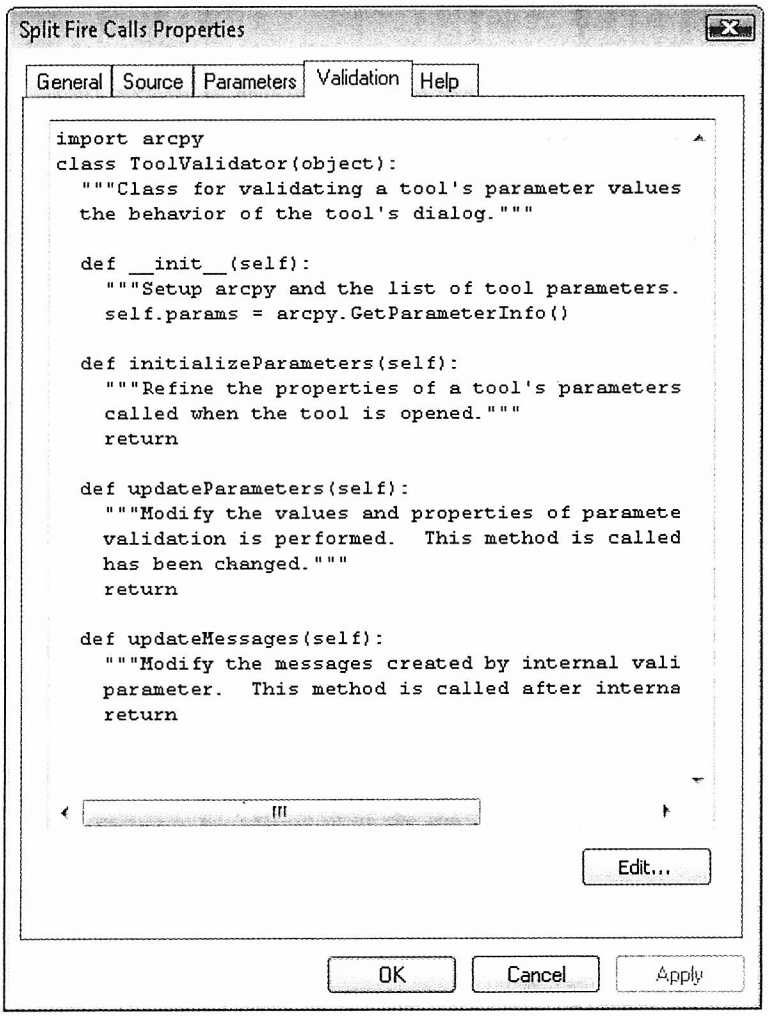
Here's the pseudo code for this script:

```
# Import the modules.
# Set up the environment.
# Prompt user for the input table.
# When this is set up as a cursor tool, set the input to tables only
# Get the fields from the input.
# Create a fieldinfo object.
# Define a fieldinfo object to bring only certain fields into the view.
#  inci_no, alm_date, alm_time, arv_date, arv_time, inci_type
#  descript, station, shift, city
#  number, st_prefix, street, st_type, st_suffix
#  (you can't add new fields to a table view, so reuse a discarded one)
#  Change the name of addr_2 to GeoAddress in the output table
# Code was copied and modified from the Help screen.
# Iterate through the fields, and set them to fieldinfo.
# Create a table view of the input table.
# The created fire_view table view will have fields as set in fieldinfo object.
# Do the address formatting into GeoAddress for the whole table.
# Concatenate number + st_prefix + street + st_type + st_suffix and remove spaces.
# Create new geodatabase to store results for year ("Fire Files for " + last 4 digits of file name).
# Create a cursor to go through the table view row by row.
# Use cursor to find each unique city name, and add it to a list.
# City names included may differ from file to file.
# Set up a list to hold unique city names.
# Start cursor iteration.
# Result is a list object with all the unique values of the CITY field.
# Use the names in the list object to select records.
# Repeat for all names in the list.
# Use the script to create a script tool.
# Add validation code to the script tool.
```

The tool runs fine and creates the required outputs, but there is still one problem: the user could accidentally enter a table with the wrong formatting, and the script would fail. It would be helpful to test the input file first and make sure it contains the correct type of data in the correct format. All the tables start with OFD_Run_Data, which makes them easy to identify. Rather than put this identifier in the script and have the script stop running if incorrect data is provided, you can instead have the tool input validate the user's selection before the tool runs.

You have seen this practice with system tools: the user enters a value for a parameter, and a red *X* appears next to the entry and prevents the tool from running. This preventative technique is called *tool validation* and is available for use on custom scripts as well. The tool validation code is accessed through the script properties and allows you to check the user's input as values are entered, change the tool parameters based on the input, and change output messages based on certain conditions. For this script, add a check at the tool's initialization to make sure the first 12 characters are OFD_Run_Data before allowing the user to click OK.

16. **Right-click the new script tool SplitFireCalls, click Properties, and click the Validation tab.**



The Validation tab shows the standard Python code for the validation object associated with the tool. All the code here is required; you can add to it, but you cannot delete anything. The object created is named *self* and has an index to each parameter value that the toolbox asks for, starting with zero (0). For example, the first parameter would be self.params[0], and the fifth parameter would be self.params[4]. This object has methods, which can be used to raise an error, and parameters, such as value, data type, and a True/False flag, to identify whether the value has been altered in the input box. Using these methods and parameters, you will be able to control the data entry with more precision than with the normal filter in the script tool entry box.

ArcGIS for Desktop Help has more information on tool validation under "Programming a ToolValidator class." Research this topic, and follow along with the code that follows.

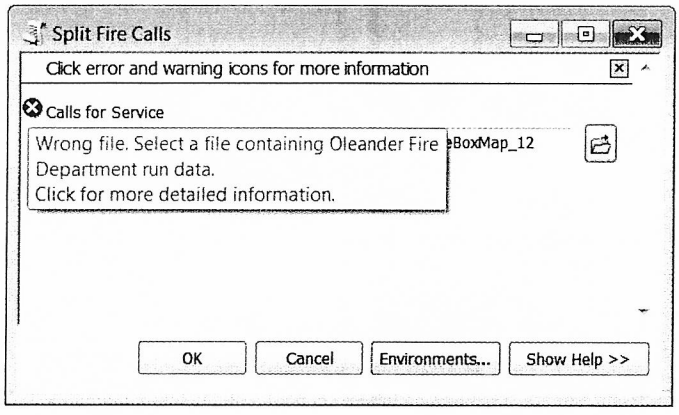
The first step is to get the input value from the user interface. When the script detects that the input value has been changed, you can check its file name to make sure that the first 12 characters use the proper naming convention. If not, an error message appears, blocking the user from continuing.

17. **Click Edit to open your IDE with this script. Under updateMessages, add the following code:**

```
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:
      inputValue = arcpy.Describe(self.params[0].value)
      if not inputValue.file[:12] == "OFD_Run_Data":
          self.params[0].setErrorMessage("Wrong file." +\
          " Select a file containing Oleander Fire Department run data.")
  return
```

Note that the code is first using the .altered property to check whether the user entered anything. If so, it describes the data into a variable named inputValue. Using the arcpy.Describe method slices the input value into its various components, including the path, file name, and extension. Using the file property and slicing the first 12 characters, the code then checks that the file uses the correct naming convention. If it does not, an error message is returned, which adds the red *X* on the input screen and lets the user know that the entry is not valid.

18. **Close your IDE, and save the code. In the Script Properties dialog box, click Apply. Your code now appears in the validation code block. Click OK. Run the script tool with both a valid and an invalid entry. Shown is the error message that appears with an invalid entry:**



19. Alter the script's description to add useful help in the standard tool Help dialog box.

Using the tool validation in conjunction with the tool validation code is a fantastic way to put custom restrictions on user input and to prevent users from accidentally running a tool incorrectly. Research this topic more in ArcGIS for Desktop Help to fully understand how much control you have.

Exercise 2-7

After the Fire Department separates the data by city, it would like to select only the Oleander data and separate it into the different incident types (inci_type). Again, no set list exists of what incident types the file will have, and the output file name should indicate the month and year of the data.

Create a separate script tool to reprocess the results of the Split Fire Calls tool to split the Oleander calls from the resulting file.

Add controls in the tool validation code to make sure the correct file is chosen before the tool runs.

Tutorial 2-7 review

This application uses a variety of techniques learned from previous tutorials and introduces the idea of input validation, which can be extremely important in script tools that might be used by a large number of users who may possess different levels of GIS knowledge. This concept is known as data integrity—the use of code to keep users from creating errors in your data and to help them make correct choices. The more data integrity techniques you can incorporate in your applications, the easier it will be to maintain the integrity of your dataset.

Study questions

1. Make a list of the techniques used by this application that you learned from other tutorials.
2. Give examples of other uses of input validation code.
3. Which input data types allow for filters and validation code?

Tutorial 2-8 Combining loops

Any of the looping techniques, such as while and for statements, can be combined to provide a flexible method of working with data in Python.

Learning objectives

- Validating data
- Summarizing data
- Managing flow of processing

Preparation

Research the following topics in ArcGIS for Desktop Help:

- "Summary Statistics (Analysis)"
- "Get Count (Data Management)"

Introduction

By completing all the previous tutorials in this book, you have learned many different analysis and programming techniques. These techniques include branching using if statements, for and while loops, cursors, and lists. This tutorial uses many of these techniques in a single script tool and requires investigation into how the analysis processes work, how the user will be using this script tool, and what tools and techniques you will need to incorporate into your script tool. This tutorial also requires detailed pseudo code, which you should be an expert at by now.

Scenario

The Oleander Public Works Department in the Engineering division is gearing up for a new storm water testing program to make sure that the water being discharged into the streams and rivers meets the federal clean-water standards. You already have a dataset of the storm drain system, and thanks to the department's summer intern, who slogged up and down every creek in Oleander with a GPS unit, you also have the locations of all the outfalls. These locations are the points at which a storm drain collection system for a particular watershed empties into the creek, and there are many of these outfalls. The department has set up monitoring stations at a few of the locations to start taking water quality readings, and a system name has been given to each station. To complete the analysis, the department needs to know the characteristics of the drainage system for the watershed connected to the particular outfall at the monitoring station. These descriptions need to include an inventory of the fixtures attached to the watershed system, along with a summary of pipe sizes. From this information, the department can calculate maximum capacity and determine whether the readings at the outfalls are within specifications.

The script tool you will write to automate the process will have the user select one outfall, and then the tool will trace the connected pipes until all are selected. This tool can be used to create the pipe inventory. Then the tool will select and form an inventory of the fixtures attached to these pipes. A data maintenance task can also be completed while you're at it. The fixtures are supposed to have a value for the size pipe they are connected to, but that data was not

populated when the storm drain database was created. This problem will be easy for your script to solve as it goes through the features.

Data

The data provided is the storm drain data for Oleander. The data includes a pipeline database with a field named PipeSize and a fixtures database with a field named Type, which identifies the fixture type. These fields will be used for the summaries of each outfall. Both datasets have a field named System, into which you will store the system name that has been assigned to the metering station. The fixtures dataset also has a field named PipeSize, in which you can store the size of pipe that each fixture is connected to.

SCRIPTING TECHNIQUES

You have used all the techniques required for this tutorial before, but here is some help on the pseudo code.

The user will select an outfall before running the script. As an extra data integrity rule, make sure only one feature is selected before you continue. An if statement can check this condition.

Start selecting the lines connected to the outfall. The first selection will be to get the line connected to the outfall, and the next selection will be to select the other lines connected to the first selected line. This process will repeat until all the lines in that particular watershed system have been selected. How will you know when to stop selecting? Try setting up a count of the selected features, and then use a while statement to see whether the number has increased after an iteration of the selection process. If the number has increased, keep on selecting. Once the number remains the same through an iteration, you are done.

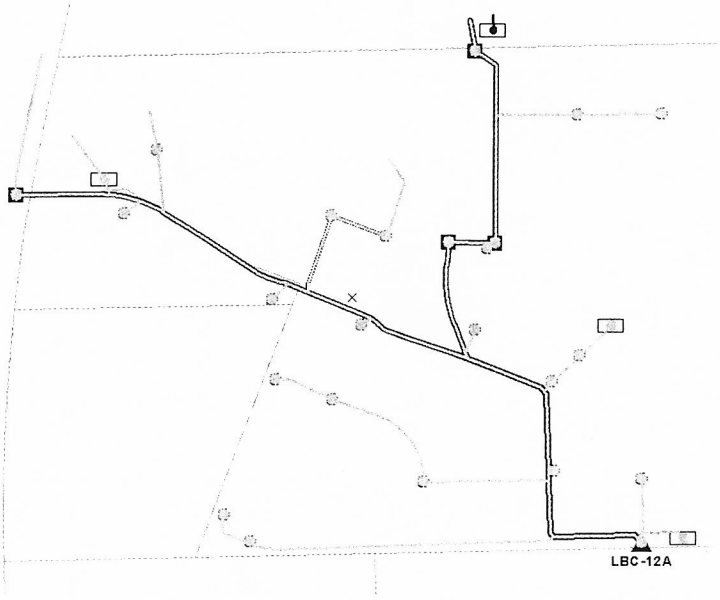
The selection of the attached fixtures is straightforward and can be done in a single command. Then you can use a cursor to go through the selected set of features one by one and get the pipe size from the attached line and transfer it to the point.

The final processes of storing the system name and performing the summaries can be done with a single step for each process.

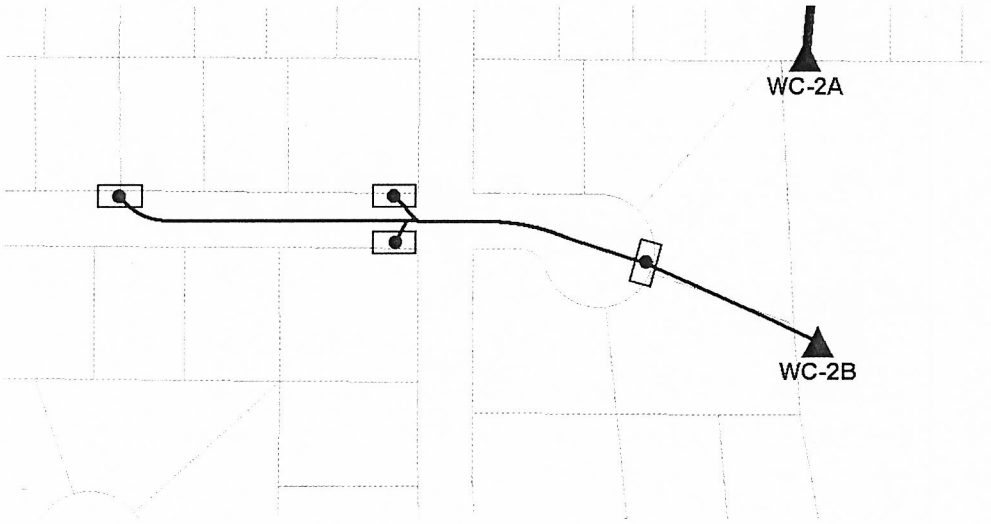
This task sounds simple, but it requires some careful coding to make sure all the interrelated steps occur in the correct order. As with the other tutorials, perform it in steps, and code and debug each individual process before adding the next process. Be sure to write detailed pseudo code! Note that there is no sample pseudo code given for this script.

Combine loops

1. Open the map document Tutorial 2-8. An overall view of the storm drain system at the north end of Oleander is displayed. The purple triangles represent the outfalls for the study, as shown in the graphic. Open bookmark LBC-12A to zoom to the first outfall. From this point, you can get an idea of what a single drainage shed system looks like. Make a manual selection of the outfall, and then use the other selection tools to select all the lines and fixtures associated with this drainage shed. Also investigate the attribute tables of these feature classes, and look for the fields you will be working with. This inspection will give you an idea of what the script will be doing.



2. For simplicity, start with a simpler drainage shed, so move to bookmark WC-2B, as shown in the graphic. Once you have the script written and debugged, move on to more complex drainage sheds.



3. After you have familiarized yourself with the process and written your pseudo code, begin writing the script. Open your IDE, and create a new script file named DrainageshedAnalysis.py with the standard lines to import module and set the working environment.

The selection process is at the heart of this script, so you should write and debug that part of the code first and create the script tool. Then, as you continue to edit the script, the script tool will gain more functionality. Start your code by making sure that only one feature is selected.

4. Write the step to make sure that only one feature in the Fixtures feature class is selected. Otherwise, send a message to the user to let them know that the script will not run, as shown:

```
# Check to make sure that only one feature is selected before continuing
fixCount = int(arcpy.GetCount_management("Fixtures").getOutput(0))
arcpy.AddWarning("The number of features selected is " + str(fixCount) + ".")
if fixCount <> 1:
    arcpy.AddError("The number of selected features MUST be only one." +\
    " Prepare a new selection and try again.")
    raise exception
else:
    arcpy.AddWarning("Only one feature is selected and the script is continuing ...")
```

Next, get the drainage system name. You could prompt the user to supply it, or you could retrieve it from the field System in the Fixtures feature class. Determining the drainage system name sounds like a simple process, but you must set up a cursor and use the row object with the getValue object to get the value of the single feature.

5. Set up a cursor using the Fixtures feature class. Then set up a variable to accept the value of the field System, as shown:

```
# Set up a cursor to iterate through the selected row of Fixtures
# arcpy.da.SearchCursor(in_table, field_names, {where_clause},
#          {spatial_reference}, {explode_to_points}, {sql_clause})
# By only specifying one field name, the value of row[0] contains that field value
fixCursor = arcpy.da.SearchCursor("Fixtures","System")
for row in fixCursor:
    systemName = row[0]
    arcpy.AddWarning("You are working on the " + systemName + " system.")
```

You will be making selections of both the storm drain lines and the fixtures, so make a feature layer for each of the feature classes. Remember that making selections inside a script can only be done on feature layers. An interesting note is that if you made the fixtures feature layer using the feature class in the current table of contents, the layer would contain only one feature because it has a selected feature. Instead, you should make the feature layer from the data in the geodatabase.

6. Add the code to make a feature layer for the line and point feature classes, as shown:

```
# Selections can only be done on feature layers,
#    so create one for the lines ...
stormLinesLayer = arcpy.MakeFeatureLayer_management("Main_at")
# ... and one for the fixtures
fixturesLayer = arcpy.MakeFeatureLayer_management(\
r"C:\EsriPress\GISTPython\Data\StormDrainUtility.gdb\Storm_Drains\Fixtures")
# Note that the feature layer was made by referencing the data's location
#          in the geodatabase and not in the map document
# Referencing the data in the map document would only get the currently
#          selected features
```

7. **Add code, as shown in the graphic, to use the selected outfall to select the line(s) that intersect it. Note that this process does not use the feature layer, but rather the feature class from the table of contents with the selected feature.**

```
# Use the selected outfall to select the first line(s) connected to it
# Selection type will be "Create new selection"
# SelectLayerByLocation_management (in_layer, {overlap_type}, {select_features},
#           {search_distance}, {selection_type})
arcpy.SelectLayerByLocation_management(stormLinesLayer, \
"INTERSECT","Fixtures","","NEW_SELECTION")
```

This code selects the first line (or in some cases, more than one line), which can now be used to select other lines. You can select lines that intersect this line and repeat until you have selected all the lines in this watershed system, or basically select until the count of selected features no longer increases. To accomplish this task, you can set up two count variables to hold the current count of selected features and the new count of selected features. When these two counts are equal, it means that the selection process did not select any new features and is therefore complete.

8. **Add the code to set up two count variables to hold the feature counts, as shown:**

```
# With the first line feature selected, use it to select the other lines
# Set up two feature count variables
# The first will be the current selection
lineCount1 = int(arcpy.GetCount_management(stormLinesLayer).getOutput(0))
# The second will be the new selection (initialize outside the while loop)
lineCount2 = 0
```

9. **Add the while statement and the code to select and check the count of new features, as shown:**

```
# Set up a while statement - it will end when the current selection equals the new selection
while lineCount1 <> lineCount2:
    lineCount1 = lineCount2
    # Use selected features to select intersecting features
    # Selection type will be "Add to selection"
    arcpy.SelectLayerByLocation_management(stormLinesLayer, "INTERSECT",stormLinesLayer,\
    "","ADD_TO_SELECTION")
    # Get count and match against previous count to see if the entire
    #       drainage shed system is selected
    lineCount2 = int(arcpy.GetCount_management(stormLinesLayer).getOutput(0))
    # Message to keep track of selections
    arcpy.AddWarning("previous set = " + str(lineCount1) + " and new set = " + str(lineCount2))
```

The while statement checks its condition before it runs. Inside the statement, set the first count variable to the previous count and the second count variable to the new count. When the statement repeats, it checks the condition again. A message line is included so that the user can check the results on the fly. This line is useful for debugging, too. As an option, you may want to include a message to signify that the selection process is finished and to report how many features are in the selected set, as shown:

```
    # Message to show end of line selections
    arcpy.AddWarning("Finished selecting: " + str(lineCount1) + " = " + str(lineCount2))
```

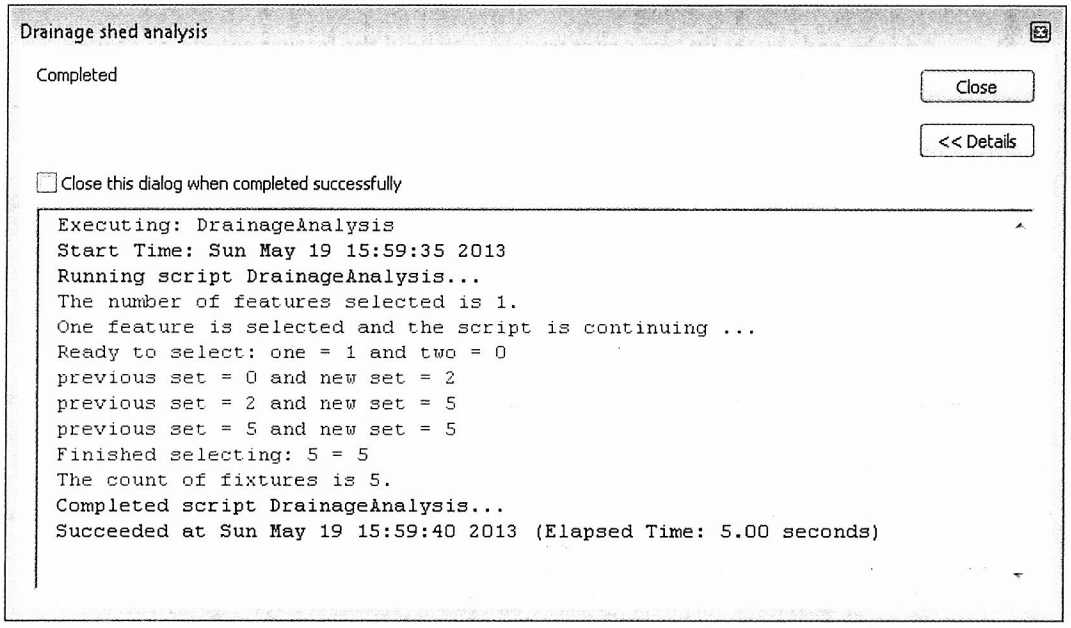
To complete the selection process, select the fixtures that intersect the selected line features.

10. **Add the code to select the point features based on their intersection with the selected line features, as shown:**

```
# Select only the fixtures that intersect the selected line features
arcpy.SelectLayerByLocation_management(fixturesLayer,"INTERSECT",stormLinesLayer,\
"","NEW_SELECTION")
fixCount = int(arcpy.GetCount_management(fixturesLayer).getOutput(0))
arcpy.AddWarning("The count of fixtures is " + str(fixCount) + ".")
```

The selection processes are all complete, and you can go ahead and create the script tool and test it.

11. **Create a new toolbox in your MyExercises folder named Engineering, and add a new script tool using the DrainageshedAnalysis.py script, as shown in the graphic. The script will have no input parameters. When this is done, select the outfall for system WC-2B, and run the script. You may also want to try running it without any features selected to test the count-checking routine.**



If everything checks out, continue with the script. Otherwise, finish debugging the selection processes.

The next set of processes to code is the summary statistics. For the lines, the storm water supervisor wants a table showing a summary of length categorized by line size. For the points, the supervisor wants a count of features categorized by type. If the summaries are done on the feature layers, they will contain information from only the selected features, so the code for the summaries should follow the selection of the fixtures. The names can include the system name you extracted earlier followed by either _StormLineSummary or _StormFixtureSummary.

12. Add the code for the Summary Statistics tool for both the lines and the fixtures, as shown:

```
# Perform a summary statistics process on the storm drain lines
# Statistics_analysis (in_table, out_table, statistics_fields, {case_field})
arcpy.Statistics_analysis(stormLinesLayer,r"C:\EsriPress\GISTPython\MyExercises\MyAnswers.gdb\\" + \
systemName.replace("-","_") + "_StormLineSummary", [["Shape_Length", "SUM"]], "PipeSize")

# Perform a summary statistics process on the fixtures
arcpy.Statistics_analysis(fixturesLayer,r"C:\EsriPress\GISTPython\MyExercises\MyAnswers.gdb\\" + \
systemName.replace("-","_") + "_StormFixtureSummary", [["Type","COUNT"]],"Type")
```

This procedure fills the requirements of the Public Works Department, and the department can use the two output tables for its calculations.

Exercise 2-8

The city has done an elevation study of the creeks in Oleander and wants to do a more detailed study in 3D using this new data, which can be found at City of Oleander > Planimetrics > AnalysisCreeks. Each segment has a field named Slope, which contains the creek's measured percentage of slope. The elevation of the outfalls (where the smaller creeks enter the major waterways) has also been recorded for each creek network, but for 3D analysis, each creek line segment must have the input and output elevations recorded. The fields for the input and output already exist as FlowLine_In and FlowLine_Out, respectively. Only FlowLine_Out for the outfall segments is recorded, which is shown in blue in the map document. Write a script to populate the rest of the flow line fields.

The user will select one of the creek outfalls and run the script tool you create. The tool will trace the creek line by line, calculate each new elevation upstream, record it, and then move on.

The key to this process is to calculate the change in elevation from one end of the line to the other. The change in elevation is found by multiplying the slope by the segment length and dividing by 100 (because the slopes are percentages). The change in elevation can then be added to the FlowLine_Out value to get the next FlowLine_In value. Repeat this step for each segment, moving upstream and using the last calculated value of FlowLine_In as the FlowLine_Out value for the next segment.

(**Hint:** This process will work until you hit a fork in the creek. Why is this a problem? What can be done to fix it?)

Use the map document Exercise 2-8 for this project.

Tutorial 2-8 review

All the techniques used in these processes were used in previous tutorials and exercises, except in this tutorial no pseudo code was given for the script. By now, you probably appreciate the importance of pseudo code in keeping track of all the steps the code must complete and the order in which they must occur. If you completed this tutorial successfully, designing and coding all the processes without the aid of prewritten pseudo code, you are well on your way to becoming a Python programmer.

Study questions

1. What was the hardest part about writing pseudo code for this project?
2. Does it help or hurt to add messages in the code to report progress?
3. The potential for an endless loop existed in this project. Can you explain what it was and how you mitigated it?

Tutorial 2-9 Creating custom toolbars

Script tools make running your tasks from a custom toolbox faster and easier, but these tools may also be placed on a custom toolbar, making them more convenient.

Learning objectives

- Creating custom toolbars
- Adding tools and scripts to custom toolbars

Preparation

Research the following topics in ArcGIS for Desktop Help:

- "Creating a new toolbar"
- "Adding and removing tools on menus and toolbars"

Introduction

Toolbars are easy to create and can contain your custom script tools, models, and system tools. You can categorize items on a toolbar, and even make drop-down submenus. It is important to note, however, that the tools still operate as stand-alone tasks and do not interact with each other. Also, any custom toolbars you create through the customized interface exist only in the map document in which they are created. These toolbars cannot be shared easily with other users.

Scenario

The script tool you created in tutorial 2-8 requires that an item be selected before the tool runs. It would make things easier for the user if you created a separate toolbar and assembled all the tools together to accomplish the task, including the script tool, the Select Features tool, and the Clear Selected Features tool. As an option, you can include the Continuous Pan tool to make moving around the map area easier.