Chapter 10
Repetition: Looping for Geoprocessing

Abstract Advances in technology are enabling the collection and storage of massive amounts of GIS data. To learn from this data we need to conduct analysis efficiently. Batch processing is a powerful scripting capability, saving time by automating repetitive tasks. Chapter 8 discussed the three basic workflow structures (sequences, decision-making, and repetition) in terms of pseudocode and Chapter 9 presented the Python syntax for decision-making and describing GIS data. This chapter focuses on Python repetition structures and looping syntax, Python `FOR`-loops and `WHILE`-loops, looping with the range function for geoprocessing, nested looping, and listing directory contents. Then the chapter concludes with a tip about debugging whitespace glitches.

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

- Implement `WHILE`-loops and `FOR`-loops in Python.
- Identify the three key iterating variable components in a `WHILE`-loop.
- Explain how Python `FOR`-loops work.
- Repair infinite loops.
- Call a geoprocessing tool in a `WHILE`-loop to vary a numerical parameter.
- Automatically generate numerical lists.
- Loop with the `range` function.
- Branch and loop within loops.
- List the files in a directory.
- Geoprocess each file in a list.
- Correct indentation errors.

10.1 Looping Syntax

Python has two structures for performing repetition, `WHILE`-loops and `FOR`-loops, which use the Python keywords `while` and `for`. They provide two slightly different approaches. `FOR`-loops are used more often in geoprocessing scripts; however, both looping techniques are used, so it is helpful to be familiar with both. We'll start

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_10

with WHILE-loops which provide an easier introduction to looping because Python WHILE-loops have the looping mechanisms exposed; whereas, the workings of Python FOR-loops are less explicit. Many programming languages share a similar WHILE-loop structure with Python; whereas, the structure of Python FOR-loops is less common; both will seem easy, once you gain some practice.

10.1.1 WHILE-Loops

The code in Example 10.1 uses a Python WHILE-loop and line numbers are shown on the left. Can you predict what this script will print?

Example 10.1

```
1  # simpleWhileLoop.py
2  x = 1
3  while x <= 5:
4      print x
5      x = x + 1
6  print 'I'm done!'
```

A WHILE-loop checks a Boolean expression placed on the same line as the `while` keyword. In Example 10.1, it checks if `x<=5`. The 'condition', like the ones used in decision-making statements, evaluates to either be true or false and the statements in the indented code block that follows (such as, lines 4 and 5) are only performed so long as the condition is true. After it runs the last indented step, it evaluates the value of the condition again (true or false?). If the condition is true, it repeats the indented steps from the top again (line 4, then line 5). If the condition is false, it moves to the first dedented line of code after the WHILE-loop (line 6). We call the indented lines of code the 'WHILE-loop code block'. The WHILE-loop repeatedly runs this code block as long as the test condition is still true. If the condition is never true in the first place, the indented code block will never be executed. Run 'simpleWhileLoop.py' and the results should look like this:

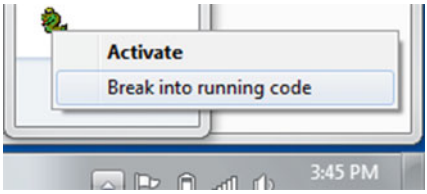
```
>>> 1
2
3
4
5
I'm done!
```

`x` is printed five times and `"I'm done!"` is only printed once, since this print statement is outside the WHILE-loop code block.

WHILE-loops rely on a test condition. If that condition never becomes false, this code can run in an infinite loop. The Python interpreter has to be interrupted in a special way if this happens. Comment out line 5 in Example 10.1 by placing a hash sign (#) at the beginning of the line. Predict what will be printed before you run it. Then follow the 'How to interrupt running code' instructions.

How to Interrupt Running Code

1. For PythonWin,
   (a) Right-click on the PythonWin icon that represents your running process, in the lower right corner of the screen.
   (b) Click "Break into running code". You may need to select this multiple times before the code actually stops running.



   (c) Once you successfully stop the running code, a 'KeyboardInterrupt' traceback is printed in the Interactive Window.
2. In PyScripter,
   (a) Click CTRL+F2
   (b) Click 'Yes' on the box that says 'The interpreter is busy. Are you sure you want to terminate?'

Removing line 5 from Example 10.1 creates an 'infinite loop' because the test condition is checking the value of x. When the value of x is not modified, the test condition never becomes false. If x is always 1, x<=5 is always true! WHILE-loop test conditions usually involve a variable that needs to be initialized outside the loop and updated inside the loop. Usually the update should occur after all the other WHILE-loop block statements. The WHILE-loop generally has this format:

```
initialize test condition variable    <-- component #1
while condition:                      <-- component #2
    code statement(s)
    update test condition variable.   <-- component #3
```

If any of the three labeled components—initializing, checking, or updating the test condition variable—are missing, the loop will not work as expected.

Notice the syntax elements, the colon and indentation. All Python code blocks—structures containing multiple lines of related code—require the colon and indentation. This includes conditional code blocks, `WHILE` and `FOR`-loop code blocks and other structures such as functions and classes, which are discussed in upcoming chapters. The colon always indicates that a block of related code will follow. A missing colon will cause a syntax error. A syntax error will also occur if the code block is empty or if the first line after the colon is not indented. As long as you type the colon, indentation is easy, because IDEs automatically indent the next line when you press the 'Enter' key to move to the next line.

Example 10.2 shows a `WHILE`-loop geoprocessing example with the row numbers on the left. This script generates three rasters containing randomized cell values. The looping variable, `n`, is initialized, checked, and updated on lines 10, 11, and 15. `n` is used to create a unique name for each output raster. The output data name is set at the beginning of the loop code block on line 12. The iterating variable, `n`, starts with the value 1, then becomes 2, and finally 3. Line 12 creates the names `out1`, `out2`, and `out3`. Notice that the general setup steps, importing modules, setting the environment properties, and checking out the Spatial Analyst Extension, are outside of the loop, before the repetition starts. These statements are only executed once; there is no need to repeat them. The repeated geoprocessing steps—naming, creating, and saving the output raster—are placed inside the loop where they belong.

Example 10.2: Use a `WHILE`-loop to create 3 rasters containing random values with a normal distribution.

```
1  # normalRastsLoop.py
2  # Purpose: Create 3 raster containing random values.
3
4  import arcpy
5  arcpy.env.workspace = 'C:/gispy/data/ch10'
7  arcpy.env.overwriteOutput = True
8  arcpy.CheckOutExtension('Spatial')
9
10 n = 1
11 while n < 4:
12     outputName = 'out{0}'.format(n)
13     tempRast = arcpy.sa.CreateNormalRaster()
14     tempRast.save(outputName)
15     n = n + 1
16 del tempRast
17 print 'Normal raster creation complete.'
```

10.1.2 FOR-Loops

FOR-loops are essential for processing data in Python. Suppose you have a list of files. You can perform batch processing by using a list to control the loop. In Example 10.3, 'simpleForLoop.py' prints the names of each file in a list. Upcoming examples will substitute geoprocessing for printing and the arcpy module has functions for getting lists of ArcGIS data (discussed in Chapter 11), so that the data list won't need to be hard-coded.

Example 10.3: Basic FOR-loop.

```
1 # simpleForLoop.py
2 dFiles = ['data1.shp', 'data2.shp','data3.shp','data4.shp']
3 for currentFile in dFiles:
4     print currentFile
5 print 'I'm done!'
```

Printed output:

```
data1.shp
data2.shp
data3.shp
data4.shp
I'm done!
```

When you first see the Python FOR-loop, it may appear mysterious because the mechanisms that make the looping occur are not explicit. The variable placed between the for and in keywords automatically *iterates* over the items in the list that follows the in keyword. In other words, it assumes the value of each item successively. To understand what's happening in Example 10.3, it may help to picture an arrow labeled currentFile moving through the list, as shown in Figure 10.1. currentFile iterates over the items in the dFiles list. It assumes the value of each item successively. Each time it assumes a new value, the indented code block is executed (the indented code block only consists of line 4 in Example 10.3). When there are no remaining unused values in the list, the looping is complete and the next dedented line of code is executed (line 5 in Example 10.3).

To summarize the discussion of Example 10.3, the Python FOR-loop structure repeats the indented code block as many times as there are items in the list. Example 10.3 executed print currentFile four times. The Python keywords for and in are paired together. Whenever you use the for keyword, you must use the in keyword along with it. The syntax for Python FOR-loops is as follows:

```
for iteratingVar in sequence:
    code statement(s)
```

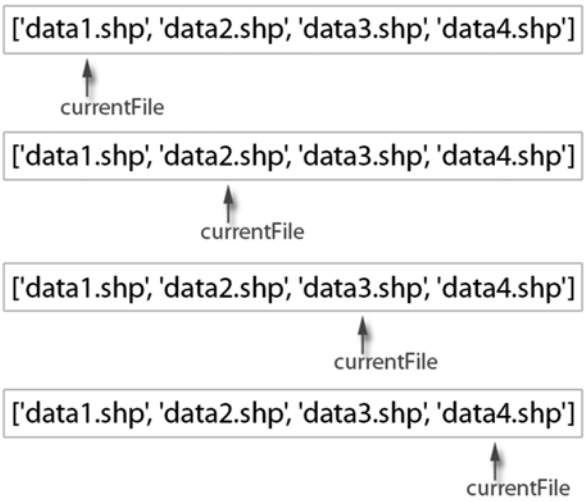


Figure 10.1 Python automatically updates the variable, `currentFile` in Example 10.3, to each item in the list.

As with the `WHILE`-loop, the colon and indented code block are required syntax. The *iterating variable*, `iteratingVar`, is assigned the successive values in the sequence. Python data types that consist of a collection of items are *sequence data types*. Examples include Python strings, tuples, and lists. Strings are made of characters; Tuples and lists are made of comma separated items. The sequence data object we use most frequently is a Python list, though other types such as Python strings and tuples can also be placed after the `in` keyword.

Example 10.3 used a variable named `currentFile` as the iterating variable, but there's nothing special about this name. Though you can use any variable name, most often you don't want to use one that is already being used for some other purpose because this will change that variable's value. Any variable name you place between the `for` and `in` keywords automatically gets assigned each value in the list as the loop iterates. Suppose you're fond of elephants. The following code shows that you can name your looping variable `elephant` (though we usually select a term that is more indicative of the items in the list):

```
>>> dFiles = ['data1.shp', 'data2.shp','data3.shp','data4.shp']
>>> for elephant in dFiles:
...     print elephant
...
data1.shp
data2.shp
data3.shp
data4.shp
```

Notice the triple dots in the PythonWin Interactive Window code sample above. These dots are used in the Interactive Window to reinforce that the current line of code is part of a code block. They appear automatically when a code block (following a colon) is entered in the Interactive Window. In fact, when you enter the last line of this code block, you must click the 'Enter' key again to escape from the loop code block. Only then will the code be executed.

Example 10.4 shows a `FOR`-loop geoprocessing example with the row numbers on the left. This script connects the points in each input file to form polyline output files. When a script creates output inside a loop, the output name needs to be unique at each iteration; otherwise, a file with the same name will be written over each time and the output in the end will be a single file. In order to build a unique output name during each iteration of the loop, you need to use something that is being updated during each iteration. In Example 10.4 the `currentFile` variable is being updated at each iteration, so we need to use this in the output file name.

> **Note** In general, when geoprocessing a set of files within a loop, the iterating variable is used to build the output file name, since this variable is changing each time.

In Example 10.4, the Points To Line (Data Management) tool is called on line 14. The other lines inside the code block are concerned with creating the unique output file names. To do so, they use the name of the input file as part of the name for the output file. Line 11 gets the base name of the input (`data1`, `data2`, …). Line 13 adds 'Line.shp' to create names like 'data1Line.shp', 'data2Line.shp', and so forth. This is a basic example of geoprocessing within a loop. Environment variables only need to be set once, so they are set prior to the loop (just like in Example 10.2). In Example 10.4, a list has to be initialized prior to the loop, because this is used to create the loop. Next comes the loop statement followed by the indented code block. This code sets up the output variable name and then performs geoprocessing.

Example 10.4: Use a `FOR`-loop to perform a point to line conversion on a hard-coded file list.

```
1  # point2Line.py
2  # Purpose: Create a set of line features from a
3  #          set of point features in a list.
4  import arcpy
5  arcpy.env.workspace = 'C:/gispy/data/ch10'
6  arcpy.env.overwriteOutput = True
7
8  theFiles = ['data1.shp', 'data2.shp','data3.shp','data4.shp']
9  for currentFile in theFiles:
10     # Remove file extension from the current name.
```

```
11      baseName = currentFile[:-4]
12      # Create unique output name. E.g., 'data1Line.shp'.
13      outName = baseName + 'Line.shp'
14      arcpy.PointsToLine_management(currentFile,outName)
15      print '{0} created.'.format(outName)
```

Examples 10.1–10.4, show number-based iteration and list-based iteration. There are other uses of these structures, but these are the most common. The examples used `WHILE`-loops for number-based iteration and `FOR`-loops for list-based iteration. In fact, a `FOR`-loop can also perform number-based iteration by using a numeric list. Rather than hard-coding a list of numbers, the built-in `range` function, which returns lists of numbers, can be used to automatically generate numeric lists. For example, the following code returns a Python list containing the integers from 1 to 5:

```
>>> range(1,6)
[1, 2, 3, 4, 5]
```

You can either set a variable to the return value or embed the `range` function directly in the `FOR`-loop statement itself, as follows.

```
>>> for i in range(1,6):
...     print i
...
1
2
3
4
5
```

Example 10.5 employs the `range` function in a `FOR`-loop which buffers data. The numbers one through five are used to vary the buffer distance from one to five miles. The numbers are also used to create output names. The iterating variable forms the unique portion of the output names.

Example 10.5: `FOR`-loop using the `range` function to update the linear unit.

```
# bufferLoopRange.py
# Purpose: Buffer a park varying buffer distances from 1 to 5 miles.

import arcpy
arcpy.env.workspace = 'C:/gispy/data/ch10'
arcpy.env.overwriteOutput = True
inName = 'park.shp'
for num in range(1, 6):
    # Set buffer distance based on num ('1 miles', '2 miles', ...)
    distance = '{0} miles'.format(num)
    # Set output name ('buffer1.shp', 'buffer2.shp', ...)
```

```
outName = 'buffer{0}.shp'.format(num)
arcpy.Buffer_analysis(inName, outName, distance)
print '{0} created.'.format(outName)
```

10.2 Nested Code Blocks

Scripts often need to use nested codes structures—such as conditional blocks inside FOR-loops (Example 10.6) or FOR-loops inside FOR-loops (Example 10.7). Notice how the indentation steps inward as the nesting occurs. In Example 10.6 the if, elif, and else keywords are aligned at one level and the print keywords contained within these blocks are aligned at the next level. The 'emotaLoop.py' script gets its name from the emoticons it prints. This script loops through each file name in the list and determines which face to print, based on the file extension. The string endswith method, which returns True or False, is used to check the file extension.

Example 10.6

```
# emotaLoop.py
# Purpose: Nest conditions inside a loop to print an emoticon
#          for each file name.

theFiles = ['crops.csv', 'data1.shp', 'rast','xy1.txt']

for f in theFiles:
    if f.endswith('.shp'):
        print '    ;]    ' + f
    elif f.endswith('.txt'):
        print '    :(    ' + f
    else:
        print '    :o    ' + f
```

Printed output:

```
:o    crops.csv
;]    data1.shp
:o    rast
:(    xy1.txt
```

Example 10.7, 'scatting.py', uses nested loops to print a rhythmic pattern. Can you predict the output by reading the code? Spaces are embedded in the print statements to reflect the level of nesting. Those outside the FOR-loops have no spaces. Those inside the first FOR-loop are preceded by four spaces. Those inside the nested FOR-loop are proceeded by eight spaces. With this spacing inserted in the print statements, the output indentation mimics indentation of the lines of code that generated it.

Example 10.7

```
# scatting.py
# Purpose: Use nested loops to scat.
print '\nskeep-de'
for i in range(2):
    print '    beep'
    for j in range(3):
        print '        bop'
print 'ba-doop!'
```

Printed output:

```
skeep-de
    beep
        bop
        bop
        bop
    beep
        bop
        bop
        bop
ba-doop!
```

10.3 Directory Inventory

FOR-loops can be used with Python built-in `os` module functions `listdir` and `walk` to navigate data and directories. We demonstrate the simple `listdir` function here and the more complex `os.walk` function in Chapter 12. The `listdir` function lists the files in an input directory, such as sample data directory 'pics'. The following code prints a list of the files in 'C:/gispy/data/ch10/pics':

```
>>> import os
>>> theDir = 'C:/gispy/data/ch10/pics'
>>> # os.listdir returns a list of the files
>>> theFiles = os.listdir(theDir)
>>> theFiles
['istanbul.jpg', 'istanbul2.jpg', 'italy', 'jerusalem', 'marbleRoad.
jpg', 'spice_market.jpg', 'stage.jpg']
```

The output is a Python list, so a FOR-loop can be used to iterate through the files as follows:

```
>>> for fileName in theFiles:
...     print fileName
...
istanbul.jpg
istanbul2.jpg
italy
jerusalem
marbleRoad.jpg
spice_market.jpg
stage.jpg
```

Example 10.8 lists the files in 'C:/gispy/data/ch10' and creates a copy of any text files in the directory. The geoprocessing workspace is set and then this value is passed into the `listdir` function. Only `arcpy` module functionality is affected when the `arcpy.env.workspace` property is set. The `os` module does not consider this environment setting, so this value must still be passed to the `listdir` function. 'copyLoop.py' uses the string `endswith` method to find the files with a 'txt' extension, but this trick won't always work. For example, suppose you want to copy any files that are rasters. It could be difficult to check all of the extensions. Some may not even have extensions, such as the Esri GRID raster files. In fact, the need to process a specific type of file is so common in the GIS workflow that the `arcpy` package has functions for getting lists of specific data types within a directory. These functions are introduced in the next chapter.

Example 10.8: List files with the `os` module and geoprocess the files.

```
# copyLoop.py
# Purpose: Make a copy of each ASCII .txt extension file.

import arcpy, os

arcpy.env.workspace = 'C:/gispy/data/ch10'
outDir = 'C:/gispy/scratch/'
theFiles = os.listdir(arcpy.env.workspace)
for fileName in theFiles:
    if fileName.endswith('.txt'):
        outName = outDir + fileName[:-4] + 'V2.txt'
        arcpy.Copy_management(fileName, outName)
        print '{0} created.'.format(outName)
```

Notice that the `listdir` function returns a list of base names, not full path file names. But `os.path` methods that return file information, such as size or modification date, need to know where the file is located. If only a base name is given, this

method will look for the file in the os module current working directory. If it is not found there, it won't be able to find the file. For example, the os.path.exists function returns True if it determines that the file passed as an argument exists. The following code lists the files in a directory, then checks if they exist:

```
>>> import os
>>> theDir = 'C:/gispy/data/ch10/pics'
>>> # os.listdir returns a list of the files
>>> theFiles = os.listdir(theDir)
>>> for fileName in theFiles:
...     print os.path.exists(fileName)
...
False
False
False
...(and so forth)
```

The files were not deleted between listing them and checking for existence. Rather, the files were not found by the os.path.exists method, because the full path was not specified. To specify the full path, join the directory and file name:

```
>>> fullName = os.path.join(theDir, fileName)
>>> os.path.exists(fullName)
True
```

Example 10.9 uses the os.path.getmtime method to print a time stamp for the last time a file was modified. The os.path.join method is used inside the loop to create full path file names for the files. Example 10.8 set arcpy.env.workspace so that the arcpy.Copy method could locate the files. But the arcpy workspace has no effect on the os module current working directory. This is why you need to use the full path file names in Example 10.9, but not in Example 10.8. It is also possible to change the os module current working directory, using the os.chdir command. But it's important to realize that, this has no influence on the arcpy.env.workspace setting.

Example 10.9: Use os.path.join inside a loop to create full path file names.

```
# printModTime.py
# Purpose: For each file, print the time of most
#          recent modification.
# Input:   No arguments required.

import os, datetime

theDir = 'C:/gispy/data/ch10/pics'
theFiles = os.listdir(theDir)
```

```
for f in theFiles:
    fullName = os.path.join(theDir, f)
    # Get the modification time.
    print os.path.getmtime(fullName)
```

10.4 Indentation and the TabNanny

Chapter 4 introduced the PythonWin syntax check button  which checks the syntax and runs the TabNanny as well. Now that you're writing indented blocks of code, let's revisit the TabNanny, which checks for consistent spacing. The TabNanny inserts underline marks before indented code if inconsistencies in the indentation are found. By now you know that indentation within a block needs to be aligned. The TabNanny marks indicate a more subtle problem which sometimes occurs when code is copied from sources such as PDF files or Microsoft® software documents into PythonWin. For example, the following code was copied from a PowerPoint® presentation to PythonWin:

```
1 # tabNannyExample.py
2 decList = []
3 for num in range(5):
4     decNum = num + 0.5
5     decList.append(decNum)
6 print decList
```

To understand the cause of the error, make 'Whitespace' visible in PythonWin (View > Whitespace). *Whitespace* consists of the invisible characters (such as single spaces and tabs) used to indent and separate elements in a line of code. PythonWin displays spaces as gray dots and tabs as arrows. This is how our example looks with Whitespace turned on:

```
1 # tabNannyExample.py
2 decList = []
3 for num in range(5):
4     decNum = num + 0.5
5 →   decList.append(decNum)
6 print decList
```

The problem occurred because the Whitespace usage is inconsistent; Line 4 uses four spaces and line 5 uses one space and one tab. Replace the tab on line 5 with three spaces to repair the problem.

10.5 Key Terms

The `while` keyword
The `for` keyword
The `for` and `in` keyword pairing
Iterating variable
Sequence data objects
Nested looping
The `os.listdir` function

10.6 Exercises

1. **whileLoopino.py** Rewrite the following script, replacing the `FOR`-loop with a `WHILE`-loop.

```
for n in range(-50,150,5):
    print n,
```

   Note: The comma at the end of the print statement in NOT a typo. It's a way to make consecutive print statements appear on the same line separated by a space.

2. **forLoopino.py** Rewrite the following script, replacing the `WHILE`-loop with a `FOR`-loop and use the built-in `range` function.

```
index = 9
while index <= 99:
    print index
    index = index + 10
```

3. **outline.py** Write a script that uses nested `FOR`-loops to print the output shown below. Use the built-in `range` function for the numerical (outer) loop and use a hard-coded list [‘a’, ‘b’, ‘c’] for the inner loop. Prefix the inner loop print statements with a tab to indent them as shown.

```
1) Hehe
    a) Hoho
    b) Hoho
    c) Hoho
2) Hehe
    a) Hoho
    b) Hoho
    c) Hoho
```

3) `Hehe`
    a) `Hoho`
    b) `Hoho`
    c) `Hoho`
4) `Hehe`
    a) `Hoho`
    b) `Hoho`
    c) `Hoho`

4. **loopDistance.py** Write a script to find a set of distance tables from fire stations to schools in the feature classes in 'C:/gispy/data/ch10/county.gdb'. Use the Point Distance (Analysis) tool to determine the distance from the fire stations (input point features) to schools (near features) and use a `WHILE`-loop to run the tool with ten different search radius values: 500 meters, 1000 meters, … , 5000 meters. Use 'C:/gispy/scratch' as the output directory for the ten output tables, instead of creating them inside the file geodatabase. Name the output files using the numerical distance as 'dist500.dbf', 'dist1000.dbf', 'dist1500.dbf', and so forth. Hard-code the input features, near features, and output directory in this script, so that no script arguments are needed.

5. **loopAggregate.py** To compare the results of using the Aggregate Polygons (Cartography) tool with various values for the aggregation distance parameter, write a script that aggregates polygons in 'park.shp' for ten different aggregation distances: 150 yards, 250 yards, … , 1050 yards. Use the built-in `range` function and a `FOR`-loop. Vary the output names based on the loop iterator ('park150_agg.shp', 'park250_agg.shp', and so forth). Assume 'park.shp' resides in 'C:/gispy/data/ch10/' and place the output in 'C:/gispy/scratch'. No script arguments are needed.

6. **loopSimplify.py** To investigate the results of using the Simplify Line (Cartography) tool with various tolerance values, write a script that simplifies lines in 'parkLines.shp' with four different tolerance values. Use the built-in `range` function and a `FOR`-loop to perform line simplification for tolerance values: 30 feet, 60 feet, 90 feet, and 120 feet. Use the 'POINT_REMOVE' algorithm. Name output as 'simp30.shp', 'simp60.shp', 'simp90.shp', and 'simp120.shp', and place the output in 'C:/gispy/scratch'. No script arguments are needed.

7. **loopGetCount.py** Use the `os.listdir` method to get a list of the shapefiles in 'C:/gispy/data/ch10/'. Then for any shapefiles whose name contains the word 'out', case-insensitive, use the Get Count (Data Management) tool to determine the number of records in the attribute table. Use one script argument, a directory path. Report the results as shown in the example.

   Example input: C:/gispy/data/ch10/archive

   Example output:

```
linesOUT.shp has 530 entries.
outData.shp has 100 entries.
parkOutput.shp has 426 entries.
```

8. **loopFileSize.py** Use the `os.listdir` method to get a list of the files in a directory. Then use the `os.path.getsize` function to print the names of small files and their sizes. Allow the user to specify a maximum size in bytes. Use two script arguments, the directory path and the size threshold in bytes. Report the results as shown in the example.

   Example input: C:/gispy/data/ch10/64

   Example output:

```
data.txt is 42 bytes.
xy1.txt is 42 bytes.
xy_current.txt is 44 bytes.
```