6
Working with ArcPy Geometry Objects

The essence of geospatial analysis is using geometric shapes – points, lines, and polygons – to model the geography of real world objects and their location-based relationships. The simple shapes and their geometric properties of location, length and area are processed using geospatial operations to generate analysis results. It is the combination of modeled geographic data and the associated attribute information that separate geospatial information systems from all other information systems.

Until ArcPy, processing the feature class geometry using the geospatial operations was depended on the pre-built tools within ArcToolbox. ArcPy has made it possible to directly access the geometric shapes which are stored as mathematical representations in the shape field of feature classes. Once accessed, this geometric data is loaded into ArcPy geometry objects to make the data available for analysis within an ArcPy script. Because of this advance, writing scripts that access geometry fields and use them to perform analysis has transformed ArcGIS geospatial analysis. In this chapter, we'll explore how to generate and use the ArcPy geometry objects to perform geospatial operations, and apply them to the bus stops analysis.

In this chapter, we will cover: `Point` and `Array` constructor objects and `PointGeometry`, `Polyline`, and `Polygon` geometry objects

- How to use the geometry objects to perform geospatial operations
- How to integrate the geometry objects into scripts
- How to perform common geospatial operations using the geometry objects
- How to replace the use of ArcToolbox tools in the script with geometry object methods

ArcPy geometry object classes

In designing geometry objects, the authors of ArcPy made it possible to perform geospatial operations in memory, reducing the need to use tools in the ArcToolbox for these operations. This will result in speed gains as there is no need to write the results of the calculations to disk at each step of the analysis. Instead, the results of the steps can be passed from function to function within the script. The final results of the analysis can be written to the hard drive as a feature class, or they can be written into a spreadsheet or passed to another program.

The geometry objects are written as Python classes- special blocks of code that contain internal functions. The internal functions are the methods and properties of the geometry objects; when called they allow the object to perform an operation (a method) or to reveal information about the geometry object (a property). Python classes are written with a main class that contains shared methods and properties, and with sub-classes that reference the main class but also have specific methods and properties that are not shared. Here, the main class is the ArcPy `Geometry` object, while the sub-classes are the `PointGeometry`, `Multipoint`, `Polyline` and `Polygon` objects.

The geometry objects are generated in three ways. The first requires using data cursors to read existing feature classes and passing a special keyword as a field name. The shape data returned by the cursor is a geometry object. The second method is to create new data by passing raw coordinates to a constructor object (either a `Point` or `Array` object), which is then passed to a geometry object. The third method is to read data from a feature class using the Copy Features tool from the ArcToolbox.

Each geometry object has methods that allow for read access and write access. The read access methods are important for accessing the coordinate points that constitute the points, lines and polygons. The write access methods are important when generating new data objects that can be analyzed or written to disk.

The `PointGeometry`, `Multipoint`, `Polyline`, and `Polygon` geometry objects are used for performing analysis upon their respective geometry types. The generic geometry object can accept any geometry type and an optional spatial reference to perform geospatial operations when there is no need to discern the geometry type.

Two other ArcPy classes will be used for performing geospatial operations in memory: the `Array` object and the `Point` object. They are constructor objects, as they are not sub-classed from the geometry class, but are instead used to construct the geometry objects. The `Point` object is used to create coordinate points from raw coordinates. The `Array` object is a list of coordinate points that can be passed to a `Polyline` or `Polygon` object, as a regular Python list of ArcPy `Point` objects cannot be used to generate those geometry objects.

ArcPy Point objects

`Point` objects are the building blocks used to generate geometry objects. Also, all of the geometry objects will return component coordinates as `Point` objects when using read access methods. `Point` objects allow for simple geometry access using its *X*, *Y* and *Z* properties, and a limited number of geospatial methods, such as `contains`, `overlaps`, `within`, `touches`, `crosses`, `equals`, and `disjoint`. Let's use IDLE to explore some of these methods with two `Point` geometry objects with the same coordinates:

```
>>> Point = arcpy.Point(4,5)
>>> point1  = arcpy.Point(4,5)
>>> Point.equals(point1)
True
>>> Point.contains(point1)
True
>>> Point. crosses(point1)
False
>>> Point.overlaps(point1)
False
>>> Point.disjoint(point1)
False
>>> Point.within(point1)
True
>>> point.X, Point.Y
(4.0, 5.0)
```

In these examples, we see some of the idiosyncrasies of the `Point` object. With two points that have the same coordinates, the results of the `equals` method and the `disjoint` method are as expected. The `disjoint` method will return `True` when the two objects do not share coordinates, while the opposite is true with the `equals` method. The `contains` method will work with the two Point objects and return True. The `crosses` method and `overlaps` method are somewhat surprising results, as the two Point objects do overlap in location and could be considered to cross; however, those methods do not return the expected result as they are not built to compare two points.

ArcPy Array objects

Before we progress up to `Polyline` and `Polygon` objects, we need to understand the ArcPy `Array` object. It is the bridge between the `Point` objects and those geometry objects that require multiple coordinate points. `Array` objects accept `Point` objects as parameters, and the `Array` object is in turn passed as a parameter to the geometry object to be created. Let's use Point objects with an Array object to understand better how they work together.

The `Array` object is similar to a Python list, with `extend`, `append`, and `replace` methods, and also has unique methods such as `add` and `clone`. The `add` method will be used to add `Point` objects individually:

```
>>> Point = arcpy.Point(4,5)
>>> point1  = arcpy.Point(7,9)
>>> Array = arcpy.Array()
>>> Array.add(point)
>>> Array.add(point1)
```

The `extend()` method would add a list of Point objects all at once:

```
>>> Point = arcpy.Point(4,5)
>>> point1 = arcpy.Point(7,9)
>>> pList = [Point,point1]
>>> Array = arcpy.Array()
>>> Array.extend(pList)
```

The `insert` method will put a `Point` object in the Array at a specific index, while the `replace` method is used to replace a `Point` object in an Array by passing an index and a new `Point` object:

```
>>> Point  = arcpy.Point(4,5)
>>> point1  = arcpy.Point(7,9)
>>> point2  = arcpy.Point(11,13)
>>> pList = [Point,point1]
>>> Array = arcpy.Array()
   >>> Array.extend(pList)
>>> Array.replace(1,point2)
>>> point3  = arcpy.Point(17,15)
>>> Array.insert(2,point3)
```

The `Array` object, when loaded with `Point` objects, can then be used to generate the other geometry objects.

ArcPy Polyline objects

The `Polyline` object is generated with an `Array` object that has at least two `Point` objects. As given in the following IDLE example, once an `Array` object has been generated and loaded with the `Point` objects, it can then be passed as a parameter to a `Polyline` object:

```
>>> Point  = arcpy.Point(4,5)
>>> point1  = arcpy.Point(7,9)
>>> pList = [Point,point1]
>>> Array = arcpy.Array()

>>> Array.extend(pList)
>>> pLine = arcpy.Polyline(Array)
```

Now that the Polyline object has been created, its methods can be accessed. This includes methods to reveal the constituent coordinate points within the polyline, and other relevant information:

```
>>> pLine.firstPoint
<Point (4.0, 5.0, #, #)>
>>> pLine.lastPoint
<Point (7.0, 9.0, #, #)>
pLine.getPart()
<Array [<Array [<Point (4.0, 5.0, #, #)>, <Point (7.0, 9.0, #, #)>]>]>
>>> pLine.trueCentroid
<Point (5.5, 7.0, #, #)>
>>> pLine.length
5.0
>>> pLine.pointCount
2
```

This example `Polyline` object has not been assigned a spatial reference system, so the length is unitless. When a geometry object does have a spatial reference system, the linear and areal units will be returned in the linear unit of the system.

The `Polyline` object is also our first geometry object with which we can invoke geometry class methods that perform geospatial operations, such as buffers, distance analyses, and clips:

```
>>> bufferOfLine = pLine.buffer(10)
>>> bufferOfLine.area
413.93744395
>>> bufferOfLine.contains(pLine)
True
>>> newPoint = arcpy.Point(25,19)
>>> pLine.distanceTo(newPoint)
20.591260281974
```

Another useful method of `Polyline` objects is the `positionAlongLine` method. It is used to return a `PointGeometry` object, discussed in the following, at a specific position along the line. This position along the line can either be a numeric distance from the first Point or as a percentage (expressed as a float from 0-1), when using the optional second parameter:

```
>>> nPoint = pLine.positionAlongLine(3)
>>> nPoint.firstPoint.X, nPoint.firstPoint.Y
(5.8, 7.4)>>> pPoint = pLine.positionAlongLine(.5,True)
          >>> pPoint.firstPoint.X,pPoint.firstPoint.Y
(5.5, 7.0)
```

There are a number of other methods available to `Polyline` objects. More information is available here: `http://resources.arcgis.com/en/help/main/10.2/index.html#//018z00000008000000`

ArcPy Polygon objects

To create a `Polygon` object, an `Array` object must be loaded with `Point` objects and then passed as a parameter to the `Polygon` object. Once the `Polygon` object has been generated, the methods available to it are very useful for performing geospatial operations. The geometry objects can also be saved to disk using the ArcToolbox `CopyFeatures` tool. This IDLE example demonstrates how to generate a `shapefile` by passing a `Polygon` object and a raw string filename to the tool:

```
>>> import arcpy
>>> point1 = arcpy.Point(12,16)
>>> point2 = arcpy.Point(14, 18)
>>> point3 = arcpy.Point(11, 20)
```

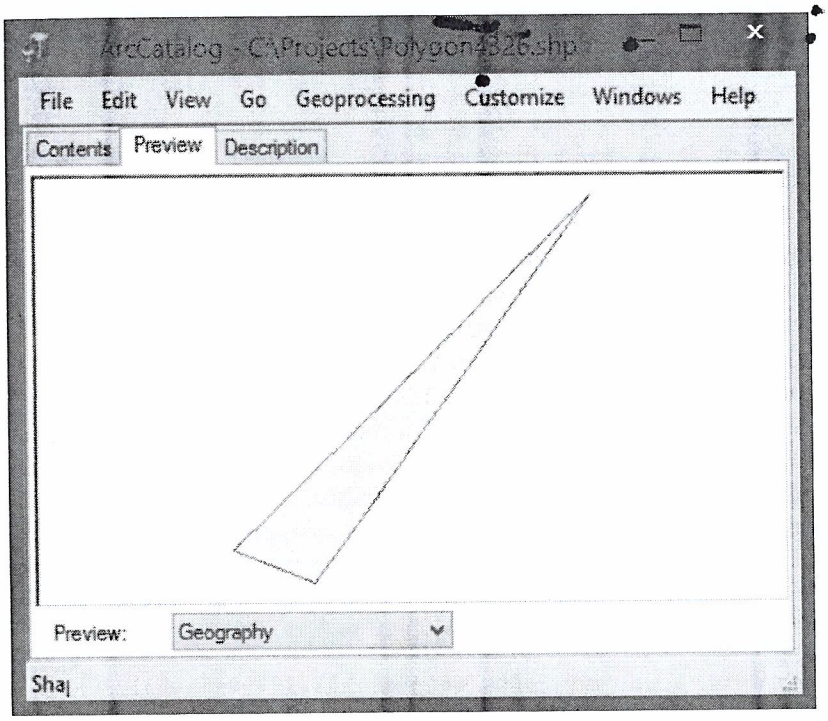
```
>>> Array = arcpy.Array()
>>> Points = [point1,point2,point3]
>>> Array.extend(points)
>>> Polygon = arcpy.Polygon(array)
>>> arcpy.CopyFeatures_management(polygon, r'C:\Projects\Polygon.shp')
<Result 'C:\\Projects\\Polygon.shp'>
```

Polygon object buffers

`Polygon` objects, like `Polyline` objects, have methods that make it easy to perform geospatial operations such as buffers. By passing a number to the buffer method as a parameter, a buffer will be generated in memory. The unit of the number is determined by the `SpatialReference` system. Internal buffers can be generated by supplying negative buffer numbers; the buffer generated being the area within the `Polygon` object at the specified distance from the Polygon perimeter. Clips, unions, symmetrical differences, and more operations are available as methods, as are within or contains operations; even projections can be performed using the `Polygon` object methods as long as it has a `SpatialReference` system object passed as a parameter. Following is a script that will create two shapefiles with two separate `SpatialReference` systems, each identified by a numeric code (2227 and 4326) from the EPSG coding system:

```
import arcpy
Point  = arcpy.Point(6004548.231,2099946.033)
point1  = arcpy.Point(6008673.935,2105522.068)
point2  = arcpy.Point(6003351.355,2100424.783)
Array = arcpy.Array()
Array.add(point1)
Array.add(point)
array.add(point2)
Polygon = arcpy.Polygon(array, 2227)
buffPoly = Polygon.buffer(50)
features = [Polygon,buffPoly]
arcpy.CopyFeatures_management(features,
                              r'C:\Projects\Polygons.shp')
spatialRef = arcpy.SpatialReference(4326)
polygon4326 = Polygon.projectAs(spatialRef)
arcpy.CopyFeatures_management(polygon4326,
                              r'C:\Projects\polygon4326.shp')
```

Here is how the second shapefile looks in the **ArcCatalog Preview** window:

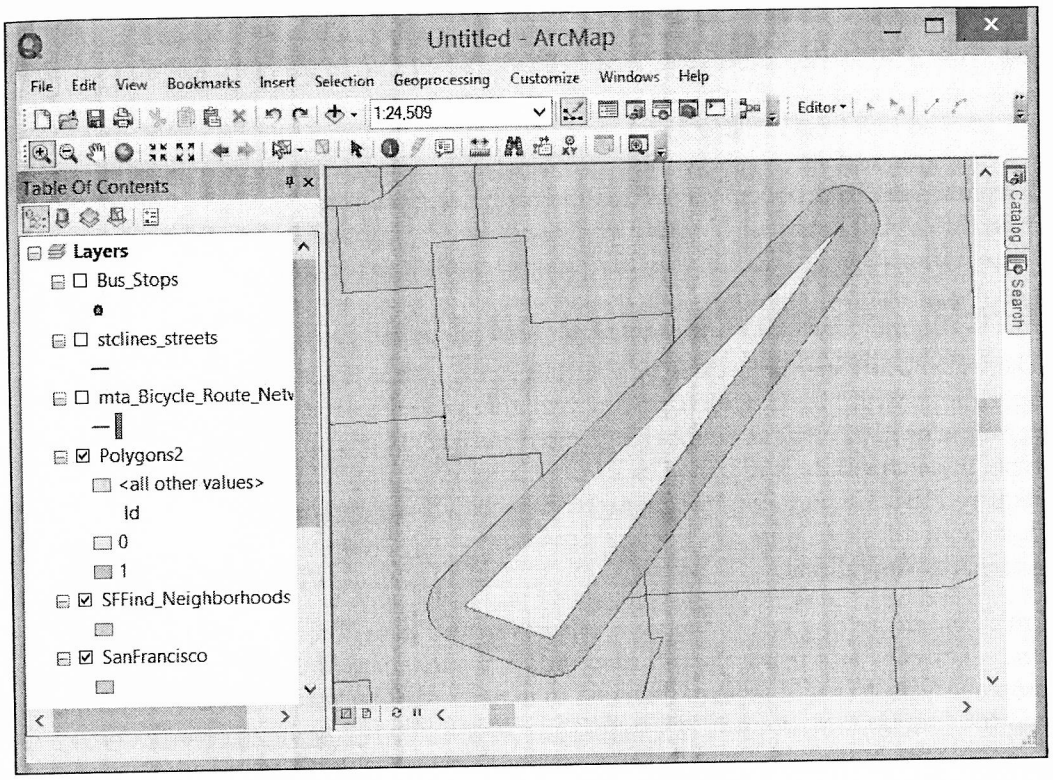


Other Polygon object methods

Unlike the clip tool in the ArcToolbox, which can clip a Polygon using another polygon, the clip method requires an extent object (another ArcPy class) and is limited to a rectangular envelope around the area to be clipped. To remove areas from a polygon, the difference method can work like the clip or erase tool in the ArcToolbox:

```
buffPoly = Polygon.buffer(500)
donutHole =buffPoly.difference(Polygon)
features = [Polygon,donutHole]
arcpy.CopyFeatures_management(features,
                              r"C:\Projects\Polygons2.shp")
```

Here is the donut hole-like result of the buffer and difference operation. The buffer with the donut hole surrounds the original `Polygon` object:



ArcPy geometry objects

The generic geometry object is quite useful for creating in memory a copy of the geometry of a feature class, without first needing to know which type of geometry the feature class contains. Like all of the ArcPy geometry objects, its read methods include the extraction of the data in many formats such as JSON, WKT, and WKB. The area (if it is a polygon), the centroid, the extent, and the constituent points of each geometry are also available, as demonstrated previously.

Here is an example of reading the geometry of a feature class into memory using the `CopyFeatures` tool:

```
import arcpy
cen2010 = r'C:\Projects\ArcPy.gdb\SanFrancisco\CensusBlocks2010'
blockPolys = arcpy.CopyFeatures_management(cen2010,
                                           arcpy.Geometry())
```

The variable `blockPolys` is a Python list containing all of the geometries loaded into it; in this case it is census blocks. The list can then be iterated to be analyzed.

ArcPy PointGeometry objects

The `PointGeometry` object is very useful for performing these same geospatial operations with points, which are not available with the `Point` objects. When a cursor is used to retrieve shape data from a feature class with a `PointGeometry` type, the shape data is returned as a `PointGeometry` object. While `Point` objects are required to construct all other geometry objects when a cursor is not used to retrieve data from a feature class, it's the `PointGeometry` object that is used to perform point geospatial operations.

Let's explore getting `PointGeometry` objects from a data access module `SearchCursor` and using the returned data rows to create buffered points. In our bus stop analysis, this will replace the need to use the ArcToolbox Buffer tool to create the 400 foot buffers around each stop. The script in the following uses a dictionary to collect the buffer objects and then searches the census blocks using another Search Cursor. To access the shape field using the `SearchCursor()` method, the `SHAPE@` token is passed as one of the fields. Then, the script will iterate through the bus stops and find all census blocks with which each stop intersects:

```
# Generate 400 foot buffers around each bus stop
import arcpy,csv
busStops = r"C:\Projects\PacktDB.gdb\SanFrancisco\Bus_Stops"
censusBlocks2010 = r"C:\Projects\PacktDB.gdb\SanFrancisco\
CensusBlocks2010"

sql = "NAME = '71 IB' AND BUS_SIGNAG = 'Ferry Plaza'"
dataDic = {}
with arcpy.da.SearchCursor(busStops, ['NAME','STOPID','SHAPE@'], sql) as
cursor:
    for row in cursor:
        linename = row[0]
```

```
        stopid = row[1]
        shape = row[2]
        dataDic[stopid] = shape.buffer(400), linename
```

Now that the data has been retrieved and the buffers have been generated using the buffer method of the `PointGeometry` objects, the buffers can be compared against the census block geometry using iteration and a Search Cursor. There will be two geospatial methods used in this analysis: `overlap` and `intersect`. The overlaps method is a boolean operation, returning a value of true or false when one geometry is compared against another. The `intersect` method is used to get the actual area of the intersect as well as identifying the population of each block. Using the intersect requires two parameters: a second geometry object, and an integer indicating which type of geometry to return (1 for point, 2 for line, 4 for polygon). We want the polygonal area of intersect returned to have an area of intersection available along with the population data:

```
# Intersect census blocks and bus stop buffers
processedDataDic = {} = {}
for stopid in dataDic.keys():
    values = dataDic[stopid]
    busStopBuffer = values[0]
    linename = values[1]
    blocksIntersected = []
    with arcpy.da.SearchCursor(censusBlocks2010,
    ['BLOCKID10','POP10','SHAPE@']) as cursor:
for row in cursor:
            block = row[2]
            population = row[1]
            blockid = row[0]
            if busStopBuffer.overlaps(block) ==True:
                interPoly = busStopBuffer.intersect(block,4)
                data = row[0],row[1],interPoly, block
                blocksIntersected.append(data)
    processedDataDic[stopid] = values, blocksIntersected
```

This portion of the script iterates through the blocks and intersects against the buffered bus stops. Now that we can identify the blocks that touch the buffer around each stop and the data of interest has been collected into the dictionary, it can be processed and the average population of all of the blocks touched by the buffer can be calculated:

```
# Create an average population for each bus stop
dataList = []
for stopid in processedDataDic.keys():
    allValues = processedDataDic[stopid]
    popValues = []
    blocksIntersected = allValues[1]
    for blocks in blocksIntersected:
        popValues.append(blocks[1])
    averagePop = sum(popValues)/len(popValues)
    busStopLine = allValues[0][1]
    busStopID = stopid
    finalData = busStopLine, busStopID, averagePop
    dataList.append(finalData)
```

Now that the data has been created and added to a list, it can be outputted to a spreadsheet using the `createCSV` module we created in *Chapter 4, Complex ArcPy Scripts and Generalizing Functions*:

```
# Generate a spreadsheet with the analysis results
def createCSV(data, csvname, mode ='ab'):
    with open(csvname, mode) as csvfile:
        csvwriter = csv.writer(csvfile, delimiter=',')
        csvwriter.writerow(data)

csvname = "C:\Projects\Output\StationPopulations.csv"
headers = 'Bus Line Name','Bus Stop ID', 'Average Population'
createCSV(headers, csvname, 'wb')
for data in dataList:
    createCSV(data, csvname)
```

The data has been processed and written to the spreadsheet. There is one more step that we can take with the data and that is to use the area of the intersection to create a proportional population value for each buffer. Let's redo the processing of the data to include the proportional areas:

```
dataList = []
for stopid in processedDataDic.keys():
    allValues = processedDataDic[stopid]
    popValues = []
    blocksIntersected = allValues[1]
    for blocks in blocksIntersected:
       pop = blocks[1]
        totalArea = blocks[-1].area
        interArea = blocks[-2].area
        finalPop = pop * (interArea/totalArea)
        popValues.append(finalPop)
    averagePop = round(sum(popValues)/len(popValues),2)
    busStopLine = allValues[0][1]
    busStopID = stopid
    finalData = busStopLine, busStopID, averagePop
    dataList.append(finalData)
```

Now the script is taking full advantage of the power of ArcPy geometry objects, and the script is running completely in memory which avoids producing any intermediate datasets.

Summary

In this chapter, we discussed in detail the use of ArcPy geometry objects. These varied objects have similar methods and are, in fact, sub-classed from the same Python class. They are useful for performing in-memory geospatial analyses, which avoids having to read and write data from the hard drive and also skips creating any intermediate data.

ArcPy geometry objects will become an important part of automating geospatial workflows. Combining them with Search Cursors makes ArcPy more useful than any earlier implementation of Python scripting tools for ArcGIS. Next, we will convert the raw script into a script tool that can be executed directly from the ArcToolbox or a personal toolbox in a geodatabase.