# Chapter 8 Working with geometries

## 8.1 Introduction

This chapter describes how to work with geometry, including how to create geometry objects from existing feature classes and how to read the properties of these geometry objects.

Individual features, such as points, polylines, and polygons, can be broken down into their vertices.

Geometries can also be written by creating geometry objects from a list of coordinates.

Being able to read and write geometries provides detailed control of feature classes, features, and the parts and vertices that make up features.

## 8.2 Working with geometry objects

Each feature in a feature class contains a set of points that define the vertices of the feature.

These points can be accessed using geometry objects, such as Point, Polyline, Point Geometry, and MultiPoint, which returns an array of Point objects.

Accessing full geometry objects is somewhat time consuming.

As a result, script that work with the full geometry objects of large datasets can become very slow.

If you need only speicific properties of the geometry, you can use geometry tokens as shortcuts to access geometry properties.

For example, SHAPE@XY will return a tuple of x,y coordinates representing the feature's centroid, and SHAPE@LENGTH will return the feature's length as a double.

On the other hand, SHAPE@ will return the full geometry object.

## 8.3 Reading geometries

Each feature in a feature class consists of a set of points that define the vertices of point, polyline, or polygon features.

In the case of a point feature class, each point consists of only a single vertex.

Polyline and polygon features consist of multiple vertices. Each vertex is a location defined by coordinates.

The figure illustrates how points, poly lines, and polygons are defined by vertices.

In the following example, a search cursor and a for loop are used to iterate over the rows of a point feature class called hospitals.shp.

A geometry token is used to retrieve the x,y coordinates of the point objects, which are then printed.

The script is as follows:

The hospitals shape file, which is a point feature class, is shown in the figure Each point has a pair of x,y coordinates.

Point feature classes are relatively simple because there is only a single point object for each feature.

For other types of feature classes such as poly lines and polygons, an array of point objects is returned for each feature.

To work with these arrays, an extra iteration is needed.

In a typical script, a for loop is used to iterate over the rows in the table and a second for loop is used to iterate over the point objects in each array.

In the following example code, a for loop is used to iterate over the rows in a shapefile.

For every row, the value of the OID (object identifier) field is printed - without it, you could not tell the start and end of each array of points.

For each row, a geometry object is obtained, which consists of an array containing an array of point objects.

The getPart method is used to obtain an array of point objects for the first (and only) part of the geometry.

(Note: Geometry covered in more detail in the next section.)

A for loop is used to iterate over all the point objects in the array and print the x,y coordinates.

The code is as follows:

The roads shapefile, which is shown in the figure, has vertices that are shown for emphasis-these are usua11y not visible for a polyline shapfile.

The shapefile consists of three polylines, which have coincident geometry in one location.

A coup1e of other things should be noted about this script.

First, the getPart method uses an index value of zero (0).

This means the method returns only the first part of the geometry object that has index va1ue 0.

For regular (that is, sing1e part) feature classes, the first part is also the only part.

If no index value is specified, the getPart method returns an array containing an array of point objects.

This is addressed in more detail in the next section, on multipart features.

Second, the script can be used for both po1yline and po1ygon feature classes.

## 8.4 Working with multipart features

Features in a feature class can have multiple parts, making them multipart features.

Such features are sometimes needed when there are multiple physical parts to a feature but only one set of attributes.

A classic example of a multipart feature is the state of Hawaii: each of the islands is its own part, but for Hawaii to be shown as a single record in the attribute table, these parts must form a single feature.

In the case of points, these features are referred to as multipoint, and in the case of polylines and polygons, they are referred to as multipart.

Whether a feature class is multipart can be determined using the shapeType property of the Describe function.

Valid return values for this property are Point, Polyline, Polygon , MultiPoint, and MultiPatch, which is used to represent three-dimensional data.

When a feature class is multipart, however, it does not mean that every feature in the feature class is multipart.

The isMultipart property of the geometry object is used to determine whether a particular feature is multipart.

The partCount property of the geometry object can be used to obtain the number of geometry parts of a feature.

The syntax for working with multipart geometries is very similar to the syntax for single-part features.

The key difference is that for multipart features, an array containing multiple arrays of point objects is returned, instead of the single array of point objects for single-part features.

A script working with geometry, therefore, has to iterate over not just the rows of a table (and over the array of point objects for polylines and polygons), but also over the array of parts for each geometry object.

The following example code illustrates how this is accomplished for polyline and polygon feature classes:

|  |
| --- |
|  |

This script works for both single-part and multipart features.

For single-part features, the number of parts is one (1) and the code block in the for loop is run only once for each geometry object.

This script works for polyline and polygon feature classes, as well as for single-part and multipart features.

When applied to the same roads shapefile as before, the output is as follows:

|  |
| --- |
|  |

The script prints the feature and part number (starting with index 0) followed by the vertices in each part.

For a single-part feature class, the part number will always be zero (0).

When the script is run on multipart features, however, the vertices in each part are printed separately.

For example, when the script is app lied to a feature class of the state of Hawaii where all the islands are part of a single multipart feature, the output is as follows.

|  |
| --- |
|  |

## 8.5 Working with polygons with holes

If a po1ygon contains holes, it will consist of a number of rings: one exterior ring and one or more interior rings.

A ring is a closed path that defines a two-dimensiona1 area.

A path is a series of vertices with a starting vertex (from) and an ending vertex (to).

A va1id ring consists of a va1id path in which the from and the to points of the ring have the same x,y coordinates.

An exterior ring is defined as a clockwise ring and an interior ring is defined as a counterclockwise ring.

For a po1ygon with holes, the geometry object returns an array of point objects that contains the points of the exterior ring and all the inner rings The exterior ring is a1ways returned first, followed by the inner rings.

A null point object-that is, a point object without va1ues-is used as the separator between rings.

A script to read the geometry of po1ygons with ho1es is very similar to the script in the preceding section for mu1tipart features.

The third for loop is rep1aced by the following:

|  |
| --- |
|  |

The only addition consists of an else statement.

For a geometry object with rings, null points are used as separators between rings.

Therefore, if the next object following a null point is a point object, it means there is an interior ring.

This block of code is run for every null point separator, resulting in the vertices of each interior ring being 1isted separate1y.

Polygons with holes are quite common, especially in feature classes that describe natural features , such as vegetation and soils.

The example in the figure illustrates a typical soil polygon.

Running the script on the soil polygon results in the following output.

There is only a single feature (Feature 0) that has an outer ring (Part 0) and multiple inner rings.

In this example, the rings are not numbered.

The output is as follows:

## 8.6 Writing geometries

New features can be created or updated using the insert and update cursors.

A script can define a feature by creating point objects, populating their properties, and placing the point objects in an array.

This new array can then be used to set the geometry of a feature.

For example, the following text file lists 21 points, each starting with an ID number and followed by an x-coordinate and a y-coordinate, with the coordinates of each pair separated by a space (" ").

The coordinates of these points will be used to create a new polygon-notice that the coordinates of the first point and the last point in the list are identical.

These coordinates are stored in a text (.txt) file called points.txt.

The text file reads as follows:

|  |
| --- |
|  |

The CreateFeatureclass function can be used to create a new, empty feature class, which will be used to hold the new point objects whose coordinates are taken from the preceding list.

The syntax of this tool is as follows:

|  |
| --- |
|  |

The only required parameters are the path for the location of the new feature class (folder or geodatabase) and the name of the new feature class.

The default value of the geometry is Polygon.

There is no default for the spatial reference, so if none is specified, the coordinate system will be "unknown."

The first part of the script is as follows:

|  |
| --- |
|  |

So far, this has created a new, empty feature class called newpoly.shp.

The point objects representing the vertices of the polygon can be created using the ArcPy Point class.

These point objects have to be stored in an array.

An array object can be created using the ArcPy Array class.

In general, an array can contain any number of geoprocessing objects such as points, geometries, or spatial references.

In this case, the array will contain point objects.

In addition, an insert cursor is created to make it possible to create new rows-that is, new features.

These lines of code are as follows:

|  |
| --- |
|  |

Next, the properties of the point objects have to be set using the values in the text file.

This requires the fileinput Python module to read the text file, and the split method to parse the text into separate strings for the point ID number, the x-coordinate, and the y-coordinate.

These lines of code are as follows:

|  |
| --- |
|  |

The split method returns a list of strings using the argument of the method as the delimiter.

When no argument is specified (as is the case here), the split method uses consecutive whitespace as the delimiter.

For each line, the split method returns a list of three strings.

These values are then assigned to ID, X, and Y.

Finally, the script needs to iterate over the lines of the input text file and create a point object for every line.

The result is a single array with 21 point objects.

The completed script is as follows:

|  |
| --- |
|  |

The result of the script is a new shapefile called newpoly.shp with a single polygon feature, as shown in the figure.

The example script is still relatively simple because it created only a single polygon with no other attributes.

However, it illustrates the concept of using the Point and Array classes to create new geometry objects.

## 8.7 Using cursors to set the spatial reference

The spatial reference for a feature class describes the coordinate system, the spatial domain, and the precision.

The spatial reference is typically set when the feature class is created.

However, since specifying a spatial reference is not required, it results in an unknown coordinate system when none is specified.

In this case, the Define Projection tool can be used to record the coordinate system information for the feature class.

A spatial reference app lies to all the features in a feature class.

By default, the spatial reference of the geometry of an object returned from a cursor is, therefore, the same as the spatial reference of the feature class opened by the cursor.

In certain circumstances, however, you may be working with geometries that have a different spatial reference from the feature class-for example, if you have a feature class in a state plane coordinate system and you want to insert new features using a text file that has universal transverse Mercator (UTM) coordinates.

In this case, you could set the spatial reference on the update or insert cursor to ensure proper con versions.

You would open an insert cursor on the feature class and set the spatial reference of the cursor to UTM, thus declaring that the geometries to be inserted need to be converted from UTM to state plane.

You can also set the spatial reference of a search cursor.

Specifying a spatial reference that is different from the spatial reference of the feature class results in geometries that are converted into the spatial reference of the cursor.

Consider the example of using a point feature class in state p lane coordinates and writing a script that exports the x, y coordinate pairs of the point objects in decimal degrees.

The SearchCursor function is used to establish a read-only cursor on the state plane coordinates of the feature class, but the spatial reference of this cursor is set to the desired geographic coordinate system, in decimal degrees.

This is accomplished using the following code:

|  |
| --- |
|  |

Next, an output file is created, using the open function.

This opens the file in writing mode (“w”) so that new lines of text can be written to it, follows:

|  |
| --- |
|  |

The next step is to iterate over the rows, create a geometry object for each row, and write the x, y coordinates to the output file using the write method.

This part of the code is as follows:

|  |
| --- |
|  |

The coordinates are written as decimal degrees in a string, with a space (" ") separating the coordinates of each pair, and with a line break ("\n") for each point object.

The very last step is to close the output file using the close method.

The complete code is as follows:

|  |
| --- |
|  |

## 8.8 Using geometry objects to work with geoprocessing tools

Inputs for geoprocessing tools often consist of feature classes.

Sometimes, however, these feature classes do not yet exist and need to be created from geometry information.

1n this case, you can create a new feature class, populate the feature class using cursors, and then use the feature class in geoprocessing tools.

This can become cumbersome, however.

As an alternative, geometry objects can be used instead of both input and output feature classes to make geoprocessing simpler.

For example, the following code creates a list of geometry objects from a list of coordinates, and then uses the geometry objects as input to the Buffer tool, as follows:

|  |
| --- |
|  |

In the example code, the geometry objects are created as a list of point objects.

First, an empty list is created using point1ist = [].

In the for loop, the list of coordinate pairs is used to create point objects using the Point class.

These point objects are then used by the PointGeometry class to create geometry objects, which are appended to the list.

The list becomes the input for the Buffer tool.

An alternative would be to first create a feature class based on the list of coordinates, but if this feature class is not necessary for anything else, the use of geometry objects will result in more efficient code.

Geometry objects can also be created directly as the output of geoprocessing tools.

For example, the following code uses an empty geometry object as the output of the Copy Features tool, and the result is a list of geometry objects, as follows:

# Chapter 9 Working with rasters

## 9.1 Introduction

Rasters present a unique type of spatial data, and a number of geoprocessing tools are designed specifically to take advantage of the raster data structure.

This chapter illustrates how to use regular ArcPy functions to list and describe rasters.

ArcPy also includes a Spatial Analyst module referred to as arcpy.sa, which fully integrates map algebra into the Python environment, making scripting much more efficient.

Map algebra operators are described, as well as functions and classes of the arcpy.sa module.

## 9.2 Listing rasters

The ListRasters function returns a Python list of rasters in a workspace.

The syntax of the function is:

|  |
| --- |
|  |

An optional wild\_card parameter can be used to limit the list based on the name of the rasters.

The optional raster\_type parameter can be used to limit the list based on the type of raster-for example, JPEG or TIFF.

The following code illustrates the use of the ListRasters function to print a list of the rasters in a workspace:

|  |
| --- |
|  |

The name of each raster is printed to the Interactive Window in PythonWin or the next line in the Python window, along with an optional file extension.

For example, it is .img for the ERDAS IMAGINE format, .tif for the TIFF format, .jpg for the JPEG format, and so on.

No file extensions are added for the Esri GRID (global resource information database) format or for rasters stored inside a geodatabase.

Therefore, when no file extension is present, be sure to determine whether you are working with a GRID or with a raster dataset inside a geodatabase.

The parameters of the ListRasters function can be used to filter the results.

For example, the following code prints a list of the rasters in the workspace that are in the ERDAS IMAGINE format:

|  |
| --- |
|  |

Once the names of the rasters are obtained, other functions can be used, including functions to describe the data as discussed in the next section.

## 9.3 Describing raster propeies

The Describe function returns the properties for a specified data element.

These properties are dynamic, which means the properties that are present depend on the data type being described.

For example, when the Describe function is used on rasters, a generic set of properties is present in addition to specific properties that are unique to the specific raster element.

Three different raster data elements can be distinguished:

1. Raster dataset

- a raster spatial data model that is stored on disk or in a geodatabase.

Raster datasets can be stored in many formats, including TIFF, JPEG , IMAGINE , Esri GRID , and MrSID.

Raster datasets can be single band or multiband.

2. Raster band

-one layer in a raster dataset that represents data values for a specific range in the electromagnetic spectrum or other values derived by manipulating the original image bands.

Many types of satellite images, for example, contain multiple bands.

3. Raster catalog

-a collection of raster datasets defined in a table of any format, in which the records define the individual raster data sets that are included in the catalog.

Raster catalogs can be used to display adjacent or overlapping raster datasets without having to combine them into a mosaic in one large file

Properties for each of these elements vary.

For example, the format (TIFF, JPEG, and others) is a property of the raster dataset and the cell size is a property of the raster band.

The general dataType property can be used to determine the type of data element.

All properties, however, are accessed using the same Describe function.

The following code illustrates the use of the Describe function, which returns an object with properties that can be accessed, in this case for printing.

|  |
| --- |
|  |

For this example of a raster in TIFF format, the dataType property returns the type RasterDataset.

Properties that are specific to raster datasets only include the following:

bandCount - the number of bands in the raster dataset.

compressionType - the compression type (LZ77 , JPEG, JPEG2000 , or None)

format - the raster format (GRID , IMAGINE, TIFF, and more)

permanent - indicates the permanent state of the raster: False if the raster is temporary, True if the raster is permanent

sensorType -the sensor type used to capture the image

Once it has been determined that an element is a raster dataset, these properties can be accessed.

For example, the following code includes additional properties used to describe the TIFF file.

|  |
| --- |
|  |

This particular .tif file is a single-band uncompressed TIFF, and therefore the property bandCount returns a value of 1 and compressionType returns a value of None .

For single-band raster datasets, the band itself does not have to be specified (there is only one, after all) and the properties can be accessed directly by describing the raster dataset.

For example, the following code determines the cell size and pixel type of a raster:

|  |
| --- |
|  |

For this particular example, the code returns values of 30.0 by 30.0 and U8 - this means the cell size is 30 by 30 meters and the pixel type is an unsigned 8-bit integer.

These properties do not report the unit type, which has to be obtained from the Spatial Reference property.

For example, the following code determines the name of the spatial reference and the unit:

|  |
| --- |
|  |

For multiband rasters, however, the specific band needs to be specified.

Without a particular band being specified, properties such as cell size, height, width, and pixel type cannot be accessed.

Specific bands are referenced using Band\_1 , Band\_2, and so on.

The following code illustrates h ow the properties for a band in a multiband raster dataset are accessed:

## 9.4 Working with raster objects

ArcPy also contains a Raster class that is used to reference a raster dataset.

A raster object can be created in two ways: (1) by referencing an existing raster on disk and (2) by using a map algebra statement.

The syntax for the Raster class is

The following code illustrates how to create a raster object by referencing a raster on disk:

When using map algebra statements, the code looks something like the following:

|  |
| --- |
|  |

In both cases, the resulting raster object can be used in Python statements and additional map algebra expressions.

Raster objects have many properties, which are largely similar to those already discussed earlier in this chapter, including bandCount, compressonType, format, height , width, meanCellHeight , meanCellWidth, pixelType, spatialReference, and others.

Similar to the Describe function, these properties are mostly read-only.

Raster objects have only one method: save.

The raster object (the variab1e and associated dataset) returned from a map a1gebra expression is temporary by default.

This means the variable and the referenced dataset are de1eted when the variab1e goes out of scope-for example, when ArcGIS is closed or when a stand-alone script is closed.

The save method can be called to make the raster object permanent.

The syntax of the save method is

In the earlier example, the raster object outraster is temporary but can be made permanent using the following code:

It may appear somewhat counterintuitive that map algebra expressions result in temporary outputs.

Keep in mind that a typical workflow using rasters can involve numerous steps.

If only the final output is actually needed, keeping temporary outputs as intermediate steps results in fewer output files and lower storage requirements.

## 9.5 Working with the ArcPy Spatial Analyst module

ArcPy includes a Spatial Analyst module, arcpy.sa, to carry out map algebra and other operations.

The functionality provided by the Spatial Analyst module is largely the same as that of the tools in the Spatial Analyst toolbox.

For example, you can run the Slope tool by referencing the Slope tool in the Spatial Analyst toolbox or by importing the arcpy.sa module and directly referencing the Slope tool.

The Spatial Analyst module integrates map algebra into the Python environment.

This is similar to the use of map algebra in such ArcToolbox geoprocessing tools as Raster Calculator, Single Output Map Algebra, and Multiple Output Map Algebra in earlier versions of ArcGIS.

The ArcPy Spatial Analyst module has a series of operators to support map algebra operations.

The Spatial Analyst module provides access to all the raster geoprocessing tools in the Spatial Analyst toolbox.

It offers an alternative way to run these tools that can be more efficient than running them using the Spatial Analyst toolbox.

Consider the following code that runs the Slope tool:

Notice that the Slope tool is called using arcpy.sa.Slope, which appears to follow the regular syntax used for all tools: arcpy.<toolboxalias>.<toolname>.

However, the alternative arcpy.<toolname>\_<toolboxalias> syntax does not apply here, and arcpy.Slope\_sa is not valid.

Because sa is a module, and not just the alias of a toolbox, the code can be simplified as follows:

The statement from arcpy.sa import \* imports all the functions from the arcpy.sa module, and tools can therefore be called directly-for example, Slope versus arcpy.sa.Slope.

Initially, this may not appear to be much of a saving, but imagine having several dozen raster functions in a single map a1gebra expression-omitting arcpy.sa several dozen times makes your code shorter and easier to read.

## 9.6 Using map algebra operators

In addition to providing access to all the Spatia1 Analyst geoprocessing tools, the arcpy.sa module also includes a number of map algebra operators.

Most of these operators are available as geoprocessing too1s under the Math toolset in the SpatialAna1yst toolbox yet are also available as operators in Python.

Consider the following example, which converts e1evation values from feet to meters using the Times too1:

Instead of using the Times tool, the map algebra operator (\*) can be used.

The second-to-last line of code would look as follows:

This alternative is a bit shorter, but more importantly, it make it possible to write elaborate map algebra expressions relatively easily.

Consider the example of a suitability model in which you create three different rasters, each representing a different factor in the suitability model.

In the final analysis step, you want to add these three rasters together and determine the average suitability score.

Your code could look something like the following:

The Plus tool has to be used twice to add all three rasters together because the tool can use only two inputs at a time.

The Divide tool is used to divide the sum of the three rasters by 3.

Using map algebra expressions, this code can be reduced as follows:

It looks very much like the Python code used earlier.

In effect, the map algebra operators in the arcpy.sa module allow you to create Raster Calculator-style expressions directly in Python.

You can also call the Raster Calculator tool using the following syntax:

## 9.7 Using the ApplyEnvironment function

In addition to the geoprocessing tools in the Spatial Analyst toolbox, there is one more function: the ApplyEnvironment function.

This function copies an existing raster and applies the current environment settings.

The syntax of the function is:

This function allows you to change things like the extent or the cell size or to apply an analysis mask.

The following code illustrates how the ApplyEnvironment function is used to set a new cell size of 30 and apply an analysis mask based on an existing shape file:

Not all environment settings apply to the ApplyEnvironment function.

They are limited to the following: Cell Size, Current Workspace, Extent, Mask, Output Coordinate System, Scratch Workspace, and Snap Raster These are the most relevant environment settings when working with rasters.

## 9.8 Using classes of the arcpy. sa module

The arcpy.sa module a1so contains a number of classes that are used for defining parameters of raster tools.

Typically, these classes are used as shortcuts for tool parameters that wou1d otherwise require a more complicated string value.

Consider the example of the Reclassification tool.

With this tool, raster cells are given a new value based on a reclassification table.

The tool dialog box in the figure shows an examp1e of a land-use raster being reclassified into a number of va1ues as part of a suitability model.

Typing all the va1ues of this table would be rather complicated since this table can have many different entries.

Instead, the remap parameter is expressed as a remap object.

There are two different Remap classes, depending on the nature of the reclassification:

The syntax of the RemapValue object is

A remapTable object is defined using a Python of lists that each contain old and new values, similar to the reclassification table on the tool dialog box.

The syntax of a remap table for use in a RemapValue object is

The following code illustrates the use of a remap object to carry out a reclassification of a raster representing land use:

The RemapRange object works in a similar manner but uses value ranges rather than individual values.

The syntax of a remap table for use in a RemapRange object is

The following code illustrates the use of a remap object to carry out a reclassification of a raster of elevation:

Notice that the end value of the first range is the same as the start va1ue of the second range, and so on.

This type of remap table is common when data is continuous, as in the case of a raster of elevation.

In addition to the Reclassify tool, remap tables are a1so used in the Weighted Overlay tool.

There are many other classes in the arcpy.sa module.

They can be grouped into a number of categories based on logical functionality.

Table 9.2 lists these categories.

Among the more widely used classes, in addition to the Remap classes already discussed, are the Neighborhood classes, which define neighborhoods of different shapes and sizes.

Consider, for example, the Focal Statistics tool.

This tool, as well as other tools in the Neighborhood toolbox, requires the definition of a specific neighborhood.

The neighborhood settings vary with the type of neighborhood.

For example, for the default rectangular neighborhood, settings include height and width in cell or map units.

However, for the wedge neighborhood, the parameters include start angle and end angle and a radius in cell or map units.

Because of the variability of these parameters, neighborhood functions include a neighborhood object.

For example, the syntax of the Focal Statistics tool is as follows:

For example, the syntax of the NbrRectangle object is

The following code defines a neighborhood object and uses it in the FocalStatistcs function:

In this example, the output is a raster of land cover based on a rectangular neighborhood of 5 cells by 5 cells.

## 9.9 Using raster functions to work with NumPy arrays

Two more raster functions need to be mentioned: NumPyArrayToRaster and RasterToNumPyArray .

These are regular ArcPy functions, not functions of the arcpy.sa module.

These two functions allow for conversions between rasters and NumPy arrays.

A NumPy array is designed to work with very large arrays.

NumPy itself is a package used for scientific computing with Python.

Among other things, it provides a very powerful n-dimensional array object.

This type of object makes it possib1e to move data between databases.

For example, the SciPy package contains numerous algorithms that may be useful for a particular application, such as Fourier transforms, maximum entropy models, and multidimensiona1 image processing.

Rather than trying to build a tool in ArcGIS that carries out these specialized functions, you could write a script tool that converts a raster to a NumPy array, and then calls specialized functions from the SciPy package.

A generic script would look as follows:

This is a simplified example and references a generic SciPy function, yet it illustrates how NumPy array functions can be used to export data for processing in another environment and to import the result back into an ArcGIS-compatible format-all within the same Python script.

More information on NumPy (Numerical Python) and SciPy (Scientific Library for Python) can be found at http://numpy.scipy.org and http://www.scipy.org.respectively