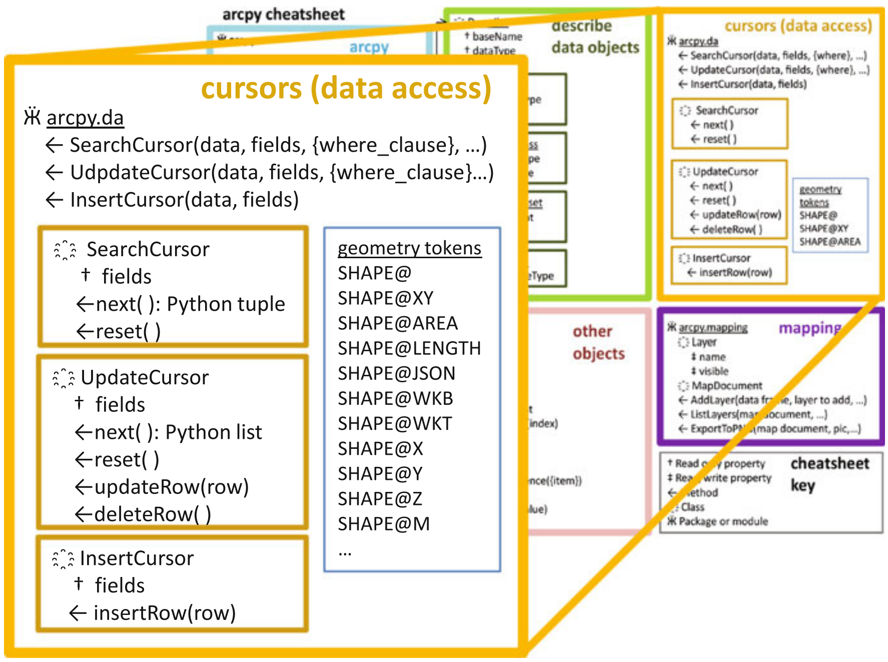
Chapter 17
Reading and Writing with Cursors

Abstract Batch processing can be extended to work with individual records within GIS data tables. Previous chapters listed fields in multiple files within nested directories; this chapter adds another dimension to data exploration. Each data table is made of *records*, the rows of values in the attribute table, which can also be listed within a script. Though the term 'records' may be more familiar to database users, we'll use the term 'rows' to refer to data records for consistency with the `arcpy` documentation. Cursors and file input/output techniques, enable reading and writing individual rows within data tables. The focus of this chapter is `arcpy` cursor functions, which work with the rows in Esri attribute tables, such as feature class tables.



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

- Read and modify GIS attribute table files.
- Query the data for a select set of table rows.
- Use error handling in cursor scripts.
- Prevent data locking errors.
- Update or insert field values.
- Insert point and polygon data.

17.1 Introduction to Cursors

GIS attribute tables are in proprietary formats not accessible with generic Python text file reading methods. `arcpy` cursors are objects designed specifically to accommodate GIS tables. You can create a cursor by calling an arcpy function and specifying a table. Then, depending on the type of cursor you requested, it can do one or more of the following:

- Read attribute values in rows.
- Modify attribute values in rows.
- Delete rows.
- Insert new rows.

For example, with cursors, you could filter to find the location of last year's forest fires, you could increment the walkability index for homes with average commute times less than 15 minutes, you could delete the records in a parcel dataset that have a blank land use value, or add new emergency call records to a log. The `arcpy` package provides three flavors of cursors to handle different functionalities:

- Search cursors for reading.
- Update cursors for updating existing rows and deleting rows.
- Insert cursors for inserting new rows.

The 10.1 and higher `arcpy` package provides both classic cursor functions and newer data access cursor functions. The classic cursors are the original implementation. The classic cursor function signatures are shown in Table 17.1. If you are using an earlier version of ArcGIS, you'll need to use this syntax. These are still available in newer versions, but a new set of cursor functions have been added.

Starting with ArcGIS 10.1, there is a data access module (da) which also has `SearchCursor`, `UpdateCursor`, and `InsertCursor` functions. Double dot

Table 17.1 Classic cursor signatures.

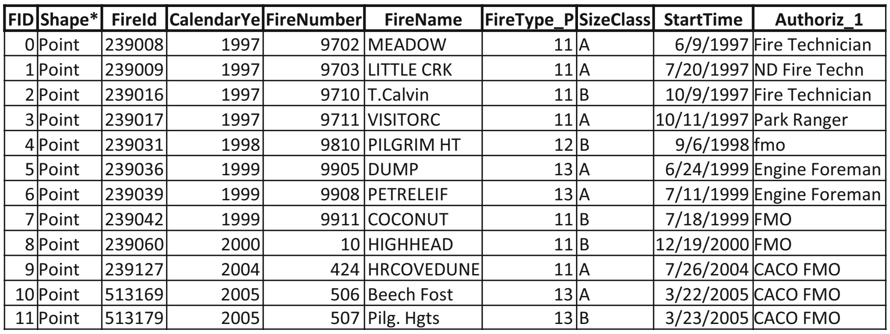
`arcpy.SearchCursor(dataset, {where_clause}, {spatial_reference}, {fields}, {sort_fields})`
`arcpy.UpdateCursor(dataset, {where_clause}, {spatial_reference}, {fields}, {sort_fields})`
`arcpy.InsertCursor(dataset, {spatial_reference})`

Table 17.2 Data access cursor signatures.

`arcpy.da.SearchCursor(in_table, field_names, {where_clause}, {spatial_reference}, {explode_to_points}, {sql_clause})`
`arcpy.da.UpdateCursor(in_table, field_names, {where_clause}, {spatial_reference}, {explode_to_points}, {sql_clause})`
`arcpy.da.InsertCursor(in_table, field_names)`

notation is used to call these functions within the da module. Table 17.2 shows their signatures. Notice the differences between the classic and data access SearchCursor signatures; Single dot versus double dot and one required argument versus two are some of the obvious differences. Both sets of functions return a cursor object which allows you to iterate through the rows. However, the data access cursors are significantly faster than the classic cursors, so this chapter focuses on the data access cursors.

Cursors are quite useful, but proceed with caution; Cursors lock the data while they are operating, meaning other ArcGIS Desktop processes cannot simultaneously delete or otherwise modify the data. Data cannot be previewed while edits are being made with update or insert cursors. If necessary, make a copy of the data and view this table instead of the original during cursor operations. Scripts need to delete cursor objects for locks to be released. Also, update and insert cursors can modify your data, so backup data before testing cursor code. This chapter begins with search cursors, to illustrate basic cursor object syntax.



FID	Shape*	FireId	CalendarYe	FireNumber	FireName	FireType_P	SizeClass	StartTime	Authoriz_1
0	Point	239008	1997	9702	MEADOW	11	A	6/9/1997	Fire Technician
1	Point	239009	1997	9703	LITTLE CRK	11	A	7/20/1997	ND Fire Techn
2	Point	239016	1997	9710	T.Calvin	11	B	10/9/1997	Fire Technician
3	Point	239017	1997	9711	VISITORC	11	A	10/11/1997	Park Ranger
4	Point	239031	1998	9810	PILGRIM HT	12	B	9/6/1998	fmo
5	Point	239036	1999	9905	DUMP	13	A	6/24/1999	Engine Foreman
6	Point	239039	1999	9908	PETRELEIF	13	A	7/11/1999	Engine Foreman
7	Point	239042	1999	9911	COCONUT	11	B	7/18/1999	FMO
8	Point	239060	2000	10	HIGHHEAD	11	B	12/19/2000	FMO
9	Point	239127	2004	424	HRCOVEDUNE	11	A	7/26/2004	CACO FMO
10	Point	513169	2005	506	Beech Fost	13	A	3/22/2005	CACO FMO
11	Point	513179	2005	507	Pilg. Hgts	13	B	3/23/2005	CACO FMO

Figure 17.1 fires.shp table contains 11 wildfire records.

17.2 Reading Rows

The data access SearchCursor function has two required arguments: the input table and a list of field names. This function returns a search cursor object. The cursor object is used to access the rows. The following code creates a search cursor for the 'FID', 'FireId', and 'FireName' fields in the table shown in Figure 17.1:

```
>>> import arcpy
>>> # Create a cursor
>>> fc = 'C:/gispy/data/ch17/fires.shp'
>>> cursor = arcpy.da.SearchCursor fc, ['FID', 'FireId', 'FireName'])
```

The cursor object has two methods `next` and `reset` and one property `fields`. Rows can be extracted from the cursor object using the `next` method or using `FOR`-loops. The `next` method gets the next row in the table. This provides a convenient way to examine row contents interactively as you become familiar with cursors. The following code uses the `next` method to get a row and prints the row:

```
>>> row = cursor.next() # Get an individual row.
>>> row
(0, 239008.0, u'MEADOW')
```

0, 239008.0, and MEADOW are the values for 'FID', 'FireId', and 'FireName' in the first row of the table in Figure 17.1. The `row` variable is a Python tuple that can be indexed to get individual field values. These indices correspond to the order in which the field names are passed into the cursor. For example, the 'FID' index is zero, but the 'FireId' index is 1, even though it is the third column in the attribute table, and the 'FireName' index is 2, even though it is the sixth column in the table. The indices are 0, 1, and 2 because a list of three field names was used to create the cursor.

```
>>> row[0] # FID
0
>>> row[1] # FireId
239008.0
>>> row[2] # FireName
u'MEADOW'
```

The `next` method gets the first row the first time it is called; Calling it again gets the second row and calling it again gets the third row:

```
>>> row = cursor.next()     # Get the second row
>>> row
(1, 239009.0, u'LITTLE CRK')
>>> row = cursor.next()     # Get the third row
>>> row
(2, 239016.0, u'T.Calvin')
```

The cursor `reset` method brings the pointer back to the first row.

```
>>> cursor.reset()
>>> row = cursor.next()
>>> row
(0, 239008.0, u'MEADOW')
```

The `fields` property returns a tuple containing the field names, as shown in the following code:

```
>>> fds = cursor.fields
>>> fds
(u'FID', u'FireId', u'FireName')
```

Notice that `fields` is not followed by parentheses because it's a cursor property; Whereas, `next` and `reset` require parentheses because they are methods. Omitting the parentheses will not throw an exception, but it can lead to other perplexing errors as shown in the following code:

```
>>> row = sc.next    # Missing parentheses!
>>> row
<method-wrapper 'next' of da.SearchCursor object at 0x10CF8700>
>>> row[0]
Traceback (most recent call last):
File '<interactive input>', line 1, in <module>
TypeError: 'method-wrapper' object is not subscriptable
```

17.3 The Field Names Parameter

The field names can be specified in any order within the `field_names` parameter list. As mentioned earlier, the row index for that field depends on the order in which they are specified, not the order of the fields in the attribute table. Compare the field name ordering in the following code to the order of the fields in the attribute table shown in Figure 17.1:

```
>>> # Create a cursor
>>> fc = 'C:/gispy/data/ch17/fires.shp'
>>> cursor = arcpy.da.SearchCursor(fc, ['FireName', 'FID'])
>>> row = cursor.next()
>>> row
(u'MEADOW', 0)
>>> row[0]
u'MEADOW'
>>> row[1]
0
```

The classic `arcpy` cursors do not require a field name list; These cursors always contain all fields. Data access cursors can be made to simulate this behavior. Using a string asterisk (`'*'`) for the field name parameter obtains all fields. The drawback is that this may affect performance for large files.

```
>>> # Create a cursor with access to ALL the fields.
>>> cursor = arcpy.da.SearchCursor(fc, '*')
>>> cursor.fields
(u'FID', u'Shape', u'FireId', u'CalendarYe', u'FireNumber',
u'FireName', u'FireType_P', u'SizeClass', u'StartTime',
u'Authoriz_1')
```

In case the field names are not hard-coded in the script, the field names can be derived using the `ListFields` method. Example 17.1 gets a subset of the fields based on the field type. This script lists all the fields, then calls a user-defined function that returns a list of field names for all fields except those that are 'Geometry' or 'OID' type data. OID stands for object identifier, the unique key for each record created automatically when a table is created. Example 17.1 uses a data access cursor without hard-coding the field names. The last line of code in Example 17.1 uses the `del` key word to delete the cursor. We'll look at what this does and why this is important shortly.

Example 17.1: Print data values with unknown field names.

```
# printTableExclude.py
# Purpose: Print the names of the non-geometry non-OID type
#          fields in the input file and the value of these
#          fields for each record.
# Usage: No script arguments required.
import arcpy

def excludeFields(table, types=[]):
    '''Return a list of fields minus those with
       specified field types'''
    fieldNames = []
    fds = arcpy.ListFields(table)
    for f in fds:
        if f.type not in types:
            fieldNames.append(f.name)
    return fieldNames

fc = 'C:/gispy/data/ch17/fires.shp'
excludeTypes = ['Geometry', 'OID']
fields = excludeFields(fc, excludeTypes)

with arcpy.da.SearchCursor(fc, fields) as cursor:
    print cursor.fields
    for row in cursor:
        print row
del cursor
```

17.4 The Shape Field and Geometry Tokens

Cursors not only provide access to the row values you see in the attribute table of a file, they can also access each feature's geometric information. When a feature class attribute table is viewed in ArcGIS, the 'Shape' column shows a shape type, such as 'Polygon'. However, more information about each feature is stored internally for visualization and geoprocessing operations. This information can be accessed through `arcpy Geometry` objects. Notice that the field 'type' of the 'Shape' field is a 'Geometry'.

```
>>> fds = arcpy.ListFields('C:/gispy/data/ch17/park.shp')
>>> [f.name for f in fds]
[u'FID', u'Shape', u'COVER', u'RECNO']
>>> [f.type for f in fds]
[u'OID', u'Geometry', u'String', u'Double']
```

The `Geometry` objects have properties to do with geometric features (Figure 17.2). Some properties are intuitive, such as `area`, `length`, `isMultipart`

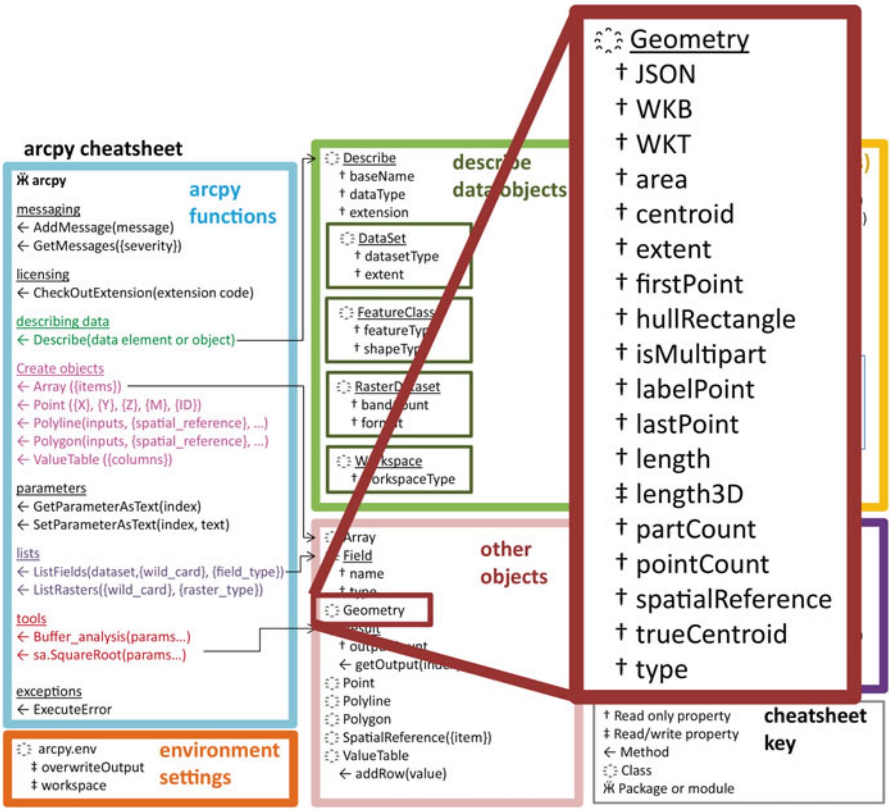


Figure 17.2 Selected `Geometry` object properties.

and so forth; Others, such as WKT (for well-known text, a portable geometry format) may be less familiar. Search the ArcGIS Resources site for ‘Geometry (arcpy)’ to see an explanation of each property. Previously, we used the Geometry object in field calculation expressions, as in the following code:

```
>>> data = 'C:/gispy/data/ch17/special_regions.shp'
>>> fieldName = 'PolyArea'
>>> expr = '!shape.area!'
>>> arcpy.CalculateField_management(data, fieldName, expr, 'PYTHON')
```

Cursor notation for Geometry objects differs from the field calculator notation. The following code creates a cursor for polygon areas and prints the area of the first polygon feature:

```
>>> parkData = 'C:/gispy/data/ch17/parks.shp'
>>> cursor = arcpy.da.SearchCursor (parkData, ['SHAPE@AREA'])
>>> row = cursor.next()
>>> row[0]
600937.0921638225
```

SHAPE@AREA is a special expression known as a *geometry token.* Geometry token strings start with SHAPE@ and can optionally include a geometry property suffix. Other examples of geometry tokens include SHAPE@, SHAPE@XY, and SHAPE@LENGTH. The general token is SHAPE@, which provides access to all available geographic object properties. The general token returns a geometry object; Use dot notation to access the geometry properties. The following code shows some examples of working with the geometric properties returned by the SHAPE@ token:

```
>>> cursor = arcpy.da.SearchCursor (parkData, ['SHAPE@'])
>>> row = cursor.next()
>>> row[0].type
'polygon'
>>> row[0].area
600937.0921638225
>>> row[0].centroid
<Point (2131483.24062, 191220.538898, #, #)>
>>> row[0].firstPoint
<Point (2131312.35816, 190450.120062, #, #)>
>>> row[0].area
600937.0921638225
```

These are called tokens because they represent the Geometry object. Using the 'Shape' field without the @ symbol, only produces a tuple with x and y coordinates of the feature centroids as shown in the following code:

```
>>>> cursor = arcpy.da.SearchCursor (parkData, ['SHAPE'])
>>> row = cursor.next()
>>> row[0].centroid
Traceback (most recent call last):
File '<interactive input>', line 1, in <module>
AttributeError: 'tuple' object has no attribute 'centroid'
>>> row[0]
(2131483.240622718, 191220.5388983136)
```

17.5 Looping with Cursors

Repeatedly calling the next method yields each subsequent row until the rows are exhausted. However, most search and update cursor scripts use looping. If the intention of the script is to use all rows, a FOR-loop should be used instead of the next method. The cursor is not a Python list, but is an *iterable* object—an object which is capable of returning its members one at a time. In other words, you can use it as the sequence in a FOR-loop just like looping on a list. The iterating variable gets the row tuples as it loops. The following code loops through the rows of the fire table and prints each FireName:

```
import arcpy
fc = 'C:/gispy/data/ch17/fires.shp'
fields = ['FireName']

cursor = arcpy.da.SearchCursor(fc, fields)
for row in cursor:
    print row[0]

del cursor
```

The del keyword deletes the cursor after the loop. This is necessary to prevent the dataset from becoming locked by the cursor object.

17.6 Locking

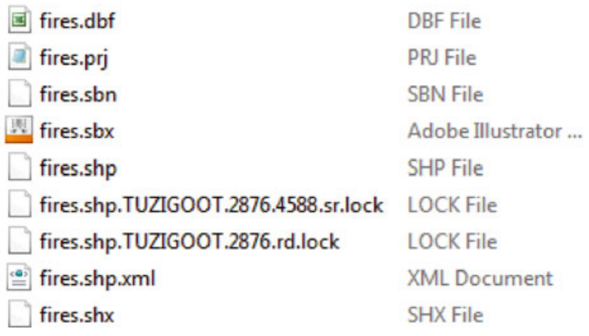
Locking is a mechanism to ensure that two processes don't modify the same dataset at once. You may have noticed 'LOCK' files in your data directories. Computer processes such as ArcCatalog or PythonWin create these locks so that they can perform operations on the data without risking data corruption. The operations that can be performed on the data depend on the type of locks placed on it.

Computers use two types of locks, exclusive and shared locks. These types of locks can be defined by how they restrict other processes while they are in place. If a process has an *exclusive lock* on the data, it won't allow another process to lock the data. If a process has a *shared lock* on data, it will allow other processes to obtain shared locks on the data, but it won't allow exclusive locks to be obtained for this data.

You can think of the data as a shared resource like a kitchen in a hostel. If someone is making a large meal for dinner party guests, he may ask that no one else use the kitchen that evening and the other house members would have to find elsewhere to eat dinner (an exclusive lock). Whereas, everyone will come and go in the mornings, so that some people may be preparing breakfast at the same time (a shared lock). It is against house rules to eject others from the kitchen and take over the room while others are preparing bowls of oatmeal (no one can claim an exclusive lock on the kitchen while a shared lock exists).

IDEs need to get an exclusive lock on data to iterate through it with update or insert cursors. Imagine a script in PythonWin iterating through a large file with an update cursor. A copy of the same script run simultaneously in PyScripter won't be able to use an insert cursor on this file, because PythonWin has an exclusive lock on the data.

Search cursors require shared locks. As an example, data can be viewed in ArcMap while a search cursor is iterating through its rows, but the lock will not allow the data to be deleted in ArcCatalog while the search cursor is active because deletion requires an exclusive lock. The following screen shot of Windows Explorer shows two 'LOCK' files associated with the 'fire' shapefile:



fires.dbf	DBF File
fires.prj	PRJ File
fires.sbn	SBN File
fires.sbx	Adobe Illustrator ...
fires.shp	SHP File
fires.shp.TUZIGOOT.2876.4588.sr.lock	LOCK File
fires.shp.TUZIGOOT.2876.rd.lock	LOCK File
fires.shp.xml	XML Document
fires.shx	SHX File

Use the search cursor `next` method in the Interactive Window to see these locks appear in the data list. `arcpy` cursors can create schema, read, and write locks. The schema and read locks are shared locks and write locks are exclusive locks. Once a cursor obtains a lock, the lock persists until the cursor is released. Lock release behavior depends where the code is being run. The best way to ensure that locks are released is to delete the cursor object after use.

17.6.1 The del Statement

The `del` Python keyword deletes objects from memory. The following code deletes the cursor object and releases all locks:

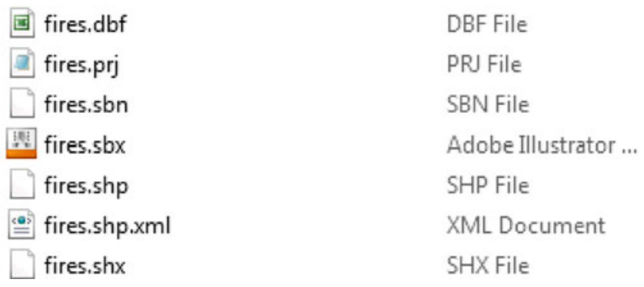
```
>>> del cursor
```

Run the following code at the prompt while watching the 'fires*' files in the 'ch17' data directory in Windows Explorer to see a lock file appear:

```
>>> fc = 'C:/gispy/data/ch17/fires.shp'
>>> cursor = arcpy.da.SearchCursor (fc, ['FireName'])
>>> for row in cursor:
...     print row[0]
```

Next, run the following code and observe the lock file disappear from the 'ch17' directory:

```
>>> del cursor
```



fires.dbf	DBF File
fires.prj	PRJ File
fires.sbn	SBN File
fires.sbx	Adobe Illustrator ...
fires.shp	SHP File
fires.shp.xml	XML Document
fires.shx	SHX File

Deleting the cursor explicitly deletes the schema, read, or write locks created by the cursor. Delete each cursor you create in a script to make sure that locks don't linger. The deletion needs to be done carefully. If something causes a script to crash, the locks might not be released. For example, the following code throws an `IndexError` before reaching the deletion statement:

```
cursor = arcpy.da.SearchCursor (fc, ['FireName'])
for row in cursor:
    # Try to index a second field, but there is none.
    print row[1]
del cursor
```

Using the deletion statement in combination with error handling ensures the cursor is deleted if an error occurs in the script. Example 17.2 illustrates this, placing a cursor deletion statement in the except block.

Example 17.2: Using try/except blocks with cursor looping.

```
# searchNprint.py
# Purpose: Print each fire name in a file.
import arcpy, traceback
try:
    cursor = arcpy.da.SearchCursor (fc, ['FireName'])
    for row in cursor:
        print row[0]
    del cursor
except:
    print 'An error occurred'
    traceback.print_exc()
    del cursor
```

17.6.2 The with Statement

The ESRI help describes a second technique for writing cursor scripts to release locks using the Python `with` and `as` keywords. When scripts are run within the ArcGIS Python Window, the `with` block automatically deletes data locks when the code exits the `with` block, even if there is an exception. Though this technique also works for `arcpy` cursors when code is run inside the ArcGIS Python window, this does not work for stand-alone scripts. The format for using a `with` block for an `arcpy` cursor is as follows:

```
with arcpy_cursor_function as cursor:
    code statement(s) using the cursor
```

If you use this technique to run stand-alone script in IDEs, such as PythonWin or PyScripter, locks are not guaranteed to be released this way. Since this doesn't work for stand-alone scripts, most `arcpy` cursor code should use error handling with try/except blocks and delete the cursor explicitly, instead of using the `with` block. We'll revisit this structure in Chapter 19, as it is useful for text file reading and writing.

Note Though the cursor examples in the `arcpy` documentation use the `with` statement, stand-alone scripts should use error handling and delete the cursor, as in Example 17.2.

17.7 Update Cursors

Update cursors are not only able to read field values, but can also modify them. Update cursors share some commonalities with search cursors. They have the same required and optional parameters (See Table 17.2). They can both be used in a loop and in a `with` block. Like the search cursor, the update cursor has `next` and `reset` methods (See Table 17.3). The `next` method returns a list instead of a tuple. Recall that Python lists are mutable, unlike Python tuples. Using a Python list during update operations makes it possible to modify individual elements of the row. Update cursors additionally have `updateRow` and `deleteRow` methods for modifying records. Examples 17.3 and 17.4 demonstrate these methods.

Example 17.3 increments the `FireType_P` by 2 and standardizes the `FireName` capitalization scheme. It performs multiple updates within the same loop by modifying the corresponding row elements. Notice that there are two steps to updating a table row. The first step changes the value of the row list. This just changes the value of a Python variable, but this does not change the value in the table. The second step changes the table value with the `updateRow` method.

Example 17.3: Perform more than one update.

```
# updateValues.py
# Purpose: Modify the fire type value and the fire
#          name in each record.
# Usage: No script arguments needed.
import arcpy, traceback
fc = 'C:/gispy/data/ch17/firesCopy.shp'
fields = ['FireType_P', 'FireName']
cursor = arcpy.da.UpdateCursor(fc, fields)
try:
    for row in cursor:
        # Make changes to the list of values in 'row'
        # Example: 13->15
        row[0] = row[0] + 2
        # Example: LITTLE CRK->Little Crk
        row[1] = row[1].title()
        # Update the table (otherwise changes won't be saved)
        cursor.updateRow(row)
        print 'Updated {0} and {1}'.format(row[0], row[1])
except:
    print 'An error occurred'
    traceback.print_exc()
del cursor
```

Table 17.3 Data access cursor methods and properties.

Cursor type	Methods	Properties

Search cursor	`next() # Returns a tuple`
	`reset()`
		Fields

Update cursor	`next() # Returns a list`
	`reset()`
	`updateRow(row)`
	`deleteRow()`
		Fields

Insert cursor	`insertRow()`	Fields

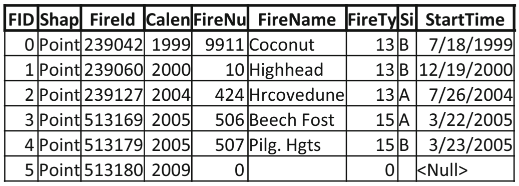
Example 17.4, deletes the first seven rows. The file's object identifier field (`FID`) is updated automatically in the remaining rows. When the deletions occur, the `FID` values are automatically renumbered to be contiguous starting with zero.

Example 17.4: Delete rows.

```
# deleteRows.py
# Purpose: Delete the first x rows.
# Usage: No script arguments required.
import arcpy, traceback
fc = 'C:/gispy/data/ch17/firesCopy.shp'
field = 'FID'
x = 7
try:
    cursor = arcpy.da.UpdateCursor(fc, [field])
    # Delete the first x rows
    for row in cursor:
        if row[0] < x:
            # Delete this row
            cursor.deleteRow()
            print 'Deleted row {0}'.format(row[0])
    del cursor
except:
    print 'An error occurred.'
    traceback.print_exc()
    del cursor
```

17.8 Insert Cursors

Use insert cursors to insert new rows. Like the other cursor functions, the `InsertCursor` method requires two arguments, an `in_table` and a `field_names` list. The code should create the cursor with `field_names` set to list the fields that you want to specify in the new row. Not all fields need to be specified. Fields with no values specified will be assigned a default value. Then there are two



FID	Shap	FireId	Calen	FireNu	FireName	FireTy	Si	StartTime
0	Point	239042	1999	9911	Coconut	13	B	7/18/1999
1	Point	239060	2000	10	Highhead	13	B	12/19/2000
2	Point	239127	2004	424	Hrcovedune	13	A	7/26/2004
3	Point	513169	2005	506	Beech Fost	15	A	3/22/2005
4	Point	513179	2005	507	Pilg. Hgts	15	B	3/23/2005
5	Point	513180	2009	0		0		<Null>

Figure 17.3 New row added by 'insertRow.py'.

steps to inserting a new row. First, create a Python list of the values to insert in the row. The list should be the same length as the `field_names` parameter list and specify the value for each field in the order of this list. Second, to insert the row, call the `insertRow` method. After the insertion, the cursor must be deleted to release locks. The code in Example 17.5 creates the last row in Figure 17.3.

Example 17.5: Insert a row.

```
# insertRow.py
# Purpose: Insert a new row without geometry.
# Usage: No script arguments needed.
import arcpy, traceback

# Create an insert cursor
fc = 'C:/gispy/data/ch17/firesCopy.shp'
fields = ['FireId','CalendarYe']

try:
    cursor = arcpy.da.InsertCursor(fc, fields)
    # Create a list with FireId=513180 & CalendarYr=2009
    newRecord =[513180, 2009]
    # Insert the row (otherwise no change would occur)
    cursor.insertRow(newRecord)
    print 'Record inserted.'
    del cursor
except:
    print 'An error occurred.'
    traceback.print_exc()
    del cursor
```

More than one row can be inserted at a time. Example 17.6 first uses a search cursor and the built-in `max` function to find the maximum fire number in the table. Then it uses an insert cursor to insert five additional fires for calendar year 2015 with successive fire numbers starting after the previous maximum.

Example 17.6: Find the maximum value of an attribute and insert multiple rows using this information.

```
# insertRows.py
# Purpose: Insert multiple rows without geometry.
# Usage: No script arguments needed.
import arcpy, traceback

fc = 'C:/gispy/data/ch17/firesCopy.shp'

# Find the current fire numbers.
try:
    fields = ['FireNumber']
    cursor = arcpy.da.SearchCursor(fc, fields)
    fireNumbers = [row[0] for row in cursor]
    print '{0} fire numbers found.'.format(len(fireNumbers))
    del cursor
except:
    print 'An error occurred in the search.'
    traceback.print_exc()
    del cursor

# Insert 5 new fires for year 2015.
try:
    fields.append('CalendarYe')
    cursor = arcpy.da.InsertCursor(fc, fields)
    # Find the max value in list and increment by 1
    fireNum = max(fireNumbers) + 1
    for i in range(5):
        # Create a row with unique fire number & year=2015
        newRow = [fireNum, 2015]
        fireNum = fireNum + 1
        # Insert the row.
        cursor.insertRow(newRow)
        print '''New row created with \
        fire {0} and year = {1}.'''.format(
        newRow[0], newRow[1])
    del cursor
except:
    print 'An error occurred in the insertion.'
    traceback.print_exc()
    del cursor
```

17.8.1 Inserting Geometric Objects

The rows added in Examples 17.6 and 17.7 appear in the tables, but no new points appear because we have not specified locations. To set the 'Shape' field, the script must create the appropriate `arcpy Geometry` objects. The `arcpy` package has functions named `Point`, `Multipoint`, `Polyline`, and `Polygon` for creating `Geometry` objects. The following code creates an `arcpy Point` object with `x = -70.1` and `y = 42.07` and stores it in the variable named `myPoint`:

```
>>> myPoint = arcpy.Point(-70.1, 42.07)
>>> myPoint
<Point (-70.1, 42.07, #, #)>
```

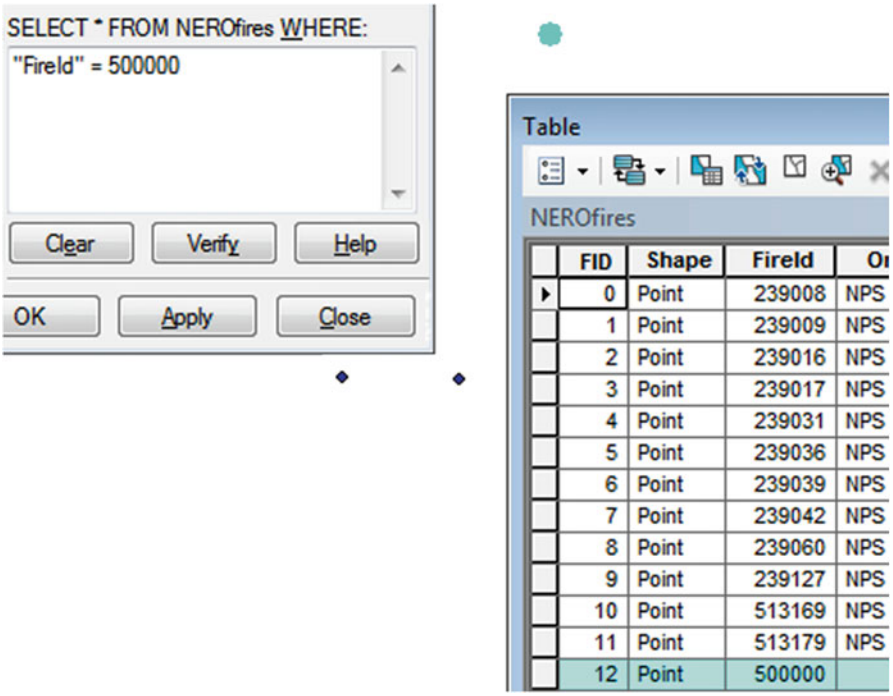
`Multipoint`, `Polyline`, and `Polygon` features are composed of multiple points. These more complex shape types require the user to create an `Array` object and then pass this to the `Geometry` object to create the feature. The following code creates a `Polyline` connecting points `a` and `b`:

```
# Create 2 polyline endpoints.
>>> x = 2134000
>>> y = 179643
>>> a = arcpy.Point(x,y)
>>> x = 2147000
>>> y = 163267
>>> b = arcpy.Point(x,y)
>>> a
<Point (2134000.0, 179643.0, #, #)>
>>> b
<Point (2147000.0, 163267.0, #, #)>

>>> # Create an array with a Python list of Point objects.
>>> myArray = arcpy.Array([a,b])

>>> # Create a line with an Array object that has points.
>>> line = arcpy.Polyline(myArray)
>>> line.length
20908.691398554813
```

Examples 17.7 and 17.8 use `Geometry` objects, along with insert cursors, to insert features. Example 17.7 inserts a point. It uses the `SHAPE@XY` token since it is only specifying a point to be added. The script creates a `Point` object and uses the point to define the new row. The new row is a Python list variable, as usual. The order of the values in the new row list corresponds to the order of the fields in the



"FireId" = 500000

	FID	Shape	FireId	Or
▶	0	Point	239008	NPS
	1	Point	239009	NPS
	2	Point	239016	NPS
	3	Point	239017	NPS
	4	Point	239031	NPS
	5	Point	239036	NPS
	6	Point	239039	NPS
	7	Point	239042	NPS
	8	Point	239060	NPS
	9	Point	239127	NPS
	10	Point	513169	NPS
	11	Point	513179	NPS
	12	Point	500000	

Figure 17.4 New point selected, highlighted in blue in the top right corner of this image.

field_names parameter list when the cursor was created. In other words, the FireId value, of 500000, is placed first and the Shape object is placed second. There is nothing special about this ordering except that these two lists must use the same ordering. Finally, the new row is added to the table. Prior to the call to the insertRow method, no changes were made to the attribute table. Once this call has succeeded, the new point is added. Figure 17.4 shows this point highlighted when the new row is selected in ArcMap.

Example 17.7: Insert a point.

```
# insertPoint.py
# Purpose: Insert a point with a Geometry object.
# Usage: No script arguments needed.

import arcpy, traceback

fc = 'C:/gispy/data/ch17/firesCopy.shp'
try:
    ic = arcpy.da.InsertCursor(fc, ['FireId', 'SHAPE@XY'])
```

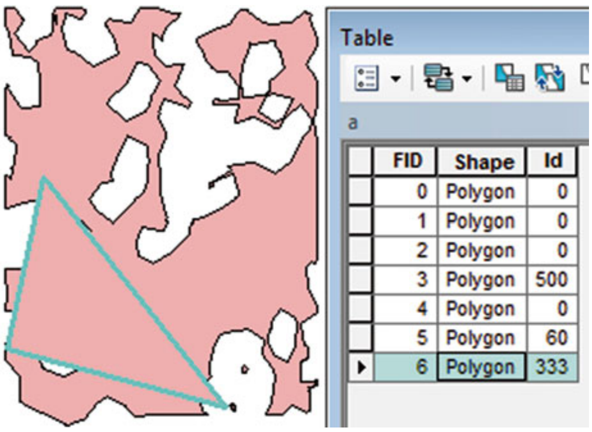
```
        # Create a point with x = -70.1 and y = 42.07
        #     to be used for the Shape field.
        myPoint = arcpy.Point(-70.1, 42.07)

        # Create a row list with FireId=500000 and the new point.
        newRow =[500000, myPoint]

        # Insert the new row.
        ic.insertRow(newRow)
        print 'New row inserted.'

        del ic
    except:
        print 'An error occurred.'
        traceback.print_exc()
        del cursor
```

Example 17.8 inserts a polygon. It uses the `SHAPE@` token to get the entire `Geometry` object, since it is inserting a polygon instead of a line. Inserting any feature more complex than a point requires this because `SHAPE@XY` is the shape's centroid. The script creates three points and then creates an `Array` object by passing it a list of the points (`myArray = arcpy.Array([a,b,c])`). Then the polygon is created from the `Array` object (the polygon is created with this line of code: `poly = arcpy.Polygon(myArray)`) and the Shape field is set using that polygon. In this example, the geometry token is given first in the `field_names` parameter list, so it is listed first in the new row. The new row is inserted, which adds the triangle highlighted in Figure 17.5.



	FID	Shape	Id
	0	Polygon	0
	1	Polygon	0
	2	Polygon	0
	3	Polygon	500
	4	Polygon	0
	5	Polygon	60
▶	6	Polygon	333

Figure 17.5 New triangular polygon created with the code in Example 17.8.

Example 17.8: Insert a polygon.

```
# insertPolygon.py
# Insert cursor polygon example.

import arcpy, traceback

# Create 3 point objects for the triangle
a = arcpy.Point(2134000, 179643)
b = arcpy.Point(2147000, 163267)
c = arcpy.Point(2131327, 167339)

# Create an array, needed for creating a polygon
myArray = arcpy.Array([a,b,c])

# Create a polygon.
poly = arcpy.Polygon(myArray)
fc = 'C:/gispy/data/ch17/aggCopy.shp'

try:
    # Create an insert cursor
    cursor = arcpy.da.InsertCursor(fc, ['SHAPE@', 'Id'])

    # Create row list
    newRow = [poly, 333]
    # Insert the new row.
    # It's automatically given an FID one greater
    # than the largest existing one.
    cursor.insertRow(newRow)
    print 'Polygon inserted.'
    del cursor
except:
    print 'An error occurred.'
    traceback.print_exc()
    del cursor
```

17.9 Selecting Rows with SQL

The `SearchCursor` and `UpdateCursor` functions have an optional `where_clause` parameter that can be used to refine a selection of features, such as selecting the features in 'fires.shp' which have a class size of `A`. This can be accomplished by looping with a nested condition, but the cursor is optimized to make this selection faster, which can be important for large files. Just like the `where_clause` parameters in ArcGIS tools that make selections, these expressions must be specified as SQL statements that use SQL comparison operators. The following expressions are examples of `where_clause` values that could be used to select rows in the 'fires' shapefile data:

```
>>> query1 = "SizeClass = 'A'" # Fires of size class A.
>>> query2 = "FireName <> 'MEADOW'" # Fires not named MEADOW.
>>> query3 = 'FID > 6' # Fires with ID greater than 6.
>>> query4 = "StartTime = date '2000-01-06'" # After Jan.6,2000
```

The same formatting described in Section 9.2, 'ArcGIS Tools that make selections', applies here. We can make a few observations from these examples. The first two expressions involve text fields so they use the single quotations around the field value. The SQL inequality operator (<>) is different from the Python one we use (!=). Only one pair of quotation marks is used for numeric fields. Fields other than text and numeric types, such as the date field used in `query4`, require specialized syntax. Read more in the ArcGIS Desktop help topic, 'SQL reference for query expressions'.

To select a subset of rows, create the cursor and use a `where_clause` query as the third parameter. Example 17.9 uses `query1` to print the years for the records when the fire size class is `'A'`:

Example 17.9: Use a `where_clause` with a cursor.

```
# sqlQueryCursor.py
# Purpose: Use a SQL query to select specific records.
# Usage: No script arguments needed.
import arcpy, traceback
fc = 'C:/gispy/data/ch17/fires.shp'

# Create the where_clause
query = "SizeClass = 'A'"
try:
    sc = arcpy.da.SearchCursor(fc, ['CalendarYe'], query)

    for row in sc:

        print row[0],
    del sc
except:
    print 'An error occurred.'
    traceback.print_exc()
    del cursor
```

The `SizeClass` field is used to specify a selection (those rated `'A'`) and the `CalendarYe` is printed for these seven records. The printed output looks like this:

```
>>> 1997.0 1997.0 1997.0 1999.0 1999.0 2004.0 2005.0
```

Queries can be formed without hard-coded field names, by using string formatting. The query used in Example 17.10 will have the same effect as `query3` if the user passes `FID` as the second script argument. This query would select the row with `FID` greater than 6. A list comprehension is used to create a list of the values of `fieldToPrint`. When this script is run with the example input, the results look like this:

```
[u'COCONUT', u'HIGHHEAD', u'HRCOVEDUNE', u'Beech Fost',
u'Pilg. Hgts']
```

Example 17.10: Use a cursor `where_clause` with a variable.

```
# whereClauseWithVar.py
# Purpose: Use a SQL query to select specific
#    records based on user arguments.
# Example: C:/gispy/data/ch17/fires.shp FID FireName
import arcpy, sys, traceback

fc = sys.argv[1]
numericField = sys.argv[2]
fieldToPrint = sys.argv[3]

query = '{0} > 6'.format(numericField) # string formatting

try:
    with arcpy.da.SearchCursor(fc, [fieldToPrint],
                               query) as cursor:
        recordList = [row[0] for row in cursor]
    del cursor
    print recordList
except:
    print 'An error occurred.'
    traceback.print_exc()
    del cursor
```

17.10 Key Terms

Cursor
Record
`arcpy DataAccess (da)` module search, update, and insert cursors
Search and update cursor `next` and `reset` methods
Cursor `fields` property
Update cursor `updateRow` and `deleteRow` methods
Insert cursor `insertRow` method
`arcpy` geometry tokens
`arcpy Geometry` objects
`arcpy Point`, `Array`, `Polyline`, and `Polygon` methods

17.11 Exercises

1. **updateUPPER.py** Use the Copy (Data Management) tool to make a copy of the input file in 'C:/gispy/scratch/' and then use an update cursor to modify the string field input by the user so that all characters in that field are uppercased. Use two required arguments, an input table and a field name. In the example below, the output 'COVER' field values become WOODS, ORCH, and OTHER.

   Example input:
   ```
   C:/gispy/data/ch17/park.shp COVER
   ```
   Example output:
   ```
   Copied C:/gispy/data/ch17/park.shp to C:/gispy/scratch/park.shp
   Modified C:/gispy/scratch/park.shp
   ```

2. **cursorBasics.py** Practice using search and update cursors. Write a script that copies 'C:/gispy/data/ch17/park.shp' to 'C:/gispy/scratch/' and then does each of the tasks in a-c for the file in 'scratch'. Since the table and FID values are hard-coded, this script takes no arguments. Find the specified rows using the `where_clause` parameter.

   (a) Using a search cursor, find the row with an `FID` value of 45 and print the `COVER` field value in that row.

   Printed output:
   ```
   Row with FID = 45 has cover orch
   ```

   (b) Using an update cursor, find the row with an `FID` value of 120 and change the cover name to ‘park’.

   Printed output:
   ```
   Row with FID = 120 has been updated to use cover: park
   ```

   (c) Using an update cursor, find the row with an `FID` value of 22 and delete it. Note that when you delete a row, it updates the FIDs so that there will be a new entry with the FID of 22 even after successful deletion. To confirm that the row was deleted, call the Get Count (Data Management) tool before and after deleting this row and print the before and after counts.

   Printed output:
   ```
   Count before deletion: 426
   Count after deletion: 425
   ```

3. **buggyCursor.py:** Sample script ‘buggyCursor.py’ is supposed to count the number of records in ‘parkCopy3.shp’ with a ‘COVER’ field value that is neither ‘woods’ and nor ‘orch’. Correct the six errors in the script. Insert comments describing each error. The correctly working script should print these results:

   ```
   Number of records with other cover types: 62
   ```

4. **printStringFields.py** Modify sample script 'printTableExclude.py' so that it prints the field names and row tuples of only the string type fields. Rename the `excludeFields` function as `includeFields` and modify it to *include* (instead of exclude) the field types being passed in. The printed output should look like this:

```
(u'FireName', u'SizeClass', u'Authoriz_1')
((u'MEADOW', u'A', u'Fire Technician')
(u'LITTLE CRK', u'A', u'ND Fire Techn')
(u'T.Calvin', u'B', u'Fire Technician')
```
   . . . and so forth.

5. **deleteHigh.py** Write a script that copies the input data to 'C:/gispy/scratch/' and then uses cursor functionality to remove every record of a table where a given field is higher than a given value. Use three required arguments, the full path file name of an input file, the name of a numeric field in that file, and a search value.

   Example input: C:/gispy/data/ch17/firesCopy2.shp FID 8

   Example output:
```
Remove row: [9]
Remove row: [10]
Remove row: [11]
```

6. **deleteSpaceRows.py** Write a script that copies the input data to 'C:/gispy/scratch/' and then uses cursor functionality to remove every row of a table in which the value of a given field contains a space. Use two required arguments, the full path file name of an input file and the name of a text field in that file.

   Example: C:/gispy/data/ch17/firesCopy3.shp FireName

   Example output:
```
Removed row containing: [u'LITTLE CRK']
Removed row containing: [u'PILGRIM HT']
Removed row containing: [u'Beech Fost']
Removed row containing: [u'Pilg. Hgts']
```

7. **insertLine.py** Sample script 'insertLine.py' finds the centroids (`point1` and `point2`) of the first two records in 'C:/gispy/data/ch17/park.shp'. Add code to this script to insert a line from `point1` to `point2` in the polyline shapefile 'C:/gispy/scratch/parkLines.shp' and set the value of 'LEFT_FID' to 50 for this record. The new line should look like the highlighted line segment in the image below: