Chapter 11
Batch Geoprocessing

Abstract Python enables you to perform geoprocessing tasks on batches of Esri format data files, fields, and workspaces. This chapter focuses on how to get lists of Esri data and use these lists together with `FOR`-loops for batch geoprocessing.

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

- List the file rasters, feature classes, workspaces, datasets, workspaces, and tables within a workspace.
- List files with a common type and/or substring in the name.
- List the fields in an attribute table.
- Get the name, type, and length of an attribute table field.
- List the printer names, tool, toolboxes, and environment settings for an ArcGIS install.
- Batch geoprocess lists of GIS data.
- Step through each line and watch variables to detect bugs.

11.1 List GIS Data

Python enables you to perform geoprocessing tasks on batches of data files, fields, and workspaces. The `os.listdir` function lists all the files in a directory, but we often need to perform batch processing on a specific GIS file type. The `arcpy` module has a set of listing methods to get lists of these items. The `arcpy` overview diagram shows two of these methods in the left column under 'listing'. The large box in Figure 11.1 shows a more extensive list of these methods. Categories I and II methods are used for listing data. As the names imply, these methods return lists of datasets, feature classes, files, fields, and so forth. This chapter discusses how to use these methods.

Using the Category I methods involves setting the workspace and then calling the method with the usual `object.method` format. In this case, the object is

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_11

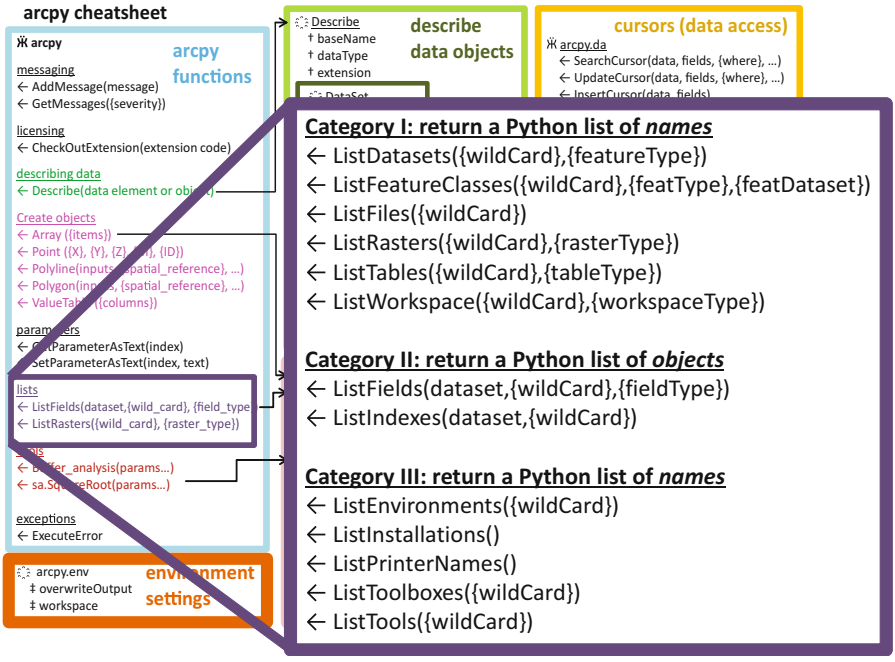


Figure 11.1 The arcpy cheat sheet shows two popular `arcpy` listing methods (`ListFeatureClasses` and `ListFields`) in the left column. The enlarged box on the right has a more complete list of listing methods.

`arcpy`. In the following code, the `ListFeatureClasses` method returns a list of the feature classes in the workspace:

```
>>> import arcpy
>>> arcpy.env.workspace = 'C:/gispy/data/ch11'
>>> # The ListFeatureClasses method returns a Python list of strings.
>>> fcs = arcpy.ListFeatureClasses()
```

The `arcpy.env.workspace` property must be set before `ListFeatureClasses` is called to determine where it looks for feature classes. Category I data listing methods return a list of names in the current workspace. In the example above, the `ListFeatureClasses` method returns a Python list containing the names of feature classes in 'C:/gispy/data/ch11', as shown here:

```
>>> fcs
[u'data1.shp', u'park.shp', u'USA.shp']
```

Each item in the list is a Python unicode string data type. The `u` that precedes the string indicates that it is a unicode string. You can ignore the `u` and use unicode string data types as you use `'str'` string data types. The following code prints the first item in the list and its data type:

```
>>> fcs[0]
u'park.shp'
>>> type(fcs[0])
<type 'unicode'>
```

Since `fcs` is a Python list, we can use it in a `FOR`-loop as follows:

```
>>> for fc in fcs:
...     print fc
...
park.shp
CT_historic_landmarks.shp
CT_national_trails.shp
NEROfires.shp
workzones.shp
```

The other Category I data listing methods, `ListDatasets`, `ListFiles`, `ListRasters`, `ListTables`, and `ListWorkspaces` are similar to the `ListFeatureClasses` method. These methods each list their data for the current `arcpy` workspace. This means you need to set the workspace before calling them and they will return a Python list of the names of items in that workspace.

> **Note** The `arcpy.env.workspace` must be set *before* calling Category I data listing methods—`ListDatasets`, `ListFeatureClasses`, `ListFiles`, `ListRasters`, `ListTables`, or `ListWorkspaces`.

'Datasets', 'files', 'tables', 'rasters', and 'workspaces' are fairly broad terms. If you're not sure which types of files these methods will list, check the ArcGIS Resources site. Or, if you have specific data in mind, it's easiest to simply try it on sample data—set the workspace, call the data listing method, and print the returned list. Example 11.1 lists the Esri workspaces found in 'C:/gispy/data/ch11'. According to this function, a file geodatabase (e.g., 'C:/gispy/data/ch11\tester.gdb'), as well as a folder (e.g., 'C:/gispy/data/ch11\pics') is a workspace. The second loop lists the tables found in 'C:/gispy/data/ch11'. In this example, it returns some 'csv' files, 'txt' files, and 'dbf' files. Others items not listed here might qualify as well. All 'txt' files are listed, regardless of content ('loveLetter.txt' has no field headers or records). Not all 'dbf' (dBASE) tables are listed. Only independent dBASE tables

in the workspace are listed. Dependent tables are ones associated with shapefiles. Even though 'park.dbf' is in 'C:/gispy/data/ch11', this table is not listed because it is part of a set of files that make up a shapefile.

Example 11.1: Call an `arcpy` listing method and loop through the results.

```
# listStuff.py
# Purpose: Use arcpy to list workspaces and tables.
import arcpy

arcpy.env.workspace = 'C:/gispy/data/ch11'

print '---Workspaces---'
workspaces = arcpy.ListWorkspaces()
for wspace in workspaces:
    print wspace

print '\n---Tables---'

tables = arcpy.ListTables()
for table in tables:
    print table
```

Printed output:

```
>>> ---Workspaces---
C:/gispy/data/ch11\pics
C:/gispy/data/ch11\rastTester.gdb
C:/gispy/data/ch11\tester.gdb

---Tables---
coords.csv
loveLetter.txt
xy1.txt
xy_current.txt
summary.dbf
```

11.2 Specify Data Name and Type

Most of the `arcpy` listing methods have optional parameters. The purple box in Figure 11.1 shows the function signatures, which list the arguments in parentheses behind each method name with optional arguments in curly braces. Most list methods have two optional arguments for filtering names and types. For example, the `ListRasters` signature includes no required arguments, but it has two optional arguments, `wild_card` and `raster_type`, as shown here:

```
ListRasters({wild_card},{raster_type})
```

If no arguments are used, all raster datasets in the current `arcpy` workspace are listed. The optional arguments allow you to restrict the search to a subset of the rasters based on names and/or types. The `wild_card` argument is a string name which can use asterisks as wild cards. A *wild card* stands for a string (or substring) that can have any value. Any string of characters can be substituted in the position where the asterisk appears in the string. For example, `'*am'` could stand for strings such as 'spam', 'ham', and 'am', but it could not stand for 'hamper' and the wild card string `'*elev*'` could stand for strings like 'elevation', 'relevant', 'peak_elev', and 'elev'.

The `wild_card` argument allows you to specify a data name substring using the asterisk as a placeholder for any string. Then if data has a semantic naming schema, e.g., any rasters related to elevation, might contain the substring 'elev', you can use this to select a subset of files for processing. The asterisk can be used anywhere in the string. Example 11.2 demonstrates wild card usage. Can you predict the output from each of these?

Example 11.2a lists all the rasters in the workspace using `'*'` as the wild card. This has the same affect as passing no arguments. Example 11.2b uses `'elev*'` as the wild card parameter; it specifies a prefix, by placing an asterisk at the end of the string. To specify a suffix, use an asterisk at the beginning. If no asterisk is used in a string, it looks for that exact string. Though it can be done, you would usually not want to use the wild card without asterisks. Example 11.2c uses 'elev' as the wild card. This returns only the raster named 'elev'. If this is the intended outcome, the `arcpy.Exists` function would be a better choice for checking one specific data element.

Example 11.2a–c: List rasters in the 'C:/gispy/data/ch11/rastTester.gdb' workspace and use `wild_card` arguments.

```
# wildcards.py
# Purpose: Use a wild_card to selectively list files.

import arcpy
arcpy.env.workspace = 'C:/gispy/data/ch11/rastTester.gdb'

# a. Use '*' or empty parentheses to list ALL
#    the rasters in the workspace.
rasts = arcpy.ListRasters('*')
print 'a. All the rasters:'
print rasts
print

# b. List the rasters whose names START with 'elev'.
rasts = arcpy.ListRasters('elev*')
print 'b. elev* rasters:'
print rasts
print
```

```
# c. List a raster whose name is exactly 'elev'.
rasts = arcpy.ListRasters('elev')
print 'c. elev raster:'
print rasts
```

Printed output:

```
>>> a. All the rasters:
[u'elev', u'landcov', u'soilsid', u'getty_cover', u'landc197',
u'landuse', u'aspect', u'soils_kf', u'plus_ras', u'CoverMinus',
u'elev_srt', u'elev_sh', u'elev_ned', u'Int_rand1', u'Int_rand2',
u'landc196', u'TimesCOVER']

b. 'elev*' rasters:
[u'elev', u'elev_srt', u'elev_sh', u'elev_ned']

c. 'elev' raster:
[u'elev']
```

Many of the `arcpy` listing methods also have an optional argument for specifying a type, such as `feature_type`, `raster_type`, or `table_type`. The values for the type variable, are listed in the ArcGIS Resources site page for each method. For example, search the help for 'ListRasters (arcpy)' and view the 'Explanation' for the `raster_type` parameter in the parameter table. Valid raster types include 'BMP', 'GIF', 'IMG', and so forth. The raster types often match the file extension ('GRID' files are an exception; they have no extensions, since they are composite files made up of directories and supporting files).

To use the type parameter, put the type in quotation marks. The only tricky aspect to using the type argument is that, it is not the first optional argument. To use arguments that are not first in the code signature, you always have to specify the arguments that precede it; sometimes this means inserting a placeholder. For example, to list all the GIF type raster files in a directory, you don't want to restrict the file names, but you still have to use a placeholder for the `wild_card` argument. A string containing only an asterisk (`'*'`) can be used as a placeholder in this case. Example 11.3 demonstrates various arguments for the `ListRasters` method. Can you predict the output for each part?

11.3a lists all the rasters in the workspace. 11.3b finds all of the 'GIF' type raster files in the workspace. Example 11.3c omits the `wild_card` parameter to demonstrate that if the placeholder is omitted, what was intended to be the `raster_type` argument would be interpreted as a `wild_card`. There are no rasters named 'GIF', so the list is empty. Example 11.3d lists all the raster files whose names start with 'tree'. Example 11.3e uses `raster_type` together with the `wild_card` argument to narrow this list to only the 'GIF' type `'tree*'` rasters.

Examples 11.2 and 11.3 shows several ways to use `wild_card` and `raster_type` parameters with the `ListRasters` method; the other Category I methods follow the same patterns.

Example 11.3a–e: List rasters in the 'C:/gispy/data/ch11/' workspace using `wild_card` and `raster_type` arguments.

```
# rasterTypes.py
# Purpose: Use a wildcard to selectively list files.
import arcpy
arcpy.env.workspace = 'C:/gispy/data/ch11/'

# a. Use empty parenthesis to list ALL the rasters
#   in the current workspace.
rasts = arcpy.ListRasters()
print 'a. All the rasters:'
print rasts
print

# b. List ALL the GIF type rasters.
rasts = arcpy.ListRasters('*', 'GIF')
print 'b. GIF rasters:'
print rasts
print

# c. List the raster whose name is GIF
rasts = arcpy.ListRasters('GIF')
print 'c. raster named GIF:'
print rasts
print

# d. List the rasters whose names start with 'tree'.
rasts = arcpy.ListRasters('tree*')
print 'd. tree* rasters:'
print rasts
print

# e. List the rasters whose names start with 'tree' which are GIF
#    type files.
rasts = arcpy.ListRasters('tree*', 'GIF')
print 'e. tree* GIF type rasters:'
print rasts
print
```

Printed output:

```
>>> a. All the rasters:
[u'jack.jpg', u'minus_ras', u'tree.gif', u'tree.jpg', u'tree.png',
u'tree.tif', u'tree2.gif', u'tree2.jpg', u'window.jpg']

b. GIF rasters:
[u'tree.gif', u'tree2.gif']
```

```
c. raster named GIF:
[]

d. tree* rasters:
[u'tree.gif', u'tree.jpg', u'tree.png', u'tree.tif', u'tree2.gif',
u'tree2.jpg']

e. tree* GIF type rasters:
[u'tree.gif', u'tree2.gif']
```

11.3 List Fields

Unlike Category I methods, Category II methods require one argument, an input dataset—the items they are listing (such as fields) pertain to a dataset, not an entire workspace. These methods are slightly more complicated than Category I because they return lists of objects instead of string names. The `ListFields` method returns a list of `Field` objects, not string names of fields. For example, the following code gets the list of `Field` objects for input file 'park.shp':

```
>>> # List the Field objects for this dataset.
>>> fields = arcpy.ListFields('C:/gispy/data/ch11/park.shp')
>>> fields
[<Field object at 0x12efa5f0[0x1241ba88]>, <Field object at
0x12efab70[0x1241bf68]>, <Field object at 0x12efa8b0[0x29888f0]>,
<Field object at 0x12efa770[0x2988fe0]>]
```

Each item in this list is a `Field` object (the numbers indicate location in memory). A `Field` object can be thought of as a compound structure that contains multiple pieces of information about the field, that is, the field properties such as name, type, length, and so forth (Figure 11.2 lists more of these properties). Usually, we don't want to use the `Field` object, but rather the field name or some other property. To access these properties, you need to use `object.property` format. The following code prints the field names as seen in the attribute table:

```
>>> for fieldObject in fields:
...     print fieldObject.name
...
FID
Shape
COVER
RECNO
```

Other `Field` object properties can be accessed in the same manner, as shown in Example 11.4.

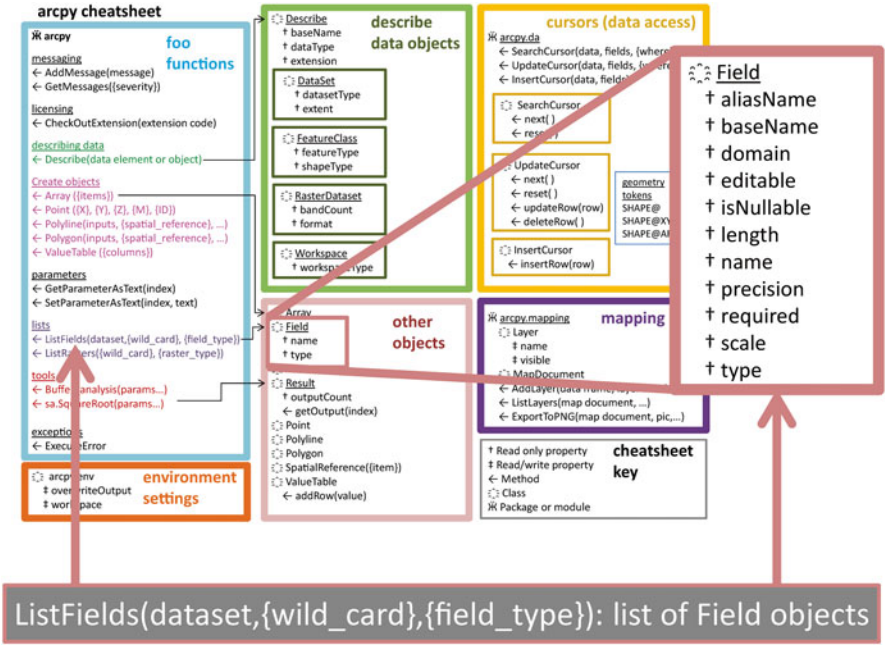


Figure 11.2 The `ListFields` method returns a list of `Field` objects. A `Field` object has the list of properties shown in the enlarged box on the right. These properties are read-only when they are returned by the `ListFields` method.

Example 11.4: List the `Field` object properties.

```
# listFields.py
# Purpose: List attribute table field properties.
import arcpy
arcpy.env.workspace = 'C:/gispy/data/ch11/'

fields = arcpy.ListFields('park.shp')
for fieldObject in fields:
    print 'Name: {0}'.format(fieldObject.name)
    print 'Length: {0}'.format(fieldObject.length)
    print 'Type: {0}'.format(fieldObject.type)
    print 'Editable: {0}'.format(fieldObject.editable)
    print 'Required: {0}\n'.format(fieldObject.required)
```

Printed output:

```
>>> Name: FID
Length: 4
Type: OID
Editable: False
Required: True
```

```
Name: Shape
Length: 0
Type: Geometry
Editable: True
Required: True

Name: COVER
Length: 5
Type: String
Editable: True
Required: False

Name: RECNO
Length: 11
Type: Double
Editable: True
Required: False
```

The ListFields method optionally accepts `wild_card` and `field_type` arguments in addition to the required dataset argument. Using these looks very similar to using the optional arguments for Category I methods, except that the one required argument must come first. The following code specifies the `field_type` as `'Double'` (and uses a placeholder for the `wild_card` parameter, so that the name is unrestricted):

```
>>> parkData = 'C:/gispy/data/ch11/park.shp'
>>> fields2 = arcpy.ListFields(parkData,'*', 'Double')
>>> # The list length shows how many Field objects were returned.
>>> len(fields2)
1
>>> fields2[0].name
'RECNO'
```

> **Note** The `ListFields` method returns a list of `Field` objects, not string names. Use the `object.property` format to access the name property.

11.4 Administrative Lists

GIS scripts use Category I and II listing methods most frequently. Category III methods, like Category I methods return lists of names. Category III methods provide access to ArcGIS software and computer system information for administrative purposes. For example, `ListEnvironments` returns a list of the

environment settings, `ListInstallations` returns a list of the type of ArcGIS software installed on the system (server, desktop, engine), `ListPrinterNames` returns a list of printer names available to the system, `ListTools` returns a list of the ArcGIS tools, and `ListToolboxes` returns a list of the built-in ArcGIS toolbox names and aliases with the format 'toolbox_name(toolbox_alias)'. The `arcpy` package has additional listing methods for web mapping administration.

11.5 Batch Geoprocess Lists of Data

The `arcpy` listing methods provide lists of existing data files to batch process. With these methods, we can automatically perform any kind of geospatial processing on a batches of files by iterating over the list with a FOR-loop and tucking the geoprocessing inside the loop. The following code that deletes the 'GRID' datasets with the prefix `'out'`:

```
>>> # List all coverage, geodatabase, TIN, Raster, and CAD datasets.
>>> datasets = arcpy.ListDatasets('out*', 'GRID')
>>> for data in datasets:
...     arcpy.Delete_management(data)
```

Calling the delete tool only requires one argument—the file to be deleted. For tools that create output data, the code needs to update the output file name inside the loop; Otherwise, the looping won't generate multiple output files. It will just overwrite the same output file multiple times. The value of the iterating variable (the variable between the `for` and `in` keywords in the `FOR`-loop) changes during each iteration. Incorporating this value into the output variable name creates unique names.

Example 11.5: Batch buffer the feature class files in 'C:/gispy/data/ch11/'.

```
# batchBuffer.py
# Purpose: Buffer each feature class in the workspace.

import arcpy, os
arcpy.env.overwriteOutput = True
arcpy.env.workspace = 'C:/gispy/data/ch11/'

# GET a list of feature classes
fcs = arcpy.ListFeatureClasses()
for fc in fcs:
    # SET the output variable.
    fcBuffer = os.path.splitext(fc)[0] + 'Buffer.shp'
    # Call Buffer (Analysis) tool
    arcpy.Buffer_analysis(fc, fcBuffer, '1 mile')
    print '{0} created.'.format(fcBuffer)
```

Table 11.1 Batch processing pseudocode format.

```
GET a list of data elements (fields, files, workspaces, etc.)
FOR each item in the list
    SET output file name
    CALL geoprocessing
ENDFOR
```

Example 11.5 buffers each feature class in the workspace. Notice that the output file name is set *inside* the loop and *before* the geoprocessing. The pseudocode template for batch geoprocessing is shown in Table 11.1.

Example 11.5 places the output in the same workspace as the input. To use a different output workspace, the file needs to do two things differently. First, the script needs to ensure that the output workspace exists by creating it, if necessary. Second, the full path file name of the output needs to be used within geoprocessing calls; otherwise, the path of `arcpy.env.workspace` will be assumed. Example 11.6 differs from Example 11.5 by using 'C:/gispy/data/ch11' for the input directory and 'C:/gispy/scratch/buffers' for the buffer output directory. These paths are hard-coded in the example, but if they are passed in by the user, we can't be sure if the output directory ends in a slash. The `os.path.join` method handles either case by inserting a slash only if needed.

Example 11.6: Batch buffer files in 'C:/gispy/data/ch11/' and place them in the 'C:/gispy/scratch/buffers' directory.

```
# batchBufferv2.py
# Purpose: Buffer each feature class in the workspace and
#    place output in a different workspace.
import arcpy, os
arcpy.env.overwriteOutput = True
# SET workspaces
arcpy.env.workspace = 'C:/gispy/data/ch11/'
outDir = 'C:/gispy/scratch/buffers'
if not os.path.exists(outDir):
    os.mkdir(outDir)
#GET a list of feature classes
fcs = arcpy.ListFeatureClasses()
for fc in fcs:
    # SET the output variable
    outName = os.path.splitext(fc)[0] + '_buffer.shp'
    fcBuffer = os.path.join(outDir, outName)
    # Call buffer tool
    arcpy.Buffer_analysis(fc, fcBuffer, '1 mile')
    print '{0} created in {1}.'.format(fcBuffer, outDir)
```

So far, we have seen loops which call a single geoprocessing tool and we have seen how to insert an existing block of code into a batch processing loop. Adding multiple geoprocessing steps inside a loop is very similar to what we have already done. The only additional concern may be determining which variables should be set inside the loop and which should be set prior to the loop. Example 11.7 lists data, performs multiple steps within a loop, and then performs an operation on the combined output.

The script lists `'stations*'` dBASE files, which are tables of (x,y) point locations and converts each one to a point feature layer. Then it connects the vertices of the feature layer with lines (This works like a 'connect the dots' game). When the looping is complete, the intersection of all the new line feature classes is computed. The result is a point file 'hubs.shp'. The polyline feature class names (`lineFile`) are set inside the loop because they need to be unique. The temporary point layer is not stored after each iteration, so that name can be reused. The `tempPoints` variable is set just once (before the loop). Make XY Event Layer and Points To Line (Data Management) tools are called once for each table, so they are inside the loop. The Intersect (Analysis) tool is called once outside the loop since that is a collective action on the entire assembled loop output.

Example 11.7

```
# tableLister.py
# Purpose: Create shapefiles from 'stations*' xy tables,
#    connect the points, and then find then
#    intersection of the lines.
# Usage:    workspace_containing_tables
# Example: C:/gispy/data/ch11/trains
import arcpy, os, sys
arcpy.env.workspace = sys.argv[1]
arcpy.env.overwriteOutput = True
tables = arcpy.ListTables('stations*', 'dBASE')

tempPoints = 'temporaryPointLayer'

for table in tables:
    # SET the output variable.
    lineFile = os.path.splitext(table)[0] + 'Line'
    # CALL geoprocessing tools.
    arcpy.MakeXYEventLayer_management(table, 'POINT_X',
                                      'POINT_Y', tempPoints)
    arcpy.PointsToLine_management(tempPoints, lineFile)
    print '\t{0}/{1} created.'.format(arcpy.env.workspace, lineFile)

# GET the list of lines and intersect the lines.
lineList = arcpy.ListFeatureClasses('stations*Line*')
arcpy.Intersect_analysis(lineList,'hubs','#', '0.5 mile', 'POINT')
print '{0}/hubs created.'.format(arcpy.env.workspace)
```

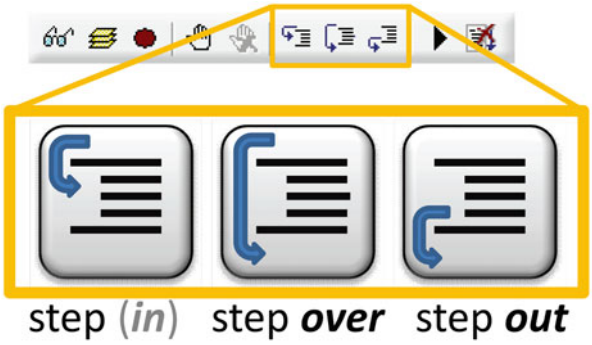
11.6 Debug: Step Through Code

Now that you have learned about block code structures for decision-making and looping and you can write more complicated scripts, you'll be able to create your own bugs! *Bugs* are errors or flaws in the code. *Debugging* is the process of locating and removing errors. Now is a good time to adopt some basic debugging techniques. We'll go over two debugging tips now—stepping through code and watching expressions.

First, you can use the debugger to *step through* the code, meaning you can run each line, advancing the cursor one statement at a time. This will help you understand the flow of the code. That is, this will reveal which statements are executed (in the case of branching) and the number of times each statement is executed (in the case of looping). There are three 'Step' buttons. Most of the time, we only use the one in the middle—the 'Step Over' button. Instructions for PythonWin are given, but any good Python IDE (such as PyScripter) will have equivalent buttons for stepping through code. Work through the following example to learn how to use the 'Step' buttons.

1. In PythonWin, make the debugging toolbar visible (View>Toolbars>Debugging). It looks like this:

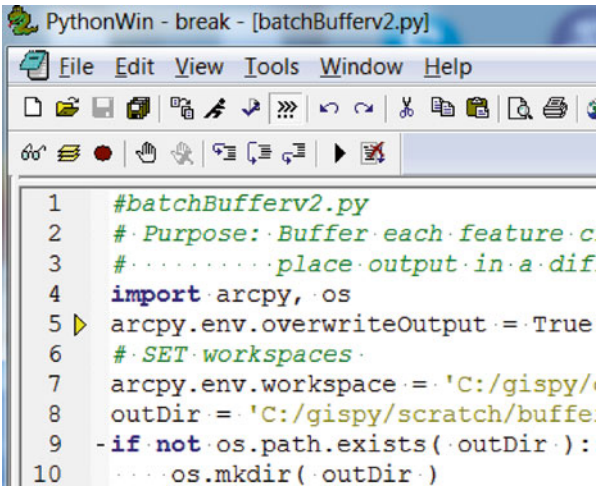
2. Mouse slowly over the three 'Step' buttons to see their labels:



3. Open 'batchBufferv2.py'.
4. To start stepping through the code, click the 'Step Over' button  once.
5. Observe the position of the arrow in the margin. This tracks your location in the script.

   Specifically, the arrow points to the next line of code that has not yet been executed. The next time you click the 'Step Over' button, this line of code will

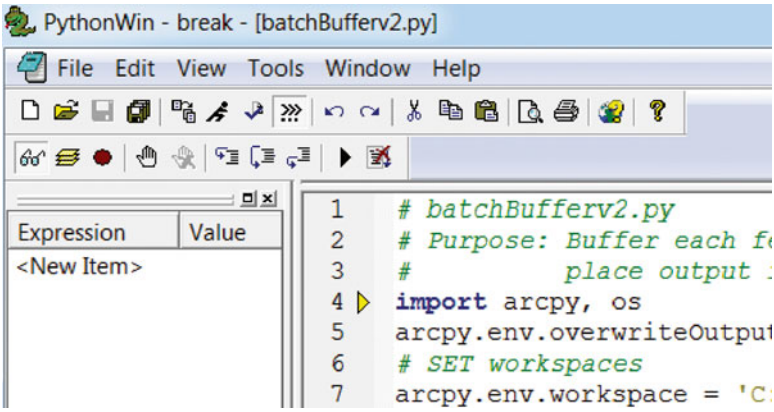
be executed. When we started the debugging session, the arrow jumped immediately to line 4, because comment lines are not executed.



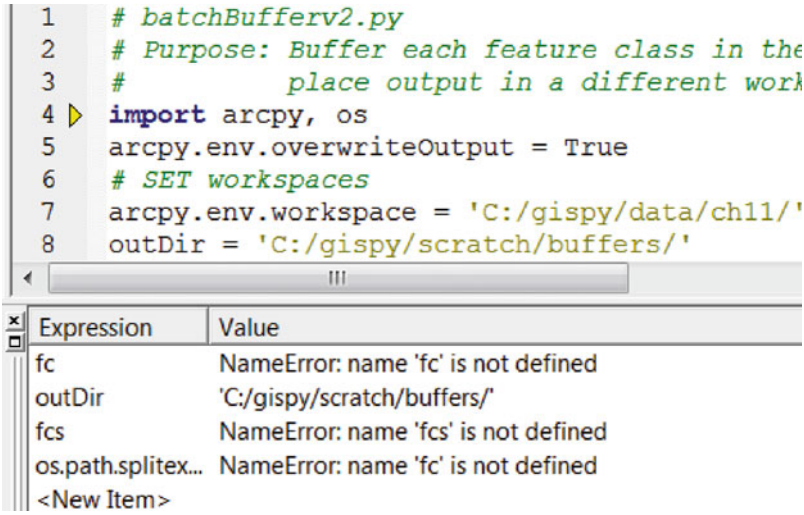
6. Click the 'Step Over' button  a second time. If you haven't yet imported `arcpy` in this PythonWin session, this step could take a few moments, after which the arrow will point at line 5. This means line 4 has been executed.
7. Now this time, click the 'Step (in)' button  instead of the 'Step over' button. This steps inside the `arcpy.env.workspace` statement. A script named '_base.py' opens and the cursor is inside an arcpy function because you stepped into the middle of this `arcpy` script. You should never alter any of these files or any of the built-in Python modules, so if you step into one of these you should step back out to your own script.
8. Click the 'Step out' button until you return to the 'batchBufferv2.py'. Sometimes you may need to click the 'Step out' button several times to return to your main script.
9. Now step over again until you reach line 12. The arrow jumps from line 9 to line 12 if the output directory has already been created.
10. Stop the debugging session using the 'Close' button  on the right end of the debugging toolbar. This doesn't close the script. It only stops the debugging session. The arrow disappears and the feedback bar reports that the script returned exit code 0, meaning a normal exit.

The second tip we'll go over now is to use the 'Watch' window to watch the variable values update as you step through the script.

1. Start stepping through the code by clicking the 'Step over' button.
2. Open the Watch window, by clicking the 'Watch' button  on the left end of the debugging toolbar. The window has two columns, 'Expression' and 'Value'. Initially, the Expression column says 'New Item' as shown here:



3. Script variables can be entered into the 'Expression' column and their current value is displayed. Double-click on '<New Item>' and type 'fc' and then click 'Enter'. Each time you add a new expression, you need to double-click on <New Item>. Enter `outDir` and `fcs` and `os.path.splitext(fc)[0]`. Optionally, you can move the Watch window to the bottom of the IDE (Shift + left-click-and-drag on the docking bar).



4. `fc` and `fcs` are not yet defined. These values will update as you step through the script. Keep an eye on the Watch window and step over each line until you reach line 12. The arrow jumps from line 9 to line 12 since the previous run created the 'buffer' directory.
5. Step again watching the value of `fcs` become defined as a list of feature classes after line 12 executes.
6. Step again and see the value of `fc` become defined as a feature class name after line 13 executes.

```
12   fcs = arcpy.ListFeatureClasses()
13  -for fc in fcs:
14       # SET the output variable
15       outName = os.path.splitext(fc)[0] + '_buf
```

Expression	Value
fc	u'data1.shp'
outDir	'C:/gispy/scratch/buffers/'
fcs	[u'data1.shp', u'park.shp', u'USA.shp']
os.path.splitex...	u'data1'

7. Step over lines 14–18 and watch how the arrow moves back up to line 13 and then the value of `fc` is updated. You can also watch your output being created one by one in the 'buffer' directory.
8. Step through the loop until the script completes or stop the debugging session using the 'Close' button 

   With these examples, you have practiced two debugging techniques:

- **Stepping**: Step for line-by-line inspection of the code. 'Step Over' to start execution of a script.
- **Watching**: The watch window allows you to view the values of variables as you step through the code. Double click <New Item> and replace it with a variable name, such as `x`. Before the variable is defined, the value will appear as "Name Error: name 'x' is not defined". This value is updated whenever a new value is assigned to the variable.

Stepping through code and watching expressions helps programmers become familiar with how Python runs code and how the debugger works. Even advanced programmers often need to slow down to closely inspect the code. We have only introduced a few of the buttons on the debugging toolbar. The remaining debugging buttons will be discussed in Chapter 13.

11.7 Key Terms

The `ListFeatureClasses`, `ListRaster`, `ListTables`, and `ListFields` methods
Asterisks as wild cards
The `arcpy Field` object
Step though code
Step, step over, step out buttons
Watch window

11.8 Exercises

1. **rasterNames.py** Write a script that uses an `arcpy` listing method to get a list of all the JPG type rasters in the input workspace. Then the script should join the list into a semicolon delimited string and print that string. Use the Python string method named `join` to create the semicolon delimitated string. Print the string. The script should take one argument, the workspace.

   Example input: C:/gispy/data/ch11/pics

   Example output:
   ```
   istanbul.jpg;istanbul2.jpg;marbleRoad.jpg;spice_market.jpg;stage.jpg
   ```

2. **batchCount.py** Write a script that uses an `arcpy` listing method twice to get a list of the Point and Polygon type feature classes in a workspace whose name contains an input substring. Then use a `FOR`-loop to print the record count result derived from calling the Get Count (Data Management) tool. The script should take two arguments, the input workspace and the desired substring.

   Example input1: C:/gispy/data/ch11/tester.gdb oo

   Example output1:
   ```
   schools has 8 entries.
   smallWoods2 has 55 entries.
   ```

   Example input2: C:/gispy/data/ch11/tester.gdb ne

   Example output2:
   ```
   workzones has 7 entries.
   ```

3. **batchPostOffice.py** For each point feature class in the input workspace whose name contains the string ‘data’, generate the postal service districts around each point by calling the Create Thiessen Polygons (Analysis) tool. Place the output in a user specified directory and append `'Postal'` to the output names. The script should take two arguments, the input and output workspaces.

   Example input: C:/gispy/data/ch11/tester.gdb C:/gispy/scratch

   Example output:
   ```
   C:/gispy/scratch/data1Postal.shp created.
   C:/gispy/scratch/data2Postal.shp created.
   C:/gispy/scratch/data3Postal.shp created.
   C:/gispy/scratch/ptdata4Postal.shp created.
   ```

4. **batchSimplifyPoly.py** Write a script that uses the Simplify Polygon (Cartography) tool to simplify all the polygon feature classes in the input work-

space. Use the POINT_REMOVE algorithm for polygon simplification and use a degree of simplification tolerance of 50. Place the output in a user specified directory and append 'Simp' to the output names. (An accompanying '*_Pnt.shp is generated for each input file as well). The script should take two arguments, the input and output workspaces.

Example input: C:/gispy/data/ch11/tester.gdb C:/gispy/scratch

Example output:

```
C:/gispy/scratch/simp/parkSimp.shp created.
C:/gispy/scratch/regions2Simp.shp created.
C:/gispy/scratch/simp/workzonesSimp.shp created.
C:/gispy/scratch/regions1Simp.shp created.
C:/gispy/scratch/smallWoods2Simp.shp created.
```

5. **batchPoly2Line.py** Batch convert the polygon feature classes in the input workspace to line files using the Polygon to Line (Data Management) tool. Have the script place the output in a user specified output directory. The script should create this directory if it doesn't already exist. Append 'Lines' to the output file name. The script should take two arguments, the input and output workspaces.

   Example input: C:/gispy/data/ch11/tester.gdb C:/gispy/scratch/lineOut

   Example output:

   ```
   C:/gispy/scratch/lineOut/parkLine.shp created.
   C:/gispy/scratch/lineOut/regions2Line.shp created.
   C:/gispy/scratch/lineOut/workzonesLine.shp created.
   C:/gispy/scratch/lineOut/regions1Line.shp created.
   C:/gispy/scratch/lineOut/smallWoods2Line.shp created.
   ```

6. **quickExport.py** Write a script that uses an `arcpy` listing method to get a list of all the feature classes in the input workspace and then calls the Quick Export (Data Interoperability) tool to export these files to comma separated value files in output directory, 'C:/gispy/scratch'. Modify the example, `"GIF,c:/data/"` for the output parameter. This example exports the input to GIF image format and places the output in 'C:/data' directory. Look at the ArcGIS Resources site for the Quick Export tool and observe how the parameter table indicates that the tool can take a Python list as the first argument. Call the function only once (without using a loop). The script should take one argument, the input workspace. When the tool call is complete, the script should again call an `arcpy` listing method to get a list the tables in the output directory and print the list.

   Example input: C:/gispy/data/ch11/

Example output:

```
C:/gispy/scratch contains:
[u'data1.csv', u'park.csv']
```

7. **tableToDBASE.py** Write a script that uses an `arcpy` listing method to get a list of the tables in the input workspace and then calls the Table to dBASE (Conversion) tool to convert the tables into dBASE format and place them in the given output directory. (Files that are already dBASE format will simply be copied). Look at the ArcGIS Resources site for the Table to dBASE tool and observe how the parameter table indicates that the tool can take a Python list as the first argument. Call the function only once (without using a loop). The script itself should take two arguments, the input and output workspaces. When the tool call is complete, the script should use an `arcpy` listing method to get a list the tables in the output directory and print the list.

   Example input: C:/gispy/data/ch11/ C:/gispy/scratch

   Example output:

```
C:/gispy/scratch contains:
[u'coords.dbf', u'loveLetter.dbf', u'summary.dbf',
u'xy1.dbf', u'xy_current.dbf']
```

8. **batchFieldNames.py** Write a script that lists the rasters in a workspace whose name contain a specified substring. Also list the field names for each raster. The script should take two arguments, the workspace and the desired substring. Indent the field names using a tab (`'\t'`) to match the example output.

   Example input: ‘C:/gispy/data/ch11/rastTester.gdb’ ‘COVER’

   Example output:

```
>>> getty_cover
    OBJECTID
    VALUE
    COUNT

CoverMinus
    OBJECTID
    VALUE
    COUNT
    Location

TimesCOVER
    OBJECTID
    Value
    Count blades
```

9. **freqFieldLoop.py** The Frequency (Analysis) tool creates a dBASE table with frequency results for a given field. Practice using nested looping; Write a script that calls a Frequency (Analysis) tool on all the `'string'` type fields in the shapefiles in a given directory. Allow the user to input the workspace name. Use the field name and 'freq.dbf' as part of the frequency table output name, as shown in the examples below.

   Example input: C:/gispy/data/ch11 C:/gispy/scratch

   Example output:

```
>>> C:/gispy/scratch/park_COVERfreq.dbf created.
C:/gispy/scratch/USA_STTE_NAMEfreq.dbf created.
C:/gispy/scratch/USA_SUB_REGIONfreq.dbf created.
C:/gispy/scratch/USA_STATE_ABBRfreq.dbf created.
```

   Frequency analysis yields a table with the count for the occurrence of each unique value. The file called 'C:/gispy/scratch/park_COVERfreq.dbf' looks like this:

OID	Frequency	Cover
0	151	orch
1	62	other
2	213	woods

   Though the help lists the first parameter, 'in_table' as a 'Table View or Raster layer', this parameter can be specified as the name of a shapefile, as in the following example:

```
arcpy.Frequency_analysis('park.shp', 'out.dbf', 'COVER')
```

10. **batchRatioField.py** Write a script to perform sequential geoprocessing steps inside a loop as follows. Copy each polygon type feature class in an input directory to an output directory, using the Copy (Data Management) tool. Then add a new field named 'ratio', using the Add Field (Data Management) tool, and calculate the field as follows. Use the Calculate Field (Data Management) tool with a Python expression that calculates the ratio of the shape area to the 'rating' field value. E.g., for area = 60000 and rating = 10, ratio = 6000. Assume the 'rating' field exists in each feature class.

    Example input: C:/gispy/data/ch11/trains C:/gispy/scratch

    Example output:

```
regions1.shp copied to C:/gispy/scratch/regions1.shp
New field ratio added to regions1.shp and calculated as
!shape.area!/!rating!
```

```
regions2.shp copied to C:/gispy/scratch/regions2.shp
New field ratio added to regions2.shp and calculated as
!shape.area!/!rating!
```

11. **batchPoint2Segment.py** Write a script that performs sequential geoprocessing steps inside a loop to create a line type feature class in an output directory for each point type feature class in an input directory. First, call the Points to Line (Data Management) tool to create a line feature class with one line feature that connects all the points. Use names such that the output from this tool creates 's1Line.shp' for a point file named 's1.shp'. Then call the Split Line (Data Management) tool to split the line at the vertices. Use names such that the output from this tool creates 's1Segment.shp' for a point file named 's1.shp'. Finally, delete the intermediate line output files using the Delete (Data Management) tool, such as 's1Line.shp' so that only the line segment files remain.

    Example input: C:/gispy/data/ch11/trains C:/gispy/scratch

    Example output:

    ```
    C:/gispy/scratch/s1Line.shp created.
    C:/gispy/scratch/s1Segment.shp created.
    C:/gispy/scratch/s1Line.shp deleted.
    C:/gispy/scratch/s2Line.shp created.
    C:/gispy/scratch/s2Segment.shp created.
    C:/gispy/scratch/s2Line.shp deleted.
    ```