5
ArcPy Cursors – Search, Insert, and Update

Now that we understand how to interact with ArcToolbox tools using ArcPy, and we have also covered using Python to create functions and import modules, we have a basic understanding of how to improve GIS workflows using Python. In this chapter we will cover data cursors and the Data Access module, introduced in 10.1. These data access cursors are a vast improvement on the cursors used in the arcgisscripting module (the precursor to ArcPy) and in earlier versions of ArcPy. Not only can the cursors search data, as we have seen, but they can update data using the Update Cursors and can add new rows of data using the Insert Cursor.

Data cursors are used to access data records contained within data tables, using a row by row iterative approach. The concept was borrowed from relational databases, where data cursors are used to extract data from tables returned from a SQL expression. Cursors are used to search for data, but also to update data or to add new data.

When we discuss creating data searches using ArcPy cursors, we are not just talking about attribute information. The new data access model cursors can interact directly with the shape field, and when combined with ArcPy Geometry objects, can perform geospatial functions and replace the need to pass data to ArcToolbox tools. Data access cursors represent the most useful innovation yet in the realm of Python automation for GIS.

In this chapter we will cover:

- Using Search Cursors to access attribute and spatial data
- Using Update Cursors to adjust values within rows
- Using insert cursors to add new data to a dataset
- Using cursors and the ArcPy Geometry object types to perform geospatial analyses in memory

The data access module

Introduced with the release of ArcGIS 10.1, the new data access module known as `arcpy.da` has made data interaction easier, and faster, than allowed by previous data cursors. By allowing for direct access to the shape field in a variety of forms (shape object, X values, Y values, centroid, area, length, and more), and a variety of formats (JavaScript Object Notation (JSON), Keyhole Markup Language (KML), Well Known Binary (WKB), Well-Known Text (WKT)), the data access module greatly increases the ability of a GIS analyst to extract and control shape field data.

The data access cursors accept a number of required and optional parameters. The required parameters are the path to the feature class as a string (or a variable representing the path) and the fields to be returned. If all fields are desired, using the asterisk notation and provide a list with an asterisk as a string as the field's parameter ([*]). If only a few fields are required, provide those fields as string fieldnames (for example ["NAME", "DATE"]).

The other parameters are optional but are very important, for both search and Update Cursors. A `where` clause in the form of a SQL expression can be provided next; this clause will limit the number of rows returned from the data set (as demonstrated by the SQL expression in the scripts in the last chapter). The SQL expressions used by the search and update cursors are not complete SQL expressions, as the `SELECT` or `UPDATE` commands are provided automatically by the choice of cursor. Only the `where` clause of the SQL expression is required for this parameter.

A spatial reference can be provided next in the ArcPy Spatial Reference format; this is not necessary if the data is in the correct format but can be used to transform data into another projection on the fly. There is no way to specify the spatial transformation used, however. The third optional parameter is a Boolean (or True/False) value that declares whether data should be returned in exploded points (that is, a list of the individual vertices) or in the original geometry format. The final optional parameter is another list that can be used to organize the data returned by the cursor; this list would include SQL keywords such as DISTINCT, ORBER BY, or GROUP BY. However, this final parameter is only available when working with a geodatabase.

Let's take a look at using `arcpy.da.SearchCursor` for shape field interactions. If we needed to produce a spreadsheet listing all bus stops along a particular route, and include the location of the data in an X/Y format, we could use the Add XY tool from the ArcToolbox. However, this has the effect of adding two new fields to our data, which is not always allowed, especially when the data is stored in enterprise geodatabases with fixed schemas. Instead, we'll use the SHAPE@XY token built into the data access module to easily extract the data and pass it to the `createCSV()` function from *Chapter 4, Complex ArcPy Scripts and Generalizing Functions,* along with the SQL expression limiting results to the stops of interest:

```
csvname = "C:\Projects\Output\StationLocations.csv"
headers = 'Bus Line Name', 'Bus Stop ID', 'X', 'Y'
create_CSV(headers, csvname, 'wb')
sql = "(NAME = '71 IB' AND BUS_SIGNAG = 'Ferry Plaza') OR (NAME = '71 OB'
AND BUS_SIGNAG = '48th Avenue')"
with arcpy.da.SearchCursor(Bus_Stops,['NAME', 'STOPID', 'SHAPE@XY'], sql)
as cursor:
    for row in cursor:
        linename = row[0]
        stopid = row[1]
        locationX = row[2][0]
        locationY = row[2][1]
        locationY = row[2][1]
        data = linename, stopid, locationX, locationY
        create_CSV(data, csvname)
```

Note that each row of data is returned as a tuple; this makes sense as the Search Cursor does not allow any data manipulation and tuples are immutable as soon as they are created. In contrast, data returned from Update Cursors is in list format, as lists can be updated. Both can be accessed using the indexing as shown previously.

Each row returned by the cursor is a tuple with three objects: the name of the bus stop, the bus stop ID, and finally another tuple containing the X/Y location of the stop. The objects in the tuple, contained in the variable `row`, are accessible using indexing: the bus stop name is at index 0, the ID is at index 1, and the location tuple is at index 2.

Within the location tuple, the X value is at index 0 and the Y value is at index 1; this makes it easy to access the data in the location tuple by passing a value as shown in the following:

```
locationX = row[2][0]
```

The ability to add lists and tuples and even dictionaries to another list or tuple or dictionary is a strong component of Python, making data access logical and data organization easy.

However, the spreadsheet returned from the previous code has a few issues: the location is returned in the native projection of the feature class (in this case, a State Plane projection), and there are rows of data that are repeated. It would be much more helpful if we could provide latitude and longitude values in the spreadsheet and the duplicate values were removed. Let's use the optional spatial reference parameter and a list to sort the data before we pass it to the `createCSV()` function:

```
spatialReference = arcpy.SpatialReference(4326)
sql = "(NAME = '71 IB' AND BUS_SIGNAG = 'Ferry Plaza') OR (NAME = '71 OB'
AND BUS_SIGNAG = '48th Avenue')"
dataList = []
with arcpy.da.SearchCursor(Bus_Stops, ['NAME','STOPID','SHAPE@XY'], sql,
spatialReference) as cursor:
    for row in cursor:
        linename = row[0]
        stopid = row[1]
        locationX = row[2][0]
        locationY = row[2][1]
        data = linename, stopid, locationX, locationY
        if data not in dataList:
            dataList.append(data)

[redacted]\StationLocations.csv"
h[redacted]us Stop ID','X','Y'
c[redacted]
for data in dataList:
    print(data)
```

The spatial reference is created by passing a code representing the desired projection system. In this case the code for the WGS 1984 Latitude and Longitude geographic system is 4326 and is passed to the `arcpy.SpatialReference()` method to create a spatial reference object that can be passed to the Search Cursor. Also, the `if` conditional is used to filter the data, accepting only one list per stop into the list called dataList. This new version of the code will produce a `CSV` file with the desired data. This CSV could then be converted into a KML with the service provided by `www.convertcsv.com/csv-to-kml.htm`, or even better, using Python. Use string formatting and loops to insert the data into pre-built KML strings.

Attribute field interactions

Apart from the shape field interactions, another improvement offered by the data access module cursors is the ability to call the fields in a feature class by using a list, as discussed previously. Earlier data cursors required the use of a less efficient `get value` function call, or required the fields to be called as if they were methods available to the function. The new method allows for all fields to be called by passing an asterisk, a valuable method to access fields in feature classes that have not been inspected previously.

One of the more valuable improvements is the ability to access the Unique ID field without needing to know whether the data set is a feature class or a shapefile. Because shapefiles had a feature ID or FID, and feature classes had an object ID, it was harder to program a Script tool to access the unique ID field. Data access module cursors allow for the use of the `OID@` string to request the unique ID from either type of input. This makes the need to know the type of unique ID irrelevant.

As demonstrated previously, other attribute fields are requested by a string in a list. The field names must match the true name of the field; alias names cannot be passed to the cursor. The fields can be in the list in any order desired, and will be returned in the order requested. Only the required fields have to be included in the list.

Here is a demonstration of requesting field information:

```
sql = "OBJECTID = 1"
with arcpy.da.SearchCursor(Bus_Stops,
            ['STOPID','NAME', 'OID@'],
            sql) as cursor:
for row in cursor:
```

If the fields in the fields list were adjusted, the data in the resulting row would reflect the adjustment. Also, all of the members of the tuple returned by the cursor are accessible by zero-based indexing.

Update cursors

Update cursors are used to adjust data within existing rows of data. Updates become very important when calculating data or converting null values to a non-null value. Combined with specific SQL expressions, data can be targeted for updating with newly collected or calculated values.

Note that running code containing an Update Cursor will change, or update, the data on which it operates. It is a good idea to make a copy of the data to test out the code before running it on the original data.

All data access module Search Cursor parameters discussed previously are valid for Update Cursors. The main difference is that data rows returned by Update Cursors are returned as lists. Because lists are mutable, they can be adjusted using a list value assignment.

As an example, let's imagine that the bus line 71 will be renamed to the 75. Both inbound and outbound lines will be affected, so a SQL expression must be included to get all rows of data associated with the line. Once the data cursor is created, the rows returned must have the name adjusted, added back into the list, and the Update cursor's `updateRow` method must be invoked. Here is how this scenario would look in code:

```
sql = "NAME LIKE '71%'"
with arcpy.da.UpdateCursor(Bus_Stops, ['NAME'],sql),) as cursor:
    for row in cursor:
        lineName = row[0]
        newName = lineName.replace('71','75')
        row[0] = newName
```

The SQL expression will return all rows of data with a name starting with `71`; this will include `71 IB` and `71 OB`. Note that the SQL expression must be enclosed in double quotes, as the attribute value needs to be in single quotes.

For each row of data, the name at position zero in the row returned is assigned to the variable lineName. This variable, a string, uses the `replace()` method to replace the characters `71` with the characters `75`. This could also just be replacing `1` with `5` but I wanted to be explicit as to what is being replaced.

Once the new string has been generated, it is assigned to the variable `newName`. This variable is then added to the list returned by the cursor using list assignment; this will replace the data value that initially occupied the zero position in the list. Once the row value has been assigned, it is then passed to the cursor's `updateRow()` method. This method accepts the row and updates the value in the feature class for that particular row.

Updating the shape field

For each row, all values included in the list returned by the cursor are available for update, except the unique ID (while no exception will be thrown, the UID values will not be updated). Even the shape field can be adjusted, with a few caveats. The main caveat is that the updated shape field must be the same geometry type as the original row, a point can be replaced with a point, a line with a line, and a polygon with another polygon.

Adjusting a point location

If a bus stop was moved down the street from its current position, it would need to be updated using an Update Cursor. This operation will require a new location in an X/Y format, preferably in the same projection as the feature class to avoid any loss of location fidelity in a spatial transformation. There are two methods available to us for creating a new point location, depending on the method used to access the data. The first method is used when the location data is requested using the `SHAPE@` tokens, and requires the use of an ArcPy Geometry type, in this case the Point type. The ArcPy Geometry types are discussed in detail in the next chapter.

```
sql = 'OBJECTID < 5'
with arcpy.da.UpdateCursor(Bus_Stops, [ 'OID@', 'SHAPE@'],sql) as cursor:
    for row in cursor:
        row[1] = arcpy.Point(5999783.78657, 2088532.563956)
```

By passing an X and Y value to the ArcPy Point Geometry, a Point shape object is created and passed to the cursor in the updated list returned by the cursor. Assigning a new location to the shape field in a tuple, then using the cursor's `updateRow()` method allows the shape field value to be adjusted to the new location. Because the first four bus stops are at the same location, they are all moved to the new location.

The second method applies to all other forms of shape field interactions, including the `SHAPE@XY`, `SHAPE@JSON`, `SHAPE@KML`, `SHAPE@WKT`, and `SHAPE@WKB` tokens. These are updated by passing the new location in the format requested back to the cursor and updating the list:

```
sql = 'OBJECTID < 5'
with arcpy.da.UpdateCursor(Bus_Stops, [ 'OID@', 'SHAPE@XY'],sql) as
cursor:
    for row in cursor:
        row[1] =(5999783.786500007, 2088532.5639999956)
```

Here is the same code using the `SHAPE@JSON` keyword and a JSON representation of the data:

```
sql = 'OBJECTID < 5'
with arcpy.da.UpdateCursor(Bus_Stops, [ 'OID@', 'SHAPE@JSON'],sql) as
cursor:
    for row in cursor:
        print row
        row[1] = u'{"x":5999783.7865000069, "y":2088532.5639999956,
                    "spatialReference":{"wkid":102643}}'
```

As long as the keyword, the data format, and the geometry type match, the location is updated to the new coordinates. The keyword method is very useful when updating points, however, the `SHAPE@XY` keyword does not work with lines or polygons as the location returned represents the centroid of the requested geometry.

Deleting a row using an Update Cursor

If we need to remove a row of data, the `UpdateCursor` has a `deleteRow` method that works to remove the row. Note that this will completely remove the data row, making it unrecoverable. This method does not require a parameter to be passed to it; instead, it will remove the current row:

```
sql = 'OBJECTID < 2'
Bus_Stops = r'C:\Projects\PacktDB.gdb\Bus_Stops'
with arcpy.da.UpdateCursor(Bus_Stops,
          ['OID@',
           'SHAPE@XY'],sql) as cursor:
    for row in cursor:
```

Using an Insert Cursor

Now that we have a grasp on how to update existing data, let's investigate using Insert Cursors to create new data and add it to a feature class. The methods involved are very similar to using other data access cursors, except that we do not need to create an iterable cursor to extract rows of data; instead, we will create a cursor that will have the special `insertRow` method that is capable of adding data to the feature class row by row.

The Insert Cursor can be called using the same `with..as` syntax but generally it is created as a variable in the flow of the script.

Note that only one cursor can be invoked at a time; an exception (a Python error) will be generated when creating two insert (or update) cursors without first removing the initial cursor using the Python del keyword to remove the cursor variable from memory. This is why the `with..as` syntax is preferred by many.

The data access module's Insert Cursor requires some of the same parameters as the other cursors. The feature class to be written to and the list of fields that will have data inserted (this includes the shape field) are required. The spatial reference will not be used as the new shape data must be in the same spatial reference as the feature class. No SQL expression is allowed for an Insert Cursor.

The data to be added to the feature class will be in the form of a tuple or a list, in the same order as the fields that are listed in the fields list parameter. Only fields of interest need to be included in the list of fields, meaning not every field needs a value in the list to be added. When adding a new row of data to a feature class, the unique ID will automatically be generated, making it unnecessary to explicitly include the unique ID (in the form of the `OID@` keyword) in the list of fields to be added.

Let's explore code that could be used to generate a new bus stop. We'll write to a test dataset called `TestBusStops`. We are only interested in the Name and Stop ID fields, so those fields along with the shape field (which is in a State Plane projection system) will be included in the data list to be added:

```
Bus_Stops = r'C:\Projects\PacktDB.gdb\TestBusStops'
insertCursor = arcpy.da.InsertCursor(Bus_Stops,
['SHAPE@', 'NAME','STOPID'])
coordinatePair = (6001672.5869999975, 2091447.0435000062)
newPoint = arcpy.Point(*coordinatePair)
dataList = [newPoint,'NewStop1',112121]
insertCursor.insertRow(dataList)
del insertCursor
```

If there is an iterable list of data to be inserted into the feature class, create the Insert Cursor variable before entering the iteration, and delete the Insert Cursor variable once the data has been iterated through, or use the with..as method to automatically delete the Insert Cursor variable when the iteration is complete:

```
Bus_Stops = r'C:\Projects\PacktDB.gdb\TestBusStops'
listOfLists = [[(6002672.58675, 2092447.04362),'NewStop2',112122],
               [(6003672.58675, 2093447.04362),'NewStop3',112123],
               [(6004672.58675, 2094447.04362),'NewStop4',112124]
               ]

with arcpy.da.InsertCursor(Bus_Stops,
          ['SHAPE@',
           'NAME',
           'STOPID']) as iCursor:
    for dataList in listOfLists:
        newPoint = arcpy.Point(*dataList[0])
        dataList[0] = newPoint
```

As a list, the `listOfLists` variable is iterable. Each list within it is considered as dataList in the iteration, and the first value in `dataList` (the coordinate pair) is passed to the `arcpy.Point()` function to create a `Point` object. The `arcpy.Point()` function requires two parameters, `X` and `Y`; these are extracted from the coordinate pair tuple using the asterisk, which 'explodes' the tuple and passes the values it contains to the function. The `Point` object is then added back into `dataList` using an index-based list assignment, which would not be available to us if the `dataList` variable was a tuple (we would instead have to create a new list and add in the `Point` object and the other data values).

Inserting a polyline geometry

To create and insert a polyline-type shape field from a series of points, it's best to use the `SHAPE@` keyword. We will also further explore the ArcPy Geometry types, which will be discussed in the next chapter. When working with the `SHAPE@` keyword, we have to work with data in ESRI's spatial binary formats, and the data must be written back to the field in the same format using the ArcPy Geometry types.

To create a polyline, there is one requirement, at least two valid points made of two coordinate pairs. When working with the `SHAPE@` keyword, there is a methodology to converting the coordinate pairs into an ArcPy Point and then adding it to an ArcPy Array, which is then converted into an ArcPy Polyline to be written back to the shape field:

```
listOfPoints = [(6002672.58675, 2092447.04362),
                (6003672.58675, 2093447.04362),
                (6004672.58675, 2094447.04362)
                ]
line = 'New Bus Line'
lineID = 12345
busLine = r'C:\Projects\PacktDB.gdb\TestBusLine'
insertCursor = arcpy.da.InsertCursor(busLine, ['SHAPE@',
'LINE', 'LINEID'])
lineArray = arcpy.Array()
for pointsPair in listOfPoints:
    newPoint = arcpy.Point(*pointsPair)
    lineArray.add(newPoint)
newLine = arcpy.Polyline(lineArray)
insertData = newLine, line, lineID
```

The three coordinate pairs in tuples are iterated and converted into Point objects, which are in turn added to the Array object called `lineArray`. The Array object is then added to the Polyline object called `newLine`, which is then added to a tuple with the other data attributes and inserted into the feature class by the `InsertCursor`.

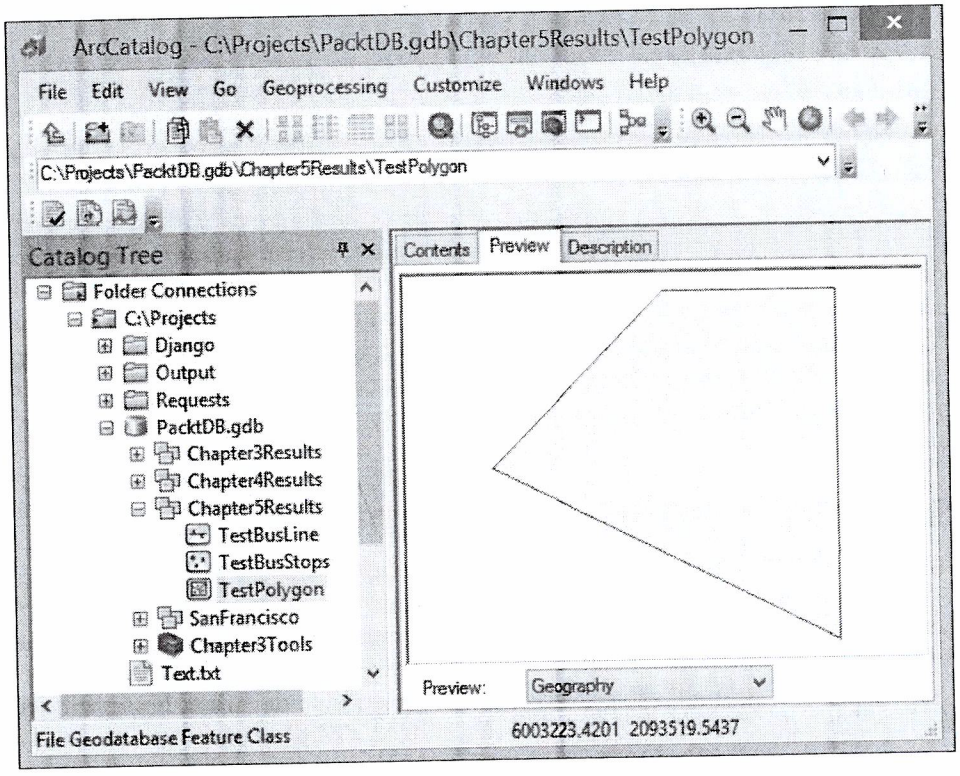
Inserting a polygon geometry

Polygons are also inserted, or updated, using cursors. The ArcPy Polygon Geometry type does not require the coordinate pairs to include the first point twice (that is, as the first point and as the last point). The polygon is closed automatically by the `arcpy.Polygon()` function:

```
listOfPoints = [(6002672.58675, 2092447.04362),
                (6003672.58675, 2093447.04362),
                (6004672.58675, 2093447.04362),
                (6004672.58675, 2091447.04362)
                ]
polyName = 'New Polygon'
```

```
polyID = 54321
blockPoly = r'C:\Projects\PacktDB.gdb\Chapter5Results\TestPolygon'
insertCursor = arcpy.da.InsertCursor(blockPoly,
['SHAPE@', 'BLOCK', 'BLOCKID'])
polyArray = arcpy.Array()
for pointsPair in listOfPoints:
    newPoint = arcpy.Point(*pointsPair)
    polyArray.add(newPoint)
newPoly = arcpy.Polygon(polyArray)
insertData = newPoly, polyName, polyID
insertCursor.insertRow(insertData)
```

Here is a visualization of the result of the insert operation:



Summary

In this chapter we covered the basic uses of data access module cursors. Search, update and Insert Cursors were explored and demonstrated, and a special focus was placed on the use of these cursors for extracting shape data from the shape field. Cursor parameters were also introduced, including the spatial reference parameter and the SQL expression `where` clause parameter. In the next chapter, we will further explore the use of cursors, especially with the use of ArcPy Geometry types.

Maybe I should create the "test" polygon first?