

A Python Primer for ArcGIS®

Workbook II

Nathan Jennings

Copyright © 2015 Nathan Jennings
All rights reserved.
ISBN: 1505893445
ISBN-13: 978-1505893441

ACKNOWLEDGEMENTS	7
-------------------------------	----------

INTRODUCTION	9
---------------------------	----------

<i>Objectives and Goals</i>	<i>9</i>
<i>Structure of the Workbooks</i>	<i>11</i>
<i>Data and Demos</i>	<i>13</i>
<i>Accessing the Data, Demos, and Code</i>	<i>14</i>
<i>Required Software</i>	<i>14</i>
<i>Older Versions of ArcGIS and Python</i>	<i>15</i>
<i>Reporting Errata</i>	<i>15</i>
<i>Prerequisite Knowledge and Skill</i>	<i>16</i>
<i>Problem Solving</i>	<i>16</i>
<i>Developing Geoprocessing Workflows</i>	<i>18</i>

WORKBOOK II: WRITING PYTHON SCRIPTS FOR COMMON GEOPROCESSING TASKS	19
---	-----------

Chapter 5 Querying and Selecting Data	21
--	-----------

<i>Overview</i>	<i>21</i>
<i>Feature Layers and Table Views in ArcGIS</i>	<i>23</i>
<i>Feature Layers and Table Views in Python</i>	<i>25</i>
<i>Query Syntax</i>	<i>26</i>
<i>Building the Query Syntax</i>	<i>26</i>
Field Name Syntax	27
Developing and Processing Strings in Query Expressions	27
Common Operators in Queries	29
Wildcard Characters	30
NULL Values	30
Numerical Expressions in Queries	31
Using Calculations in Queries	31
Combining Expressions	32
<i>Other Query Syntax</i>	<i>32</i>
<i>Make Feature Layer, Make Table View, and Selecting Data</i>	<i>33</i>
<i>Selecting Data</i>	<i>33</i>
Programming the SelectLayerByAttribute Routine	34
Select Data by Location	34
Programming the SelectLayerByLocation Routine	37
<i>Counting the Number of Records</i>	<i>38</i>
<i>Creating a New Dataset</i>	<i>38</i>
<i>Data Locks</i>	<i>39</i>
<i>Summary</i>	<i>41</i>

Chapter 5 Demos	43
------------------------------	-----------

Demo 5a: Create a Feature Layer with and without a Query	45
---	-----------

Demo 5b: Select Features by Attribute	51
Exercise 5 - Select Features by Attribute/Location and Write Data to a New Feature Class	61
Chapter 5 Questions.....	63
Chapter 6 Creating and Using Cursors and Table Joins.....	65
<i>Overview.....</i>	65
<i>Data Access Module</i>	65
<i>Cursors.....</i>	66
Illustrating Cursor Processing using the Search Cursor	67
Insert Cursor	71
Schema Locks on Data	73
Creating and Using the Insert Cursor	75
Update Cursor	77
Creating and Using the Update Cursor	81
<i>Table Joins</i>	85
<i>Attribute Indexes</i>	90
<i>Programming and Using Table Joins</i>	93
Creating an Attribute Index	93
Creating Feature Layers or Table Views	94
Creating the Table Join	95
Using and Accessing Information in Joined Data.....	95
<i>Summary</i>	98
Chapter 6 Demos.....	99
Demo 6a: Search Cursor	101
Demo 6b: Insert Cursor.....	105
Demo 6c: Search and Update Cursor	113
Demo 6d: Joining Tables.....	121
Exercise 6 - Using Cursors and Table Joins	129
Chapter 6 Questions.....	137
Chapter 7 Describing Data, Lists, and Raster Processing Basics.....	139
<i>Overview.....</i>	139
<i>Describing Data</i>	139
<i>Listing Data</i>	144
<i>Raster Processing Basics using the Spatial Analyst Module</i>	146
Raster Data Organization.....	147
Accessing and Using Image Bands.....	148
<i>Summary</i>	149

Chapter 7 Demos	151
Demo 7a: Describe Image Properties and Perform Image Processing	153
Demo 7b: List and Use Feature Classes and Images.....	159
Exercise 7 - Batch Clip Images Using a Feature Class	165
Chapter 7 Questions.....	171
Chapter 8 Custom Error Handling and Creating Log Files.....	173
<i>Overview</i>	<i>173</i>
<i>Custom Error Handlers</i>	<i>173</i>
<i>Creating and Using an Error Handling Class</i>	<i>174</i>
<i>Using Log Files to Collect Messages.....</i>	<i>176</i>
Creating and Using a Log File	177
Adding the Date or Time to a File Name or Message	179
<i>Summary.....</i>	<i>179</i>
Chapter 8 Demo Create Custom Error Messages	181
Exercise 8 - Create Custom Error Messages and Log Files for a Script	187
Chapter 8 Questions.....	189
Chapter 9 Mapping Module	191
<i>Overview</i>	<i>192</i>
<i>Map Elements</i>	<i>193</i>
<i>Map Element Relationships</i>	<i>196</i>
<i>Prerequisites</i>	<i>197</i>
Create a Map Template	197
Name Map Layout Elements	198
<i>Mapping Module Class Properties and Methods</i>	<i>200</i>
<i>Mapping Module Functions</i>	<i>201</i>
<i>Implementing Map Documents, Data Frames, Layers, and Layout Elements</i>	<i>201</i>
Import the mapping Module	202
Accessing an ArcMap Document	203
Accessing a Data Frame	204
Accessing Layers and Layer Properties	205
Using Layer Properties in the Script.....	207
Accessing and Changing Layout Elements	208
<i>Export Map to PDF or Print Map to Printer</i>	<i>209</i>
<i>Saving Map Documents</i>	<i>211</i>
<i>Working with Data Frame, Layer, and Layout Element Methods</i>	<i>212</i>
Changing the Map Extent using a Definition Query (or not)	212
Changing the Map Extent using Selected Features.....	214
Adding and Saving Layer Files	218
<i>Creating a Map Book Programmatically using the mapping Module</i>	<i>220</i>
<i>Summary.....</i>	<i>221</i>

Chapter 9 Demos..... 223

Demo 9a: Mapping Module Overview and Properties..... 225

Demo 9b: Implementing Mapping Module Methods..... 231

Exercise 9 - Create a Simple Neighborhood Map Set..... 245

Chapter 9 Questions 247

ACCESSING DATA, DEMOS, AND CODE249

REFERENCES250

INDEX.....251

Acknowledgements

A Python Primer for ArcGIS® Workbooks are a culmination of the author's experiences and relationships with a number of people and organizations and could not have been written without them. The author would like to acknowledge the Environmental Systems Research Institute (Esri®), the company that provides geographic information systems (GIS) software to most of the world's GIS users. This organization and software has made it possible for many people and organizations to explore, analyze, and depict their world using geographic information. Specific to this book, Esri has developed modules and objects that can be used with the open source Python programming language. Doing so has allowed their software to become more customized and expanded for specific geoprocessing tasks.

The author would also like to acknowledge the City of Sacramento and ICF International (formerly, Jones and Stokes). These organizations provided the impetus for the author to develop his own Python programming skills and knowledge and are sources for some of the demonstrations and exercises in this book. In addition, the author would like to acknowledge American River College in Sacramento, CA, the Geography and Science department, and especially the students in the GIS Program. The author developed and teaches the on-line GIS Programming course at American River College and the students have served as the "testers" of the material in this book. Their feedback has been valuable for many of the edits that went into this book.

The author acknowledges the full GIS staff at the City of Sacramento. These colleagues have been some of the best to work with over the author's career and represent some of the finest GIS professionals in the community. Specifically, the author would like to mention Dan McCoy. Dan has been a valuable resource to bounce ideas off of and to help clarify some of the coding logic and geoprocesses that found its way into the text. In addition, the author would like to thank the Central GIS team that the author works with. In addition to Dan, the team includes Maria MacGunigal, David Wilcox, Rong Liu, and Carlos Porras. The author would like to especially thank Dr. Este Geraghty who took her time as a student in the author's class and with her very busy schedule to review, comment, and make suggestions for *A Python Primer for ArcGIS*. Her feedback is sincerely appreciated. The author sincerely appreciates the time and efforts Ben Logan, Virginia Tech State University in Blacksburg, VA, spent providing significant editorial feedback, comments, and suggestions for this edition. His input has made the book better.

Cover design by Zach Jennings; digital media support by Josh Jennings, Urbandale Spatial.

Esri® ArcGIS® software graphical user interfaces, icons/buttons, splash screens, dialog boxes, artwork, emblems, and associated materials are the intellectual property of Esri and are reproduced herein by permission. Copyright © 2011 Esri. All rights reserved. Esri, ArcGIS, ArcInfo, ArcEditor, ArcMap, ArcCatalog, ArcView, ArcSDE, ArcToolbox, 3D Analyst, ModelBuilder, ArcPy, ArcGlobe, ArcScene, *ArcUser*, and **www.esri.com** are trademarks or registered trademarks or service marks of Esri and are used herein by permission.

Introduction

A Python Primer for ArcGIS Workbook II builds upon the fundamentals found in *Workbook I* and contains the “bulk” of the common geoprocessing tasks that many GIS professionals encounter when doing GIS. If the reader is new to GIS and to Python, it is recommended to check out *Workbook I*. For those who have some GIS background and programming experience *Workbook II* can be the launch point to dive into to some of the core concepts of ArcGIS Python programming. *Workbook III* serves as a volume focused on advanced concepts such as Python functions, custom script tools, Python Add-ins, and fully automating geoprocessing scripts. The objectives and goals as well as the sections on problem solving and developing geoprocessing workflows are repeated here for those skipping *Workbook I* since these concepts are at the heart of the *Workbook* series.

Objectives and Goals

A Python Primer for ArcGIS Workbook series is written for those who want an introduction to using Python in the context of ArcGIS. *A Python Primer for ArcGIS* is not a detailed text on Python. Others have already accomplished this task. References can be found throughout the book. *A Python Primer for ArcGIS* will help newcomers to GIS and programming. It will also help strong ArcGIS users who do not yet have a solid knowledge, or expertise, in writing scripts. For those who have some background in programming, many of the concepts—such as variables, loops, conditional statements, etc.--will be familiar and helpful in developing Python code. For those who do not, *Workbook I* will serve as a starting point to develop code using some of the basic programming structures commonly used in many of the ArcGIS geoprocessing tasks. *A Python Primer for ArcGIS Workbook* series focuses on developing geoprocesses and Python code toward the goal of standalone scripts that can be implemented both inside and outside of ArcGIS.

The workbooks accomplish the following objectives:

1. Provides a framework for code developers of different skill sets, to design logical geoprocesses.
2. Teaches how to design logical coding structures that include proper constructs for
 - error handling,
 - troubleshooting processes,
 - logic, and
 - scripting problems.
3. Introduces common Python constructs, illustrating how they are implemented with ArcGIS geoprocessing tools.
4. Teaches code developers how to obtain help with Python and ArcGIS geoprocessing functions, in the process of building their own code writing skill.
5. Introduces some of the new functionality of Python and ArcGIS, such as the mapping and data access modules.
6. Shows how to integrate custom-built scripts with the ArcToolbox™
7. Shows how to make and auto-run custom scripts.

The common Python elements used in ArcGIS and a few of the most widely used geoprocessing tasks will make up the majority of the book's content, and will serve the primary reason why the author focuses on developing standalone scripts.

With a grounding in Python structure and syntax and common ArcGIS functions, readers will be able to apply this new facility to more complex scripting and geoprocessing tasks (e.g. Python dictionaries, arrays, functions or ArcGIS extensions, ArcSDE®, and specialized geoprocessing methods). Readers should study the concepts in *A Python Primer for ArcGIS* workbooks, perform the demonstrations and exercises, and answer the chapter questions. Upon completion, readers should be able to design, develop, create, troubleshoot and successfully run Python scripts with multiple steps and multiple ArcGIS geoprocessing functions and methods.

Make sure to see the *Accessing the Data, Demos, and Code* section at the end of the book to obtain the data and scripts that accompany the workbooks.

Structure of the Workbooks

A Python Primer for ArcGIS is divided into three separate workbooks so the newcomer to Python and ArcGIS can begin with *Workbook I* and work through all of the material and obtain a firm grounding in Python programming as well as become more familiar with the ArcGIS geoprocessing structure. Those that already have a fundamental understanding of Python and ArcGIS can begin with *Workbook II* to gain more insight into common geoprocessing tasks that many GIS professionals encounter. *Workbook III* takes the fundamentals and the common geoprocessing tasks a step further and provides some guidance to relate Python scripts to custom ArcTools and to learn how to “auto run” a functional Python script. The author hopes that providing the material in several workbooks allows an economical and useful way for the reader to learn and gain valuable experience in developing geoprocessing scripts using Python and ArcGIS.

Workbook I introduces Python and augments the user's experience in ArcGIS toward writing some simple geoprocessing scripts.

Chapter 1 introduces Python and briefly discusses its history, the relation of Python to past and present versions of ArcGIS, the use of IDLE, and the all-important subject of “How to Get Help.” The chapter introduces a very useful prototype for writing code that collects errors for the user to examine in the de-bugging process.

Chapter 2 introduces ModelBuilder™. ModelBuilder is extremely useful for the beginning Python/ArcGIS user. Initially, the user builds a straightforward, simple geoprocessing model—a runnable diagram—simply by dragging and dropping elements and connecting them with arrows in a logical manner. After a successful model is run, the user can then export the model as a Python script. The budding programmer can then study the basic elements and flow of this script and how the method parameters are filled in—a particular problem for newbie and experienced scripters alike.

Chapter 3 introduces some essential Python constructs and stresses strict adherence to their syntax. Handled here are variables, lists, conditional statements and loops, modules, and `try:` and `except:` blocks. These are some of the workhorses of Python scripting, without which many processes would require hours of manual mouse-and-keyboard labor. Some bugaboos discussed are strings with forward and backward slashes, and the mixing of single and double quotes and triple double quotes.

Chapter 4 brings the user to the workstation to write the first basic geoprocessing Python scripts from scratch. This chapter introduces such good user habits as writing pseudo-code as comments within the draft of a Python script. The demo and exercises bring together the concepts, best practices, and elements introduced in the first three chapters.

Workbook II focuses on how to develop Python code for many of the commonly used GIS tasks. These include developing queries, selecting and using data, reading and writing new data to records, working with raw image data, and creating automated map production routines.

Chapter 5 introduces the topics of building and using queries as well as Feature Layers and Table Views. These concepts are key elements to the Select Layer By Attribute and Select Layer By Location geoprocessing routines. This chapter also includes a brief discussion on creating a new data set and issues with data locks.

Chapter 6 focuses on cursors. Cursors are common database structures that allow the user to uniquely interact with specific records or collections of records of data sets. This chapter also discusses the implementation of the `for` loop to iterate through the records. The chapter ends with an example of creating and using table joins with cursors.

Chapter 7 reviews the Describe routine to obtain useful information about data sets. In addition, ArcGIS lists and raster data are discussed. The raster portion of the chapter shows how individual bands of data from a multi-spectral data set can be accessed and a custom-built algorithm implemented using Python syntax as well as the Spatial Analyst extension and `sa` module.

Chapter 8 provides a brief discussion on handling errors and creating custom error handling routines that can be useful for some scripting projects.

Chapter 9 introduces and provides an overview of the ArcGIS mapping module. The reader will discover the different components of an ArcMap™ document that can be manipulated when creating automated mapping routines (such as creating a map book or map atlas).

Workbook III covers some “next steps” a GIS coder can develop to enhance geoprocessing Python scripts.

Chapter 10 shows how the code developer can tie a graphical user interface (GUI) to an ArcGIS Python standalone script using a custom ArcTool or Python script tool. Python functions are introduced.

Chapter 11 reviews the Python Add-in and show how code developers can create and add functionality to some simple GUIs on a custom toolbar.

A Python Primer for ArcGIS Workbook III concludes with Chapter 12 briefly discussing how to set up automation processes through Windows Scheduled Tasks so that Python scripts can run in a completely automated and scheduled fashion.

Most chapters will have a demonstration program that the reader can work on and develop using step by step examples. In addition, the author recommends the reader can work on the chapter exercises to obtain more experience. Most chapters have questions that reinforce the important concepts.

The author uses the following typeface conventions throughout the book:

Street_CL – bold type typically indicates a feature class or table explicitly used in the text, demo, or exercise as well as references to data, files, and scripts provided with the book. Bold is also used to highlight ArcGIS Help documentation topics so the reader can easily find additional information provided by Esri.

StreetName – italics type typically indicates an attribute field. It will also be used to indicate a published work.

`arcpy.da.SearchCursor()` – courier type indicates example Python syntax within the text, demos, and exercises.

<required_parameter> - indicates a required parameter for an ArcGIS tool or routine
{optional_parameter} – indicates an optional parameter for an ArcGIS tool or routine

Data and Demos

All of the data and demo scripts can be found at the author's website at the end of the book. The supplemental material is organized as follows: **\PythonPrimer\ChapterXX**. Within each chapter the **Data** folder contains the data files required for the demo and/or exercise. Data files can be shapefiles, file geodatabase feature classes or tables, or standalone tables (e.g. dBase format), or TIF or ERDAS (.img) images. ArcMap documents (.MXD) can be used as referenced or renamed for readers to modify and save their own work. A **MyData** folder is also provided so that readers can save their own work for demos and exercises. All of the data and ArcMap documents will be in ArcGIS 10 or later format. NOTE: The ArcMap documents reference the **\PythonPrimer\ChapterXX** structure above. If the reader changes this folder structure, the ArcMap documents provided by the author may need to have the source files in the Table of Contents revised to the new location. The scripts have been tested on Windows 7 32-bit and 64-bit operating systems. The reader may need to make some additional adjustments to data paths on 64-bit Windows systems.

The data sources exist on the one of the following web sites or organizations:

City of Sacramento – city related vector data and historical 1991 aerial photos

County of Sacramento – parcel and street subsets

CalAtlas – Landsat Thematic Mapper (TM) satellite imagery subset

Refer to the text file associated with the supplemental data as well as the websites in the References at the end of the book for more information.

Accessing the Data, Demos, and Code

See the **Accessing the Data, Demos, and Code** section at the end of the Book.

Required Software

The user must have access to ArcGIS Basic, ArcGIS Standard, and ArcGIS Advanced, (aka ArcGIS ArcView®, ArcEditor™, or ArcInfo®, respectively) version 10.0 or later and install the Python version that comes with the ArcGIS media and not any other version. **Exceptions:** Chapter 6 and Chapter 9 use cursor syntax that supports ArcGIS 10.1 or later. Readers that only have access to ArcGIS 10.0 can refer to the “legacy” syntax and material in the **Chapter06\legacy** and **Chapter09\legacy** folders, respectively.

Students enrolled in the online Introduction to GIS Programming course (Geog 375) at American River College (<http://wserver.arc.losrios.edu/~earthscience/>) can obtain a one-year student license of ArcGIS. Contact the author to check enrollment and validate academic status.

Alternatively, the reader can obtain a copy of ArcGIS for Home Use at <http://www.Esri.com/arcgis-for-home/index.html> or from one of the ArcGIS books from Esri that comes with a CD and DVD. The CD contains the data, demos, exercises, and solutions; the DVD contains a 180 day fully functional copy of ArcView. Esri can be contacted to receive an evaluation copy of ArcGIS that can be used with this book. Readers with access to ArcGIS only need to copy the data referenced in the book to get started with *A Python Primer for ArcGIS*. Readers are encouraged to review their own data or a company’s data collection and practice writing additional scripts beyond the exercises and demonstrations provided in this text.

Older Versions of ArcGIS and Python

As of the writing of this edition, ArcGIS 9.3 is officially in retired status; ArcGIS 10 is in mature status (see <http://support.esri.com/en/content/productlifecycles> for more information). The scripts and content in this edition work with ArcGIS 10.0 through the present version of ArcGIS. The two exceptions are Chapter 6 which discusses cursors and the Chapter 9 exercise that uses a search cursor. Chapter 6 and Chapter 9 use the 10.1 version of cursors and reference the data access module which was introduced with ArcGIS 10.1. The older legacy cursor format is provided in the **Chapter06\legacy** and **Chapter09\legacy** folder, however, the content of Chapter 6 references the *arcpy*TM Data Access format for cursors. It is recommended that the latest version of the software be installed to use the materials for *A Python Primer for ArcGIS*.

Reporting Errata

The author encourages readers to provide feedback on the text, demo scripts, examples, and exercises so these improvements can be added to future editions. Feel free to email the author at: nate.jennings@urbandalespatial.com.

Prerequisite Knowledge and Skill

The reader diving into *A Python Primer for ArcGIS* should have a fundamental understanding of GIS concepts such as geographic features (points, lines, and polygons), feature classes, GIS geospatial data formats, data and attribute tables, relational databases, records, rows, fields, columns, etc. As well, the reader should have a fundamental understanding of ArcGIS, how it is structured, and how to use ArcMap™, ArcCatalog™, and ArcToolbox™. She or he should also know how to use some of the geoprocessing tools (e.g. Clip, Buffer, Select Layer by Attribute, Select Layer by Location, etc.) within ArcToolbox. Familiarity with ModelBuilder™ is recommended, but not required to use this book. One may also find requisite knowledge to get started with *A Python Primer for ArcGIS* in some of the Esri courses or similar introductory college GIS courses that use ArcGIS.

The reader does not need to know how to program or know Python or any other programming language. This text will provide an introduction to Python and general Python programming constructs that can be used with ArcGIS. For those who do have some Python and *arcpy* experience, the reader can skip to *Workbook I*, Chapter 4 and can refer to Chapters 1-3 for basic review. Make sure to look at the *Accessing the Data, Demos, and Code* section at the end of the book to obtain the data, demo scripts, and supplemental scripts that are used and referenced in the book.

Problem Solving

Problem solving is an important skill to develop in an analytical field such as GIS. As a GIS professional and college instructor, the author has developed a variety of problem solving skills that he uses every day in his work. The author uses and communicates these with colleagues and clients. He teaches these to students in the classroom. In the author's experience, the workflow of these skills is roughly as follows:

1. Spend considerable time reading and studying documentation
2. Try out specific geoprocessing functions
3. Analyze data
4. Review and interpret intermediate and final results
5. Develop and test specific workflows, and
6. Build simple to complex geoprocesses.

Developing problem solving skill is not easy. It takes time and practice and hours of research to create solutions to GIS problems and scripts. One can think of this casually as a “heuristic” or modified “Scientific Method.” Readers are encouraged to

- consult ArcGIS help, on-line forums,
- study other developers’ code, and
- build a repository of scripts and samples for future reference.

Any or all of the above steps are used in code development. The proper result cannot be achieved without writing the proper code (instructions) for the “computer” to implement the script.

In addition, the author often creates written documentation (outside of in-line code documentation). This additional documentation explains

- processes,
- methods,
- data input/output, and
- solutions to intermediate problems

in “plain English.” These descriptions are later referenced for developing more comprehensive and formal documentation. The author encourages the reader to do the same. For those who enroll in the author’s classes or training, the author provides the opportunity to learn and develop problem solving skills. For those who refer to this book, consult the sources in the chapters of this book or contact the author for more information.

Developing Geoprocessing Workflows

Before a GIS person (or team) undertakes a geoprocessing problem, often a result, goal, product, or service is needed, desired, required, etc. These can take the form of creating a new data set, summarizing data to help make a decision, generating a set of maps to show results of geospatial analyses, providing a web service, or developing a process to manage and update data for a specific purpose. All of these tasks require some set of steps to generate the result and often require some kind of interpretation, analysis, and evaluation of data, and intermediate and final results.

It is beneficial to develop a geoprocessing workflow (e.g. a diagram) before a project starts. This will outline or map out the data requirements, processing steps, intermediate results, and final results. *(Oftentimes, in practice, during the hard work of strategizing and coding, the workflow is never developed or only developed afterward)*. If GIS analysts can develop an outline or diagram a workflow before a project commences, they can operate within a structured framework (i.e., the overall objectives and goals of the project). Having this larger perspective on a project or task, teams can develop solid solutions, geoprocessing tasks, products, and services. In addition, a team member can refer to an outline or workflow diagram providing documentation to the process, because many projects can take a number of weeks or months. A single person will likely not remember all of the specific tasks, data, and products. The GIS coder will likely be working on multiple projects at any given time. These outlines and workflows are also useful for internal documentation or in documents provided to other staff members or clients.

The workflow can take many forms, such as an outline of steps or a workflow diagram indicating the relationships between one step and another or how one step may be related to many steps. For example, one source dataset may be used in multiple geoprocesses. This kind of workflow is often seen when designing a geoprocessing model in ModelBuilder. The workflow can be fairly simple, involving a small number of geoprocessing tasks. Conversely, it can be complex, involving many data sources, processing steps, feedback (looping) mechanisms, and many outputs (geospatial data, tables, maps, web services, etc).

Workbook II: Writing Python Scripts for Common Geoprocessing Tasks

Workbook I described the Python fundamentals and syntax as well as the general layout and organization of Python scripts. In addition, *Workbook I* demonstrated the construction of simple Python scripts from some familiar ArcGIS geoprocessing tools inserting user defined variables that were then required as the geoprocessing tool parameters.

Workbook II focuses on the most commonly used geoprocessing tasks implemented within ArcGIS for different kinds of data and geoprocessing analyses. *Workbook II* comprises the bulk of *A Python Primer for ArcGIS* and will discuss the general Python and ArcGIS methods, syntax, and practical workflows for implementing these common tasks.

The major geoprocesses covered in *Workbook II* are:

Chapter 5 – Developing query syntax and selecting data using the *MakeFeature*, *TableView*, *SelectLayerByAttribute*, and *SelectLayerByLocation* routines.

Chapter 6 – Creating and using cursors, table joins, and the `for` and `while` loops.

Chapter 7 – Describing data, using and implementing lists, and basic raster processing using Spatial Analyst and the `sa` module.

Chapter 8 – Creating custom error handling statements, writing messages to log files, and being able to send messages to the ArcToolbox message dialog box.

Chapter 9 – Introducing and implementing the `mapping` module to create maps and automate map production processes.

At the end of *Workbook II* the reader should be well versed in coding many of the common geoprocessing tasks and have a familiarity with the `mapping` module to generate automated map production tasks.

Chapter 5 Querying and Selecting Data

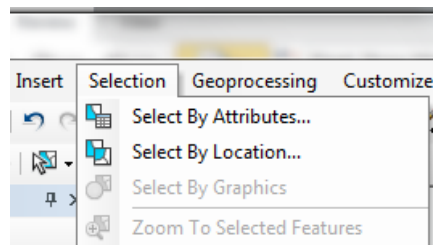
Overview

A common operation in GIS is to query data by selecting specified geographic features or rows in a table and then do other geoprocesses with the selected data. Querying and selecting data can be performed on one or more attributes (“attribute query”) or through spatial overlay processes (“spatial query”). Both kinds of queries can be performed using ArcGIS tools and Python. A number of spatial overlay processes exist in ArcGIS, such as Clip, Buffer, Intersect, and Union, however, this section covers two querying processes:

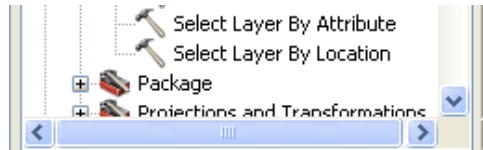
1. *Select Layer by Attribute*
2. *Select Layer by Location*

Using both kinds of queries in GIS allow an analyst to ask questions of the data both in a tabular fashion as well as based on spatial coincidence. An example of an attribute query might be “Select all of the cities in a cities point file that have a population greater than 100,000.” An example of a spatial query might be, “Select all of the parcels that fall inside of a flood zone.” In both cases, the analyst is making an inquiry of the data that meet one or more conditions. The examples shown above represent simple queries; however, in practice, queries can be complex with multiple conditions.

Often, when an ArcGIS user needs to perform an attribute or spatial query, one of the following Selection menu options are used.



For geoprocessing and model building the actual ArcGIS tools exist in the **Data Management Toolbox** within the **Layers and Table Views Toolset**. These are the tools that will be used in Python scripts.



The programming functionality of each tool is shown below.

The `SelectLayerByAttribute` contains the following parameters.

```
arcpy.SelectLayerByAttribute_management (<feature layer_or_table
view>, {selection_type}, {where_clause})
```

The `SelectLayerByAttribute` routine provides the ability to create an attribute query on either a geographic feature set or table. The selection can be one of the following selection types:

1. NEW_SELECTION
2. ADD_TO_SELECTION
3. REMOVE_FROM_SELECTION
4. SWITCH_SELECTION
5. SUBSET_SELECTION
6. CLEAR_SELECTION

The “where clause” (attribute query) parameter can be used to limit the selection based on one or more conditions.

The `SelectLayerByLocation` provides the ability to perform a spatial selection between two feature sets and contains the following parameters.

```
arcpy.SelectLayerByLocation_management (<feature layer>,
{overlay_type}, {select_features}, {search_distance},
{selection_type})
```

The `SelectLayerByLocation` contains the same “selection types” as the `SelectLayerByAttribute` except `CLEAR_SELECTION`, but performs one of numerous “spatial overlay” operations such as a geometric intersection (e.g. `INTERSECT`, the default overlay option). The `SelectLayerByLocation` routine can also operate on previously selected features (such as from an attribute or other spatial selection routine) and can operate on “selected features” from another feature set to further refine the “spatial selection.” In addition `SelectLayerByLocation` has a search distance parameter that can also be used to limit or refine the spatial selection.

These two selection operations are often at the heart of performing spatial analysis since GIS analysts typically need to perform analysis on a subset of information based on attribute or spatial queries or both. Before continuing with implementing the selection tools two key selection parameters need additional explanation:

1. *Creating feature layers and table views*
2. *Constructing appropriate query syntax*

Without understanding how either of these parameters are created and used in the selection routines programmers will struggle to develop the appropriate syntax and utilize the selection results in subsequent geoprocesses.

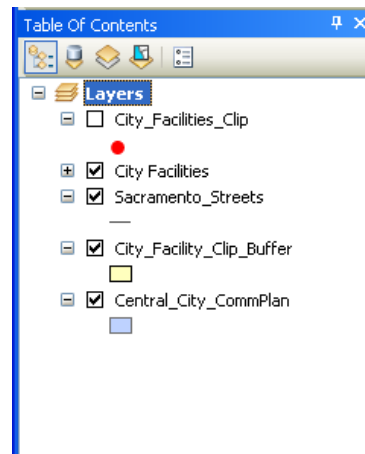
Feature Layers and Table Views in ArcGIS

The ArcGIS community often uses different terms to refer to the same topic or concept. Typically, feature classes, feature layers (or layers), tables, and table views are often used interchangeably to refer to feature classes or tables. From a code development point of view clearly understanding the distinction between these terms is important.

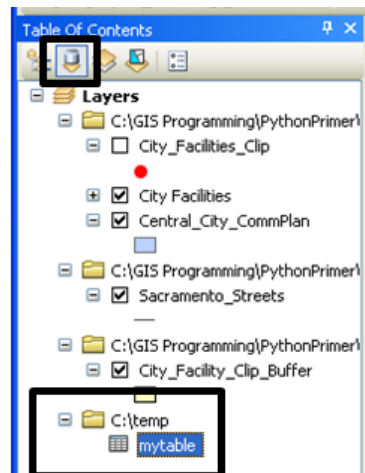
Feature classes and *tables* (objects that are often added to an ArcMap session or are created or viewed in ArcCatalog) represent actual files that can be found on disk or within a geodatabase. Feature classes can be in the form of shapefiles in a folder structure or special tables in geodatabases (file or ArcSDE) that contain the “SHAPE” field. Tables can be in the form of a standalone table (such as a dBase file) or a table within a geodatabase that does NOT contain the “SHAPE” field. Alternatively, *feature layers* and *table views* are “in memory” occurrences of feature classes and tables. No “new” data is created with feature layers or table views, however, subsequent geoprocessing tools can be used to create new feature classes (**Data Management—Features—Copy Features**) and tables (**Data Management—Tables—Copy Rows**) from them and save on disk. These will be discussed later.

Users of ArcGIS often work with feature layers and table views after adding them to an ArcMap session and working with geographic features or with geoprocessing tools. Feature layers and table views are actually shown in the Table of Contents of ArcMap. Essentially, the feature class or table becomes a feature layer or table view when either one is added to the table of contents. The user may not be aware of this, since the process is automatic and transparent. Feature layers and table views are only present during an active ArcMap or ArcCatalog session or when a model is processed. Once ArcMap or ArcCatalog is closed or a model completes processing the feature layer or table view is removed from memory.

The following illustration shows the Table of Contents of ArcMap showing “feature layers” in the table of contents.



Likewise, table views can be viewed by choosing the “List by Source” button from the Table of Contents.



When interacting directly with ArcGIS a user simply adds a feature class or table to the table of contents. Once either kind of data is added, the feature layer or table view is automatically created and the respective feature layers and table views can be used in ArcToolbox tools and ModelBuilder.

Feature Layers and Table Views in Python

The distinction between feature classes and tables and feature layers and table views becomes acutely important when coding the selection methods noted above. The reader should note that the input data type for either selection method is a *feature layer* or *table view* (not a feature class or table). Unlike using the selection tools within ArcMap where the feature layer or table view are “automatically created” and available to use in geoprocesses, the code development for standalone scripts requires a feature layer or table view to be “created” before either one can be used in the selection methods developed in Python. Two geoprocessing tools exist to provide this ability:

1. *Make Feature Layer*
2. *Make Table View*

The reader can refer to the ArcGIS Help for each tool under **Data Management—Layers and Table Views** for more details. The example below illustrates the general process of creating a feature layer and then using it in a selection routine.

```
if arcpy.Exists(street_layer):
    arcpy.Delete_management(street_layer)

arcpy.MakeFeatureLayer_management(streets, street_layer)

arcpy.SelectLayerByAttribute_management(street_layer, "NEW_SELECTION")

num_features = arcpy.GetCount_management(street_layer)

print "Number of Selected Features: " + str(num_features)
```

Two variables `streets` and `street_layer` have already been created (not shown). The `streets` variable points to the “feature class” on disk; the `street_layer` variable is just a string name representing an arbitrary name for the street feature layer.

The `MakeFeatureLayer` routine is implemented which creates the feature layer (`street_layer`) from the input feature class (`streets`). A check is put into place using the `Exists` and `Delete` routines to delete the feature layer if it already exists. The `SelectLayerByAttribute` routine is then implemented to select features from the input feature layer (remember, the feature layer is the required data type for the selection routine). In this case, since the “where clause” parameter is not used, all of the features are selected from the feature layer. The next couple of lines of code obtains the number of selected features and prints the total number of selected features to the Python Shell.

Query Syntax

The second key option to properly code and use the selection methods is developing the proper query syntax for the “where clause” parameter. Many new and seasoned programmers are often stumped at developing the proper syntax for data queries because coding the syntax requires the proper query structure and often the use of variables within the query itself. The practice of developing and testing specific components of a query (often through the use of print statements) is commonly implemented. Adding to the challenge is understanding the format of the data, field (attribute) name structure, use of “wild card” characters, and possibly the type of database (shapefile, file geodatabase, personal geodatabase, Oracle, SQL Server, Postgres, etc).

Query syntax for Python scripting follows the same syntax rules when constructing a query in ArcMap. The code developer needs to be aware of the data format being used in the geoprocess as well as the values and the Python syntax requirements that make up the conditions of the query. In some cases, special characters such as the Python escape character (“\”), a wildcard character (e.g. % or *), or a mathematical expression may be needed. Designing query syntax can take the code developer considerable time, especially when the queries are complex.

Building the Query Syntax

As mentioned above and shown in **Demo 5a**, a query expression (sometimes referred to as a “where clause”) is often necessary to subset records. It is necessary to spend some time discussing how to build query syntax using different kinds of data types since the syntax can change with different data types. Building and troubleshooting query syntax is often a challenging and sometimes a time consuming practice when developing Python scripts for ArcGIS. Search on “**arcgis query expressions**” to obtain ArcGIS Help documents to see more details on building query syntax in ArcGIS using different data types. Queries will be discussed again in Chapter 6 which introduces the concept of cursors to search, insert, and update records in a feature class or table.

Field Name Syntax

When code developers are writing scripts using “file-based” data such as shapefiles, file geodatabases, SDE geodatabases, or even ArcGIS Server feature or image service layers, field names will be enclosed in double quotes.

For example, as used in **Demo5a** below, the Sacramento streets data is a shapefile and the query used in the `MakeFeatureLayer` tool uses double quotes for the field “CLASS.” Hence the field name used in a Python query is:

```
query = """CLASS" = 'H'"""
```

Notice that the full query string is bounded by “triple double” quotes and is a more modern method of generating string syntax for queries versus using single or double quotes with escape (“\”) characters.

For personal geodatabases, field names should be enclosed in square brackets.

```
query = """[CLASS] = 'H'"""
```

*Esri encourages the use of file geodatabase over the personal geodatabase format because it can store larger data sets and the query syntax is more consistent with other file-based data formats.

Developing and Processing Strings in Query Expressions

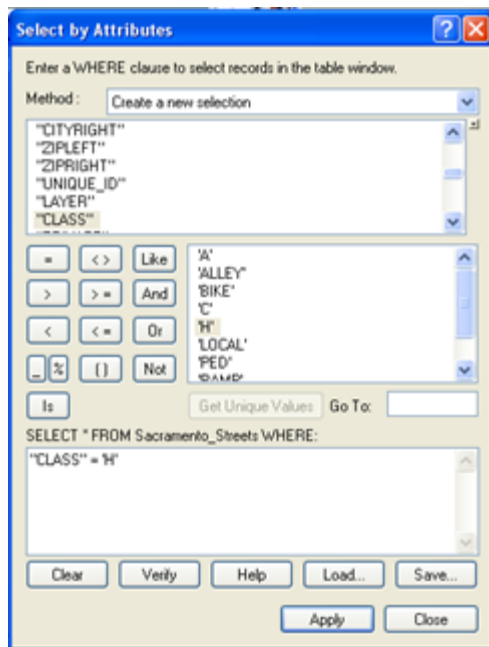
Strings for field names and values are case sensitive, so the code developer needs to make sure of the case for each attribute and value. This is another area that often challenges the code developer.

NOTE: Standardizing field names, values, and case is good practice when designing a geodatabase.

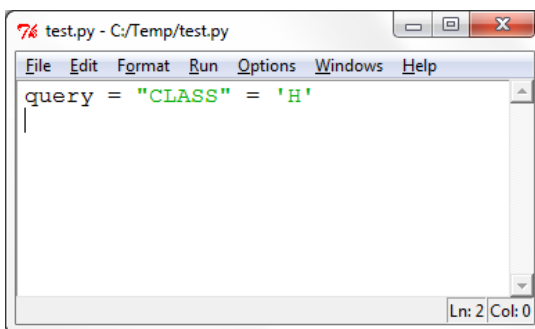
String values (right side of the equal sign) are bounded by single quotes, hence:

```
“CLASS” = ‘H’ or [CLASS] = ‘H’
```

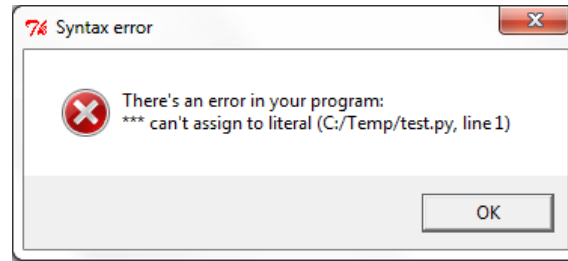
The above is typical syntax when using the Query Builder in ArcMap (for example in the `SelectLayerByAttribute` routine). Often new Python code developers incorrectly believe they can create and check the query in a Query Builder dialog box and then copy/paste this syntax into a developing script.



If the above syntax was copied from the ArcGIS query builder and pasted directly into a Python script, the following syntax would appear.



The `query` variable shows the word “CLASS” and the character ‘H’ as strings with a second equal sign between the two strings. (The reader may need to try this out using an empty Python script to see this effect). Python treats the “CLASS” and ‘H’ as strings, whereas the equal sign between “CLASS” and ‘H’ is not part of the string that is supposed to be set to the `query` variable. If the script was checked with the above syntax, the following error would appear.



In this case Python does not know how to assign the `query` variable because there are two different equal signs.

To alleviate this problem, remember that strings (even those representing queries) can be contained within two sets of “triple double” quotes. The proper syntax can then be created.

```
query = """CLASS" = 'H'"""
```

The syntax above represents the same syntax that is found when using the Query Builder tool in ArcGIS for the selection tools and is the format that the code developer expects to create.

```
"CLASS" = 'H'
```

Common Operators in Queries

As shown above the equal sign is commonly used to query a specific value or values from a list of possible values found in an attribute table. In addition, the following *string operators* are often used when developing query strings for Python and ArcGIS.

LIKE – often used with wild card characters to perform partial string character searches. For example `"Street_Name" LIKE 'M%'` will return all streets names beginning with the letter “M”.

<> – indicates not equal to. This will return all records except the value on the right side of the `<>`.

>, **<**, **>=**, and **<=** can also be used with character strings. For example, `"Street_Name" >= 'M'` will return all street names that begin with the letter M through Z.

Wildcard Characters

Wildcard characters are often used in query strings to simplify syntax and to include a variety of values that otherwise would require much more complex and lengthy queries to be written. Wildcard values also differ depending on the data type being used in the script.

For file-based data, file geodatabases, and SDE data layers, the following wildcard characters are used:

% - represents any number of characters. See the example above under the LIKE common operator

_ - (underscore) a wildcard that represents a single character. This value can be placed anywhere in a character string

For personal geodatabase, the equivalent characters are:

* – for any number of characters

? – for a single character

NULL Values

In ArcGIS and databases in general NULL values carry special meaning. NULL represents “No Data” or empty values. NULL values are not zero. NULL values can be used both in vector, tabular, or image data. To use NULL values effectively in Python and ArcGIS, one needs to know how to use them when querying data. When building queries, it might be helpful to query data that is set to NULL or represents NULL (or not). The following shows how NULL can be used to develop queries.

`“Street_Name” IS NULL` – will query data where the “Street_Name” attribute does not contain a value

`“Street_Name” IS NOT NULL` – will query data and return records that do contain a value for the “Street_Name” attribute.

In ArcGIS and Python, single quotes are not needed to “bound” the word NULL.

The Python syntax would be:

```
query = """Street_Name" IS NOT NULL"""
```

Numerical Expressions in Queries

Expressions using arithmetic notation can be used to query numerical values.

For example,

```
"City_Pop" > 10000
```

can be used to query a City_Pop attribute whose values are greater than 10000, where City_Pop is a numerical type attribute field that holds numerical values.

Standard numerical comparison operators such as =, >, <, >=, <=, <> and the keyword BETWEEN can be used in query strings. In Python the syntax will appear like this:

```
query = """City_Pop" > 10000"""
```

Using Calculations in Queries

The use of numbers in expressions can be expanded by the use of arithmetic operators such as +, -, *, /, and the use of parentheses to group and provide precedence to numerical operations.

For example,

```
"City_Pop" / "Area" <= 100
```

Or

```
"FID" < ("UNIQUE_ID" / 100) + 5
```

In Python a query like this might look like:

```
query = """FID" < ("UNIQUE_ID" / 100) + 5"""
```

Combining Expressions

Up to this point the examples have shown simple query expressions. In many cases, there may be multiple conditions that a query must meet. In these cases the code developer can use key words to combine expressions.

AND – Both query conditions must be met for the query result to be true to select or return records

OR – At least one condition must be met for the entire query to be true to select or return records

Combining multiple query expressions into a compound query is another area that a code developer can spend considerable time working through to create the right syntax and return the correct set of records or features.

A good approach for making progress on more complex queries is to start with one condition at a time, validate the syntax and returned set of queried data, and then build the next portion of the query. Compound query strings can be very long, may include a combination of character strings, numbers, NULL values, variable names, etc. providing plenty of opportunity for errors.

Recommendations

Building complex queries is one place that spending time within ArcMap to manually work through the query steps can benefit the development of coding logic and syntax. If the code developer can see what actual attributes and values are needed as well as to see the results of a query in selected records that can be interactively viewed and examined while in ArcMap, the programmer can gain insight to the specific ArcGIS and Python syntax (such as the data type, wildcard and escape characters, and keywords) that will be needed to write the queries correctly. Another recommendation is the use of print statements within the Python script to report back the results of the number of records or features which can be compared to the results from manual processes developed using ArcMap.

Other Query Syntax

The following topics are not specifically covered in *A Python Primer for ArcGIS*, however, readers are urged to review the ArcGIS Help topic **SQL reference for query expressions used in ArcGIS**. Some of these are dependent on the type of database being used and are considered more advanced procedures.

Dates – e.g. Date, Date and Time, Time, Day, Month, Year, parts of date, etc. Specifically see the section for Date and Time.

Numerical Functions – e.g. sin, cosine, round

String Functions – e.g. finding the left most, right most, or middle part of a string value

Some of these may be discussed in other chapters, but not in a comprehensive way.

Make Feature Layer, Make Table View, and Selecting Data

One may notice through reviewing the ArcGIS Help documentation and this text that the `MakeFeatureLayer`, `MakeTableView`, and `SelectLayerByAttribute` routines all contain a “where clause” parameter. Since the `MakeFeatureLayer` or `MakeTableView` is a prerequisite to implement the `SelectLayerByAttribute` routine, one may ask “Should the query be implemented in the `MakeFeatureLayer` (or Table View) or as part of the `SelectLayerByAttribute` routine?” Either option is correct, however, implementing the “where clause” as part of the `SelectLayerByAttribute` would be more appropriate, since the selection routine contains a number of “selecting options” that are not available as part of the “Make” routines. The “Make” routines are simply creating a “feature layer” from a “feature class” just as would be produced when an analyst added a feature class to the Table of Contents (or set the *Definition Query* parameter on the layer in the Layer Properties). When the data set is initially added, the feature layer does not contain any selected features; this is also true when the “Make” routine is created just before a selection routine—the selection routine requires a feature layer (not a feature class) to function properly.

Selecting Data

The main reason to select data is so that other geoprocessing operations can occur on a subset of information (geographic features or tabular records). For example, the user may need to calculate a value for the selected records or export the selected data to a new feature class or table. Selecting data (either by attribute or by location) uses the structures mentioned above.

Programming the *SelectLayerByAttribute* Routine

As shown above, the `MakeFeatureLayer` routine is required to implement the `SelectLayerByAttribute` routine. In addition, a query string is often created to limit the selection of features or records. The script below is very similar to that shown above, except a simple query string has been created and then used in the selection routine.

```
# Select features using an attribute query

if arcpy.Exists(street_layer):
    arcpy.Delete_management(street_layer)

arcpy.MakeFeatureLayer_management(streets, street_layer)

query = "\"CLASS\" = 'H'\""

arcpy.SelectLayerByAttribute_management(street_layer, "NEW_SELECTION",
query)

num_features = arcpy.GetCount_management(street_layer)

print "Number of Selected Streets: " + str(num_features)
```

Notice in this script that a `query` variable is created that is set to the string `"CLASS" = 'H'` and is used in the selection routine. The selection will only “select” streets that meet the criteria of the query string and will result in a smaller number of records than that of the selection created earlier in this section.

NOTE: When creating standalone scripts, the result of the “selection” routine will NOT show any actual features “selected” like that found in ArcMap because the script is run outside of ArcMap (ArcMap or ArcCatalog do not have to be open to run standalone Python scripts). Compare the above to the Select Layer By Attribute geoprocessing routine in ArcMap using the `"CLASS" = 'H'` as the query parameter in the tool and review the selected records in the table and on the map viewer.

Select Data by Location

A different method of selecting records can be performed by evaluating the spatial relationship between one geographic layer and another. A number of built-in operations exist to extract or merge geospatial data that has a spatial relationship such as clip, buffer, intersect, union, or

spatial join among others. One option that these functions do not have is the ability to select records from existing data and perform a subsequent operation on the selected records. For example, the buffer routine cannot select features on a separate feature class. The `SelectLayerByLocation` routine, however, can select features based on the spatial coincidence of an existing boundary or by applying a “virtual radius” around features from the same feature class. Both operations can be performed in a single step. Shown below is the general syntax for the `SelectLayerByLocation` routine.

```
SelectLayerByLocation_management (<input_feature_layer>,
{spatial_relationship_type}, {select_features}, {search_distance},
{selection_method})
```

The only required parameter is an input feature layer. Table views are not used in the `SelectLayerByLocation` routine, since it operates using a spatial relationship between two layers. The optional parameters include:

Spatial_relationship_type – the default is INTERSECT, but can be one of many different types of spatial relationships and depends on the feature type of the datasets. See the ArcGIS Help for the `SelectLayerByLocation` routine.

Select_features – the feature layer that will be used to perform the feature selection on the input feature layer. The select features layer could have “selected” features from a previous processing step, such as from a `SelectLayerByAttribute` routine.

Search_distance – the distance from features to expand the selection on the input feature layer. This parameter will have a number and a unit type (e.g. 100 METERS).

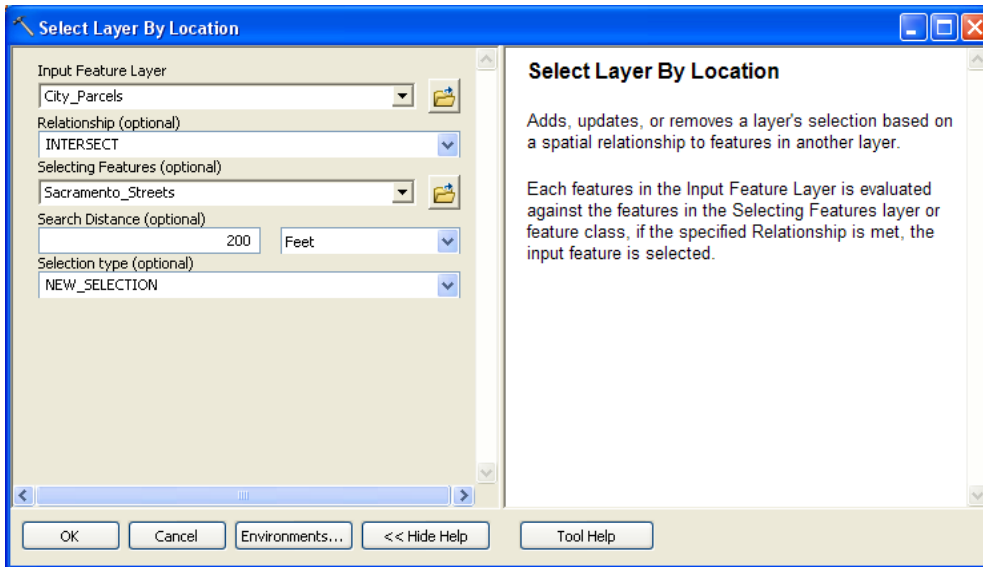
Selection_method – the type of selection. These choices are the same as the `SelectLayerByAttribute`, except for the “CLEAR_SELECTION” method.

If the optional parameters are not specifically added, the INTERSECT spatial relationship and “NEW_SELECTION” selection method will be used. In addition, a search radius of zero units will be applied.

To get an idea of how the `SelectLayerByLocation` tool works, the user can open the **Data Management—Layers and Table Views—Select Layer by Location** tool and add set the following parameters (see below). The reader can follow this example by using data from the `\PythonPrimer\Chapter05\Data` folder. In this example, the analyst wants to determine how many parcels are within 200 feet of highways.

NOTE: In this example *Sacramento_streets* is assumed to have only the highways selected which resulted from a `SelectLayerByAttribute` routine (see the previous section). The reader can implement the Select Layer By Attribute routine using the *Sacramento_streets* layer (from the

Chapter05\Data folder) as the input and use the query “CLASS” = ‘H’ as the query string to select highways from the street layer before implementing the Select Layer By Location routine shown below.



In the Select Layer by Location tool above, the following parameters are set:

Input Feature Layer - **City_Parcels** layer is the input feature layer that will have the selection applied to it.

Relationship - INTERSECT, which tells ArcGIS to select all parcels that are spatially coincident with the “Selecting Features” parameter. INTERSECT is the default. Check the ArcGIS Tool Help for other options.

Selecting Features – the **Sacramento_Streets** is the feature layer used to actually select features in the input feature layer. NOTE: It is assumed in this example that “Highways” were selected from the Sacramento_Streets layer in a previous step (such as Select Layer By Attribute).

Search Distance (and units) – the value used to determine an additional search radius around the selecting features layer (**200 Feet**)

Selection Type – **NEW_SELECTION**, the type of selection being applied to the input feature layer. NEW_SELECTION is the default. See the ArcGIS Tool Help for more details.

After clicking OK, parcels that are within 200 feet of highways will be included in the selection.

Programming the *SelectLayerByLocation* Routine

In Python, the reader can expand upon the geoprocessing tasks presented above by adding a few lines of code to implement the `SelectLayerByLocation` routine. Since the `SelectLayerByLocation` routine is selecting data from a separate feature layer, two new variables are created, one for the parcel feature class and one for the feature layer. An additional variable is provided that holds the search distance. These are defined toward the top of the Python script to keep the script organized.

```
parcels = "City_Parcels.shp"
parcel_layer = "parcel_layer"
search_distance = "200 FEET"
```

Because the `SelectLayerByLocation` routine requires a feature layer, a new `MakeFeatureLayer` line is added to create a feature layer from the *City_Parcels.shp* feature class. The code below shows the script with these additions. Formatting has been modified to fit the page.

```
...
# select layer by attribute code goes here
# parcel related variables assigned at top of script

if arcpy.Exists(parcel_layer):
    arcpy.Delete_management(parcel_layer)

arcpy.MakeFeatureLayer_management(parcels, parcel_layer)

arcpy.SelectLayerByLocation_management(parcel_layer,
    "INTERSECT", street_layer, search_distance, \
    "NEW_SELECTION")

num_features = arcpy.GetCount_management(parcel_layer)

print "Number of Selected Parcels: " + str(num_features)
```

Notice the `SelectLayerByLocation` routine includes the `parcel_layer` as the layer to select features from by “intersecting” the parcels with the `street_layer` (remember, highways have already been selected in a previous step). In addition, parcels will also be included in the selection if they are within 200 ft of the “intersected” features (i.e. selected streets). The next two lines of code set up obtaining the number of resulting selected features and then the number is printed to the Python Shell.

Counting the Number of Records

The reader has already seen the use of the `GetCount` tool (**Data Management—Table—Get Count**) that is useful for determining how many features or records there are in a data set or selection set. As shown in several of the examples above the result of `GetCount` can be assigned to a variable which is then used in print statements. This can be useful to help code developers to track and check results from selection queries. The value from `GetCount` can also be used in subsequent processes such as setting a bound on a `for` loop or a number of iterations for a `while` loop. The reader will see the use of `GetCount` throughout this book and in the example scripts.

Creating a New Dataset

As mentioned above one possible subsequent operation after a select layer operation is to write the selected records to a new feature class or table. If a feature class is written, then the `CopyFeatures` tool (**Data Management—Features—Copy Features**) is used to write out both the geographic features and accompanying attribute table. If only the attribute table or a standalone table is desired, then the table operation `CopyRows` (**Data Management—Tables—Copy Rows**) is used. Each of these is demonstrated in the **Demo5b.py** script. See the ArcGIS Tool Help for the `CopyFeatures` or `CopyRows` tools as needed. An example code snippet is shown below. The formatting has been modified to fit the page.

```
if arcpy.Exists(out_parcel_fc):
    arcpy.Delete_management(out_parcel_fc)

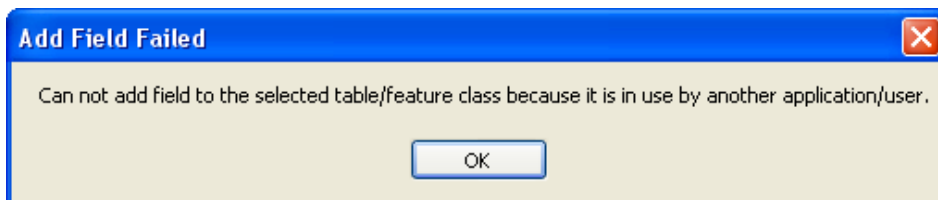
arcpy.CopyFeatures_management(parcel_layer,
                              out_parcel_fc)

print "Copied selected features from " + parcels + "
      to " + out_parcel_fc
```

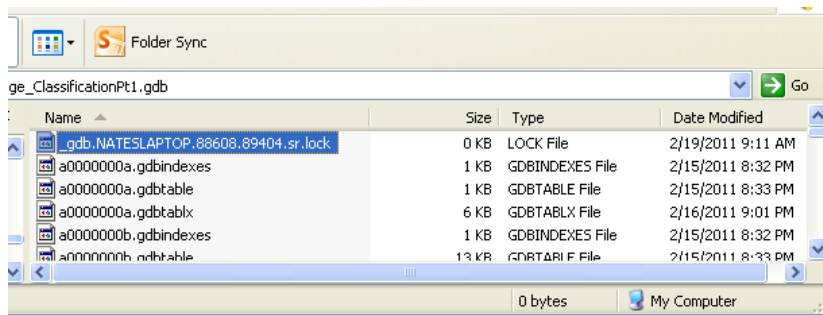
See the full set of steps for **Demo5b.py** in the **Demo5b** section below.

Data Locks

When analysts work with datasets where the data values will be created, updated or changed, or deleted some care is required to make sure that the dataset's integrity is maintained. Typically, an analyst is working in either ArcMap to interact with records or fields of data sets or ArcCatalog to add/delete fields, their data types, and possibly attribute domains. In either case, when an application accesses the dataset, effectively a data lock is placed on it indicating that another application cannot make changes (to either records, fields, data types, attribute domains, etc). For example, if an analyst has a streets layer open in ArcMap and then tries to access the dataset in ArcCatalog to add a field, the following message will appear.



With file geodatabases, when a feature class, table, raster, or other data set is accessed through an application, a .LOCK (lock file) will be added. The .LOCK file can be seen through Windows Explorer.



Essentially, data locking mechanisms are put in place as a measure of protection so that analysts do not inadvertently change their data and to provide a message back to the user letting them know the data is being used by another application or user. When one of the applications (such as ArcMap) is closed, the lock is released and changes can be made to the respective data set. Only one application can access the data if changes are to be made.

From a programming point of view, especially when developing code, similar kinds of measures are put into place, but are not always apparent to the code developer. In addition, since a code

developer often writes a geoprocess that creates a result, he or she often wants to review the data in ArcCatalog or ArcMap to make sure it was created properly. Running Python to process data is an “application” that uses geospatial data and thus, data locks can be created. Even if Python is the only application accessing geospatial data, data locks can persist with the following conditions:

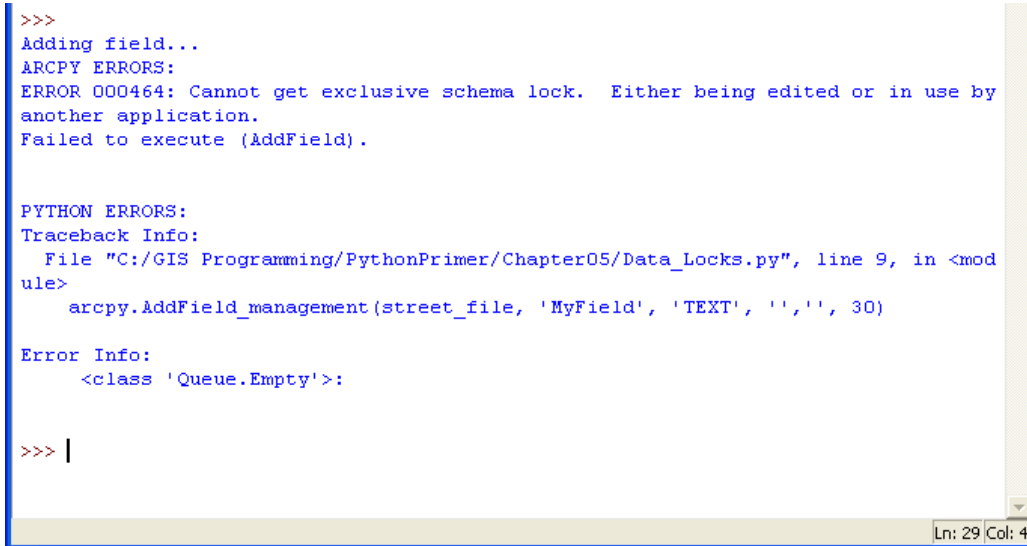
1. When a script successfully completes processing, but Python (IDLE and the shell) is not closed.
2. The Python script fails because of syntax or processing logic problems and the analyst attempts to re-run the script again without closing Python (IDLE and shell), even if changes are made to the script and it is re-saved.
3. The script creates/updates a data set and the analyst reviews it in ArcMap or ArcCatalog, Python is closed, but ArcMap or ArcCatalog remain open, and the analyst attempts to re-run a script that creates/updates the same data set that is open in ArcMap or ArcCatalog.

In order to prevent data locks that prevent code developers from reviewing or reprocessing data, the following procedure can be followed:

1. Do not have ArcMap or ArcCatalog open (any occurrence) when processing data using Python IDLE (or other script editor to run the script outside of ArcMap or ArcCatalog).
2. After Python successfully (or unsuccessfully) processes a data file, close Python IDLE and the Python Shell (all occurrences) before reviewing data derived from Python processes in either ArcMap or ArcCatalog. NOTE: If the code developer noticed error messages in the Python Shell, it is recommended to do a screen shot of the error or copy and paste the error message into a text editor before closing Python IDLE and the Python Shell so the error messages can be reviewed.
3. After Python successfully (or unsuccessfully) processes a data file and the code developer makes changes to the script or re-runs the script without any having made any changes, close Python IDLE and the Python Shell (all occurrences) before re-running the script.

One might think that using the `Exists/Delete` functions or the `overwriteOutput` environmental option (see the ArcGIS Help for the `overwriteOutput` routine) can be used in scripts and geoprocesses that takes care of data locking, but data locking has more to do with more than one application (or multiple occurrences of the same application) accessing the same data. Think of trying to edit the same Word or Excel document in more than one occurrence of Word or Excel. The application often reports to the user that the document is open in another location. With GIS data, ArcGIS, and Python, this is the same kind of issue.

The following error message is produced as a result of the data set being opened in ArcMap while attempting to add a field to it using a Python script.

A screenshot of a Python console window with a blue border. The text inside is as follows:

```
>>>
Adding field...
ARCPY ERRORS:
ERROR 000464: Cannot get exclusive schema lock. Either being edited or in use by
another application.
Failed to execute (AddField).

PYTHON ERRORS:
Traceback Info:
  File "C:/GIS Programming/PythonPrimer/Chapter05/Data_Locks.py", line 9, in <mod
ule>
    arcpy.AddField_management(street_file, 'MyField', 'TEXT', '', '', 30)

Error Info:
  <class 'Queue.Empty'>:

>>> |
```

At the bottom right of the console window, there is a status bar that reads "Ln: 29 Col: 4".

In this case, the code developer would need to exit the ArcMap document that is using the dataset before re-running the script.

Data locking is mentioned in this chapter so that the reader is aware of the concept and how it can impact the development and processing of code as well as attempting to review data derived from a Python script so that troubleshooting can take place. Data locking will occur more often when specific records are being updated and created (such as with the use of cursors in Chapter 6).

Summary

Chapter 5 discussed and reviewed the primary mechanisms for create and using queries in the `SelectLayerByAttribute` and `SelectLayerByLocation` routines, the two most common methods for querying and selecting data. Understanding the requirements of using the `MakeFeatureLayer` and `MakeTableView` routines in addition to constructing the proper syntax for queries are key to creating and using the selection methods. In many instances, the code developer will be required to construct a query to limit a selection, subset a set of records, or obtain a number of elements that will be used in subsequent geoprocessing steps. Properly building and using queries in addition to the use of the `MakeFeatureLayer` and `MakeTableView` are fundamental to many of the geoprocessing workflows to develop standalone Python scripts.

In addition, this chapter also introduced counting records and creating new datasets, which happens to be a common “next step” after selecting a subset of data. Data locking was also mentioned and will be a key element to pay attention to when the code developer tests and troubleshoots problems. Being able to recognize that ArcGIS and the Python Shell and IDLE can cause locks on data are important factors when attempting to solve problems with developing scripts, creating or modifying data, and or automating processing logic.

Chapter 5 Demos

The Chapter 5 demos are divided into two sections. **Demo 5a** focuses on using the `MakeFeatureLayer` routine and showing how to use the routine with and without the `query` parameter. **Demo 5b** focuses on extending these concepts to selecting features from a dataset. A primary objective of the reader is to gain an understanding of how to properly construct attribute queries using the proper Python and ArcGIS syntax so that the query can be used in the `SelectLayerByAttribute` and the `SelectLayerByLocation`, the two commonly used methods of selecting data records.

Demo 5a: Create a Feature Layer with and without a Query

This demonstration takes a look at how the Make Feature Layer Tool is used to create a feature layer from a feature class with and without a query. Refer to **Demo 5a.py** for the actual Python script. See the respective help for the various tools described below for additional information.

The concepts illustrated in this demo are:

ArcGIS Concepts

Workspace
Building a Query
Create and Use a Feature Layer
Counting Records
Exists Method
Delete Method

Python Concepts

Variable definitions
if statement
Use of `try:` and `except:`
Use of `import sys` and `traceback` modules

This demo uses the **Sacramento_Streets.shp** file that can be found in **\PythonPrimer\Chapter05\Data** folder. Open ArcMap or ArcCatalog to see the various attributes and values of the **Sacramento_Streets.shp** file if needed. A **Demo5.mxd** has been provided.

1. Start out by importing the `arcpy`, `sys`, and `traceback` modules. Add some commentary text to briefly describe the script.
2. Add in the `try` and `except` blocks. (HINT: Use the exception code found in **\Chapter01**).
3. Add a line to set the workspace and some variables to hold the values of the feature class (`streets`) and feature layer (`street_layer`). Note that the `street_layer` variable below is just a string of characters and can represent any name the developer chooses.

NOTE: The path to the \PythonPrimer\ChapterXX\Data may need to be modified, depending on where the reader copied the data for the workbook.

```
import arcpy, sys, traceback

# Change the path as needed
arcpy.env.workspace = "C:\\PythonPrimer\\Chapter05\\Data\\"

streets = "Sacramento_Streets.shp"

street_layer = "street_layer"

try:
```

Before creating a layer, it might be good to get an idea of how many features are actually in the feature class. This can be done by using the `GetCount` routine from the **Data Management—Table—Get Count** tool.

4. Add the lines of code to report back the number of features (records) in the **Sacramento_Streets** feature class.

```
# code from the previous step goes here

try:

    # print the number of records in the feature class
    result = arcpy.GetCount_management(streets)

    print "Number of features in the feature class " +
    streets + " : " + str(result)
```

The developer will find that `GetCount` returns the number of records in the feature class. (`GetCount` can also be used with a feature layer, table, or table view). The above code shows this value being assigned to the `result` variable. Since this value is assigned to a variable, the actual number of records can be used in other parts of a script. In this case the `result` variable is used in a print string to print the number of records back to the Python Shell. Notice also, that `result` is a number. In order to “concatenate” a number with a string, it must be converted (or cast) as a string, hence the use of `str(result)`. In addition, notice how the `streets` variable is used in `GetCount` (i.e. to get the number of records in the feature class (in this case) and in the print statement. The name of the feature class will be printed in the print statement as a result of using the `streets` variable.

5. Create a Feature Layer without a Query

After the number of records in the feature class is reported, the next section of code checks to see if the feature layer exists (and deletes it if it does) and then creates the feature layer from the feature class by using the `MakeFeatureLayer` routine. In addition, similar lines of code can be written to report back the number of records in the feature layer. The code format has been modified to fit the page. See **Demo5a.py** for the proper format. Note that this code is indented to match the code above within the `try` block.

```
if arcpy.Exists(street_layer):
    arcpy.Delete_management(street_layer)

# make the feature layer from the feature class
arcpy.MakeFeatureLayer_management(streets, street_layer)

# print the number of records in the feature layer
# without a query

result = arcpy.GetCount_management(street_layer)

print "Number of features in the feature layer " +
street_layer + " without a query: " + str(result)
```

Notice that the `MakeFeatureLayer` routine uses the feature class (`streets`) as the input and the name of the feature layer (the variable `street_layer`) as the output. These two parameters are the only required parameters to create a feature layer from a feature class. The optional parameters can be ignored for this routine (such as a query expression). *(Additional comments will be made later to describe using a “place holder” value for optional parameters to help maintain the correct order, number, and type of parameters in a geoprocessing function).*

For the `GetCount` and `print` statements notice that the `street_layer` variable is used instead of the `streets` variable. Using the `street_layer` variable allows `GetCount` to obtain the number of records from the feature layer (rather than the feature class) and subsequently print the number of records to the screen (or use the number for other geoprocesses).

6. Create a Feature Layer using a Query

The next set of lines will set up the ability to use a query in `MakeFeatureLayer` routine. Since the next step will create another feature layer for the same feature class, the developer can simply “re-use” the variable already created and used in the previous steps. The previous `MakeFeatureLayer` routine already used the name “`street_layer`” for the layer name. Attempting to “re-use” the same layer name will cause the program to result in an error

indicating the feature layer already exists. An option would be to set another unique variable to a unique feature name, however, this can be avoided by using an `if` statement to check if the feature layer name already exists using the `Exists` routine; if it does, then the feature layer name can be deleted by using the `Delete` routine. Using an `if` statement and checking to see if a data sets exists is common practice in writing Python scripts and can help in re-using variables and overwriting various datasets (feature classes, feature layers, tables, table views, work spaces, images, etc). Notice how indenting is used to make sure the `if` statement is processed correctly. The result and print lines are shown below to show continuity with the previous step.

```
# print the number of records in the feature layer
# without a query

result = arcpy.GetCount_management(street_layer)

print "Number of features in the feature layer " +
street_layer + " without a query: "+ str(result)

# check to see if the feature layer exists; if so,
# delete it

if arcpy.Exists(street_layer):
    arcpy.Delete_management(street_layer)
```

After the `if` block is written, the query expression string for the `MakeFeatureLayer` can be written. Shown in the code below is a variable that stores the syntax for the query. Notice the query statement lines up (i.e. the same indentation level) with the `if` statement.

```
if arcpy.Exists(street_layer):
    arcpy.Delete_management(street_layer)

# create a feature layer using a query

query = """"CLASS" = 'H'"""
```

Notice that the query variable is used as the query parameter in the `MakeFeatureLayer` routine below which results in a more concise script. Some code is shown to provide continuity with the previous step. The code has been formatted to fit the page.

```
# check to see if the feature layer exists; if so,
# delete it

if arcpy.Exists(street_layer):
    arcpy.Delete_management(street_layer)

# create a feature layer using a query

query = """"CLASS" = 'H'"""

# make a feature layer that also uses the query
# parameter

arcpy.MakeFeatureLayer_management(streets,
    street_layer, query)

# print the number of records in the feature layer with
# query

result = arcpy.GetCount_management(street_layer)

print "Number of features in the feature layer " +
street_layer + " with a query: "+ str(result)
```

The `MakeFeatureLayer` routine above shows that only street features that are of “class H” (Highways) will result in the feature layer. “CLASS” is an attribute where one of the values is ‘H’ and represents highways. Open ArcMap or ArcCatalog to see the various attributes and values of the **Sacramento_Streets.shp** file if needed. The same `GetCount` and print statement can be used to print out the number of records in the feature layer with the query. Notice that the query and subsequent lines are not indented like the `Delete` routine. These lines are NOT part of the `if` block that check to see if the feature layer exists. Actually, the query line could have been written before the `if` statement if the code developer desired.

7. Save, Check, and Run the script

If the reader has been writing the code for this demo, save the script and run the Check Module. Fix any syntax issues as needed. Once all of the errors are fixed, run the script from within IDLE. The print statements should print to the Python Shell reporting the number of selected features. If needed consult the **Demo5a.py** script to help fix problems. The **Demo5 script results.doc** document can be compared with the reader’s results.

Demo 5b: Select Features by Attribute

This demo expands the concepts of **Demo 5a** and implements the Make Feature Layer geoprocessing tool and uses it to perform both an attribute selection and a select by location operation. This demo illustrates the following concepts:

ArcGIS Concepts

- Building a Query
- Creating and Using a Feature Layer
- Select data by Attribute
- Select data by Location
- Counting Records
- Creating a Feature Class
- Creating a Standalone Table

Python Concepts

- `if` statement
- Line continuation character ("`\`")
- New line character ("`\n`")
- Use of `try` and `except`
- Use of `import sys` and `traceback` modules

This demo uses the **Sacramento_Streets.shp** and **City_Parcels.shp** files that can be found in **\PythonPrimer\Chapter05\Data** folder. Open ArcMap or ArcCatalog to see the various attributes and values of the shapefiles if needed. Refer to the Chapter 5 text above for additional discussion and commentary. Two files are created as output in this demo, **out_parcels.shp** (contains the selected features and attribute table from the Select Layer by Location routine) and **out_street_attributes.dbf** (contains the selected attribute records from the Select Layer by Attribute routine).

1. Start the script by importing the `arcpy`, `sys`, and `traceback` modules. Add some commentary to the beginning of the script.
2. Add the `try:` statement and copy the exception code to the `except:` block from the **Exception.py** script found in **\Chapter01**.
3. Add the line for the workspace to the data path **\PythonPrimer\Chapter05\Data**.

NOTE: The path to the \PythonPrimer\ChapterXX\Data may need to be modified, depending on where the reader copied the data for the book.

4. It is also a good idea to put in some pseudo-code to outline some of the tasks that are expected in the script.

Remember to follow proper indentation and case.

The **Demo 5b** script should look similar to this up to this point. The code has been formatted to fit the page. See the demo code for the proper formatting.

```
'''
Example using the Make Feature Layer tool...
'''

print 'Demo 5b - Select Layer by Attribute and
Location\n'

import arcpy, sys, traceback

arcpy.env.workspace = "C:\\PythonPrimer\\Chapter05\\Data\\"

try:

    # 1. Make a feature layer for streets

    # 2. Create a query statement to select highways

    # 3. Select by attributes using the query

    # 4. Select parcels that are 200 ft from highways

    # 4a. Make feature layer

    # 4b. Select Parcels

    # 4c. Get the number of selected records and
        create a print statement for each

    # 5. Write out selected parcel features to file
        (SHP)

    # 6. Write out selected street attributes to a
        table file (DBF)

except:

    #http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#//0
    02z0000000q000000

    tb = sys.exc_info()[2]
    tbinfo = traceback.format_tb(tb)[0]
```

To get started with writing the different geoprocesses it is a good idea to research and review the specific geoprocessing tool help. Refer to the ArcGIS Help as needed.

For the Make Feature Layer, a feature class and a feature layer are required parameters. It is probably a good idea to define some variables for both the feature class and feature layer.

NOTE: When first starting out with writing code, it might be a good idea to write in the “hard coded” values to get an idea of what they might be. Later, when reviewing the script, variables can be created, especially if the same hard coded values are repeated. This is an indicator that a variable might be a good idea.

5. Create two variables above the `try:` statement.

```
streets = "Sacramento_Streets.shp"
street_layer = "street_layer"
```

6. Add the Make Feature Layer tool using the variables as parameters in the routine inside and properly indented in the `try` block. The code has been formatted to fit the page. Refer to the demo for the proper syntax

```
try:

# 1. Make a feature layer for streets

    if arcpy.Exists(street_layer):
        arcpy.Delete_management(street_layer)

    arcpy.MakeFeatureLayer_management(streets, street_layer)
```

Next, since the Select Layer by Attribute will require the highways to be selected, a variable will be set up to hold the query statement. This is good practice so that the query statement can be isolated from the Select Layer by Attribute routine and keep the code “clean and concise.” The query statement is at the same indentation level as the `MakeFeatureLayer` line above.

7. Create a variable for the query and assign the variable to the proper query syntax.

NOTE: Creating the correct syntax may require the use of ArcMap and the respective data layers (in this case the **Sacramento_Streets.shp** file) to test the query statement logic. Open up ArcMap and add the **Sacramento_Streets.shp** file. Run the Select Layer by Attribute tool or the Select Layer by Attribute option from the Selection menu. The query should use the "CLASS" attribute to select all segments that have a value of 'H'. Hence, the query should look like the following within the tool.

"CLASS" = 'H'

This same syntax will be used to set up the query statement in Python. See the commentary within this chapter that discusses the proper Python syntax for the query expression. Remember, the "triple double" quotes often surround the entire query string that is set to the variable.

```
# 2. Create a query statement to select highways

# Create query statement
query = """CLASS" = 'H'"""
```

8. Add the Select Layer by Attribute code.

Now that a query statement has been defined, the Select Layer by Attribute routine can be added. Review the tool help as necessary and add the proper parameters. Since this is the first time using this tool, the selection method will be "NEW_SELECTION". The code has been formatted to fit on the page. See the demo for the proper syntax.

```
# 3. Select by attributes using the query

# Select features by attribute using query
arcpy.SelectLayerByAttribute_management(street_layer,
"NEW_SELECTION", query)
```

9. Set up parameters for the Select Layer by Location.

Reviewing the ArcGIS Help on Select Layer by Location, the developer finds that another feature layer is required. Before the Select Layer by Location can be implemented, another Make Feature Layer routine is needed.

- a. Add a Make Feature Layer line that will create a feature layer from the **City_Parcels.shp** file.
- b. Add two new variables for parcels similar to those above for streets. Put the variable definitions above the try: block.

```
parcels = "City_Parcels.shp"
parcel_layer = "parcel_layer"
```

- c. Use the variable definitions in the Make Feature Layer similar to those as the **Sacramento_Streets** layer. Notice that the “pseudo-code” is slightly modified to include the new step (4a). Updating and making notations to the pseudo-code is good practice to keep in-line code documentation up-to-date. Adding comments with date stamps can indicate when additions and modifications to code were made.

The code has been formatted to fit on the page. See the demo for the proper syntax.

```
# 4. Select parcels that are 200 ft from highways

# 4a. Make feature layer

if arcpy.Exists(parcel_layer):
    arcpy.Delete_management(parcel_layer)

arcpy.MakeFeatureLayer_management(parcels, parcel_layer)
```

10. Add the Select Layer by Location line.

Add the Select Layer by Location line (4b). The default spatial relationship “INTERSECT” is used as well as the default selection method “NEW_SELECTION” since this is a new selection on the parcel layer. Make sure to use the parcel layer (`parcel_layer`) as the *Input Feature Selection* and the street layer (`street_layer`) as the *Select Features Layer*. The user can “hard code” the search distance parameter (‘200 FEET’) or can create a variable that is assigned to the character string (‘200 FEET’). The author has chosen the latter which allows a convenient way to change the search distance (and it is placed near the top of the script with other variable definitions).

```
search_distance = '200 FEET'
```

Notice that the line continuation character (“\”) is used at the right of the select line, since the Select Layer by Location line is a little longer than the other lines. This character makes it convenient to keep long lines compact to make the code easier to read.

```
# 4b. Select Parcels

arcpy.SelectLayerByLocation_management(parcel_layer,\
    "INTERSECT", street_layer, \
    search_distance, "NEW_SELECTION")
```

11. Count the number of records in each selection and print to the Python Shell (4c).

It might be a good idea to make sure that the selection code is selecting the expected number of records. The `GetCount` method can be used to obtain the number of selected records and assign them to a variable. Subsequently, this variable can be used in a print statement to report back to the code developer the number of records to the Python Shell.

Add the `GetCount` code for both feature layers and a print statement. Note that the “result” variable is “cast” (i.e. converted) to a string so that it can be properly concatenated with the other text in the print statement.

Note the use of the new line character (“\”) and the new line feed continuation character (“\n”) in the print statements. The new line feed continuation character will break the long print statement into two lines. The “pseudo-code” has also been updated. The code has been formatted to fit on the page. See the demo for the proper syntax.

```
# 4c. Get the number of selected records and create a print
# statement for each

# print the number of selected records in the respective
# layers with a query

result = arcpy.GetCount_management(street_layer)

print "Number of selected streets in the street layer " +
street_layer + " with a query: " + str(result)

result = arcpy.GetCount_management(parcel_layer)

print "Number of selected parcels in the parcel layer " +
parcel_layer + \
"\n within a distance of " + search_distance + ": " + str(result)
```

12. Write out the selected parcels to a new feature class (**out_parcels.shp**).

Add the `CopyFeatures` line to the code to write out the selected parcels to a new feature class. Define a variable at the top of the code (above the `try` block) and assign it to the name of the output shapefile (**out_parcels.shp**). A print statement has also been added to report that the file has been created.

```
out_parcel_fc = "out_parcels.shp"
```

```
# 5. Write out selected parcel features to file (SHP)
```

```
# Copy the selected polygon features to a new shapefile
```

```
if arcpy.Exists(out_parcel_fc):
```

```
    arcpy.Delete_management(out_parcel_fc)
```

```
arcpy.CopyFeatures_management(parcel_layer, out_parcel_fc)
```

```
print "Copied selected features from " + parcel_fc + "  
    to " + out_parcel_fc
```

13. Write out the selected street attributes to a new standalone table (**out_street_attributes.dbf**).

In this case, only the selected attributes (not the geographic features) are written to a standalone table (a dBase formatted table). A variable is added to the top of the script that is assigned to the value of the output table name. The `CopyRows` routine is used to write the selected rows to a new table. A `print` statement is added to report that the output table has been created. The code has been formatted to fit on the page. See the demo for the proper syntax.

```

out_street_attributes = "out_street_attributes.dbf"

# 6. Write out selected street attributes to a table file # (DBF)

# Copy the selected street segment attributes to a dBase
# table

if arcpy.Exists(out_street_attributes):
    arcpy.Delete_management(out_street_attributes)

arcpy.CopyRows_management(street_layer, out_street_attributes)

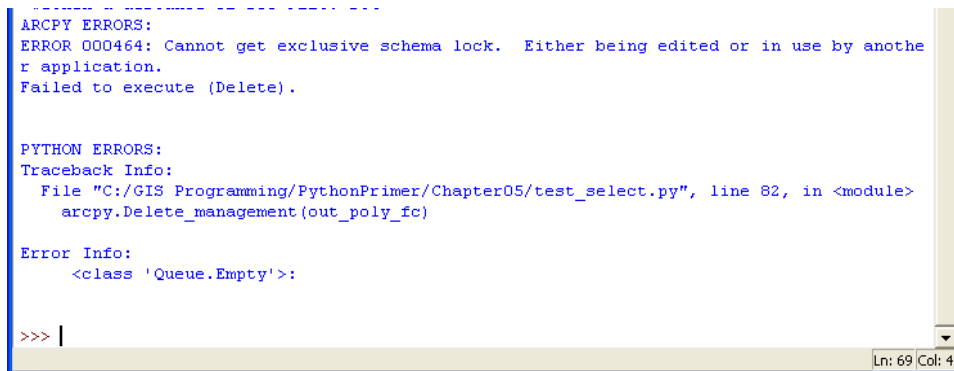
print "Copied selected attributes from " + streets + " to
" + out_street_attributes

```

At this point, the script is complete. The code developer can Save and Check the script. Fix any syntax errors that occur. Make sure to review the ArcGIS Help for any of the functions and use ArcMap to assist in building the proper query syntax.

Notice the use of the `Exists` and `Delete` statements to perform data clean up as the script progresses and if the code developer needs to run the script multiple times.

The figure below shows an example of an error code if the existing data cannot be deleted.



```

-----
ARCPY ERRORS:
ERROR 000464: Cannot get exclusive schema lock. Either being edited or in use by another application.
Failed to execute (Delete).

PYTHON ERRORS:
Traceback Info:
  File "C:/GIS Programming/PythonPrimer/Chapter05/test_select.py", line 82, in <module>
    arcpy.Delete_management(out_poly_fc)

Error Info:
  <class 'Queue.Empty'>:

>>> |
Ln: 69 Col: 4

```

Check the `\Chapter05\Data` folder to verify that the feature class and table were created. After closing Python and the Python Shell, open up ArcMap or ArcCatalog to review the data.

The **Demo5 script results.doc** document can be compared with the reader's results for this demo.

Exercise 5 - Select Features by Attribute/Location and Write Data to a New Feature Class

Exercise 5 will provide the opportunity for the code developer to put into practice some of the concepts illustrated in Chapter 5.

Write a Python script with the following conditions. Use the demo scripts, chapter content, ArcGIS Help, and Python resources as needed.

Review the **Sacramento_Streets.shp** file and **Sacramento_Neighborhoods.shp** file in the **Exercise5.mxd**. You will only need to use the ArcMap Document and look at the data and attributes for your review. You will not need the ArcMap document to write the script.

Write a script with the following conditions. The goal is to have a new feature class that only contains the local streets that are located within (or touching) a neighborhood of your choice.

1. Create a selection on the Neighborhoods shapefile that uses a query variable that queries one of the neighborhoods using the "NAME" field that is covered by the Sacramento Streets file. Simply use one of the Neighborhood names in the query.
2. Using the Selected neighborhood from above, use it to "spatially select" the streets within the neighborhood.
3. Using a query, select only the "LOCAL" streets from the previous selection that are located within the selected neighborhood. The query will need to use the "CLASS" attribute in the streets data.
4. Write a print statement that indicates the number of selected streets
5. Write these selected street features to a new feature class in the **\MyData** folder. Use the shapefile format.
6. Make sure to use variables, queries, and feature layers appropriately
7. Make sure to use `Exists` and `Delete` functions as necessary
8. Use the `try` block and exception code from previous exercises

Extra

Do the same as above, but for the **Z'berg Park** neighborhood and collector street types (CLASS value is 'C' for collectors). Write out a feature class and a separate standalone table of the selected street features. NOTE: Make sure to check the syntax in the **Query Builder** before writing the query syntax for the Z'berg Park neighborhood. Also, refer to the ArcGIS help document for **SQL reference for query expressions used in ArcGIS**. A single quote in a text string must be escaped with a second single quote, so Z'berg Park becomes Z' 'berg Park when writing the query string in Python.

The author strongly recommends that the Exercise is working first before attempting the **Extra** section.

Chapter 5 Questions

1. What is the difference between the following:
 - a. Feature Class
 - b. Feature Layer
 - c. Table
 - d. Table View
2. What are the following used for with developing queries?
 - a. % (non numeric)
 - b. * (non numeric)
 - c. LIKE
 - d. AND
 - e. OR
3. How are NULL values used in query strings? Give a query example using querying for NULL values.
4. What Toolbox and Toolset contains the Select Layer tools and Make Feature Layer or Make Table View Tools?
5. What is the default selection type for the Select Layer By Attribute?
6. What is the default selection type for the Select Layer By Location?
7. Can a feature class be used as the input in a Select Layer By Attribute or Select Layer By Location? If no, what must be used?
8. What is the geoprocessing routine (tool) to create a new feature class from the selected features in a feature class?
9. What is the geoprocessing tool to create a new table from selected records in a table?

Chapter 6 Creating and Using Cursors and Table Joins

Overview

Cursors are common database constructs that are used to access, read, and update values in a database table (or in the case of GIS, a feature class attribute table). One can think of cursors as a means to “point to” a collection of data records (features or rows in a table) from which the programmer can systematically process individual records. For example, suppose a GIS analyst performed an attribute selection on a number of records or features and then wanted to apply a *different* value to one of the attributes (columns) or read a value from one of the attributes to then use as an input for a query or copy it into a different table. Performing this action by using the `Select Layer by Attribute` and `Calculate Field` functions can be time consuming and repetitive within ArcMap. Even in a Python script the selection and calculate tasks would need to be re-used for each unique combination of selection criterion and field calculation. But, by using cursors, a more logical series of steps can be performed allowing reading from and/or updating multiple attribute fields for a given record (or feature).

Another concept often used with implementing cursors is “table joins.” A table join is where attributes from two different sources (tables or feature classes) are connected to one another through a common attribute containing the same values. Cursors can read values from “joined” data and write them to a separate table or feature class or to summary or reporting tables. The latter part of this chapter will focus on how to programmatically create and use “joined” tables.

Data Access Module

Beginning at 10.1 Esri provides a Data Access (da) Python module that contains a number of routines to access data elements (using cursors) as well as working with lists of domains, replicas, subtypes, and versions. Also, the Data Access Python module provides functionality to convert tables and features classes to/from Python *numpy* arrays. Although not discussed in this book, *numpy* arrays are structures that provide efficient access to applying numerical operations to data sets (for example, statistical measures, linear algebra, and other complex

mathematical operations). Refer to the ArcGIS Help documentation as well as the Python documentation for more information on *numpy* arrays and how to use them within ArcGIS.

Cursors

The general concept of a cursor is to access one or more records from a database (tabular or geographic). Once this set of records is obtained, each cell within the record can be acted upon. Esri recommends using the cursors from the Data Access module because they perform more efficiently and the cursor parameters use more standardized Python syntax. The Data Access cursors can be used with ArcGIS 10.1.x and beyond and is the focus of this chapter. The “legacy” cursor structures can be used in ArcGIS versions 10.0.x and earlier. ArcGIS Help indicates that pre-ArcGIS 10.1 cursor structures will work in ArcGIS 10.1+ and suggests that the Data Access version of cursors be implemented in ArcGIS 10.1+ environments, since the older version of cursors may become deprecated in the future. See the appropriate ArcGIS documentation regarding legacy cursor routines and Python syntax. Code examples for ArcGIS 10.0 are included in the data for download package from the author’s website in the **Chapter06\legacy** folder.

The three types of cursors are covered in this chapter:

1. *Search* – read values from a record
2. *Insert* – create new records (and optionally, provide initial cell values)
3. *Update* – modify existing records

The search and update cursors also have a query parameter (“where clause”) that can be used to access a subset of data. If the query parameter is not used, then the cursor will operate on all of the records in a dataset.

The cursor is different from a “selected” set of records in that the records in a cursor are not “selected” where the selected records would appear highlighted in ArcMap. In addition, the records in a cursor are not going to be used in a spatial overlay such as a `SelectByLocation` operation.

Cursors found within the Data Access module for feature classes or tables use the general syntax:

Search Cursor

```
arcpy.da.SearchCursor(<in_featureclass or in_table>, <field_names>,
{where_clause}, {spatial_reference}, {explode_to_points},
{sql_clause})
```

Insert Cursor

```
arcpy.da.InsertCursor(<in_featureclass or in_table>, <field_names>)
```

Update Cursor

```
arcpy.da.UpdateCursor (<in_featureclass or in_table>, <field_names>,
{where_clause}, {spatial_reference}, {explode_to_points},
{sql_clause})
```

Illustrating Cursor Processing using the Search Cursor

The *search cursor* is used to access and read values from a data table. A typical use of a search cursor is to read values from specific rows (records) and columns (fields/attributes) from a database feature class or table and write them to another geodatabase feature class or table.

Given a set of records (i.e. all of the features or tabular records in a data set) the cursor creates a collection of records. The number of records in a cursor can be one or more and depends on the use of the query parameter of the cursor.

UNIQUE_ID	FULLSTREET	LAYER	CLASS
77245	I 5 NB	HIGHWAY	H
77246	I 5 SB	HIGHWAY	H
76250	I 5 NB	HIGHWAY	H
75721	US 50 WB	HIGHWAY	H
75598	HWY 99 SB	HIGHWAY	H
76209	HWY 99 SB	HIGHWAY	H
23041	L ST	CITY	LOCAL
23042	J ST	CITY	LOCAL

The shaded records from the table above indicate the group of records accessed by a search cursor that uses the query (where clause) parameter with the condition “CLASS” = ‘H’. The

code below shows the Python syntax for a search cursor that uses a query parameter that gets a collection of rows and then iterates over each row to print the street name and the unique id associated with the street segment. Refer to the **search_cursor_sample.py** script found under the **\PythonPrimer\Chapter06** folder. Note the code has been reformatted to fit the page. See the script file for the proper formatting.

```
try:
```

```
# set a variable for the "FULLSTREET" and "UNIQUE_ID"
# fields. These are used in the field_names parameter
# in the search cursor

fullstreet_field = "FULLSTREET"
uID_field = "UNIQUE_ID"

# set a variable to hold the query string

query = """CLASS" = 'H'"""

# Create a search cursor to access rows that
# have a road class of "H" - Highway. Obtain
# only the "FULLSTREET" and the "UNIQUE_ID" fields
# Get a collection (srows) of rows from the streets
# shapefile

with arcpy.da.SearchCursor(streets_shp, (fullstreet_field,
uID_field), query) as srows:

    for srow in srows:      # for each row in the cursor

        # assign a variable to store the value in the
        # FULLSTREET attribute (field)

        fullstreet = srow[0]

        # assign a variable to store the value in the
        # UNIQUE_ID attribute (field)

        unique_id = srow[1]

        # print the value of the variables
        # i.e. the value of srow[0] and srow[1]

        print fullstreet + ' ' + str(unique_id)
```

The first section of code defines two variables set to the specific field names of the attribute table. These fields will be used in the `field_names` parameter. A variable is also set to the syntax of the query statement so that only highway street segments are included in the cursor.

The line beginning with the word “with” represents the Python syntax for a search cursor that uses the query parameter (where clause). The resulting collection of records will be stored in the cursor object (the `srows` variable).

Once the collection of records is obtained, each record (row) of the cursor is processed using a `for` loop (in this case the value of the `FULLSTREET` and the `UNIQUE_ID` fields for each record are printed out to the Python Shell).

The general `for` loop structure was described in Chapter 2.

```
for <specific_elements> in <a_collection_of_objects>:
```

In ArcGIS the `for` loop is often used to iterate over the table rows, features of a geographic file, and specific elements of the ArcGIS “List” routines (See Chapter 7 and the ArcGIS Help on working with ArcGIS lists).

In order to access a specific element in a collection, a variable is used (e.g. `srow`) within the `for` loop. See the code above.

The shaded record below represents the first record (i.e. the specific row) when the `for` loop is initially processed by Python. Python interprets the `for` loop line as “get a row” (`srow`) from the collection of rows (`srows`).

UNIQUE_ID	FULLSTREET	LAYER	CLASS
77245	I 5 NB	HIGHWAY	H
77246	I 5 SB	HIGHWAY	H
76250	I 5 NB	HIGHWAY	H
75721	US 50 WB	HIGHWAY	H
75598	HWY 99 SB	HIGHWAY	H
76209	HWY 99 SB	HIGHWAY	H
23041	L ST	CITY	LOCAL
23042	J ST	CITY	LOCAL

The reader should note that when implementing cursors in Python, there is no “selection” done, but rather a method to access records for further processing. One can think of the cursor as an *in-memory* function to access data and then process it.

The above example uses a feature class directly in a search cursor. Feature layers, tables, and table views can also be used with cursors. See Chapter 5 for creating and using feature layers and table views.

To access a specific value (cell) within a cursor, a specific reference to a specific field (attribute column) is required. This is accomplished by identifying specific field names within the `field_names` parameter and then referencing them within the `for` loop. The code above specifies separate variables to refer to the `FULLSTREET` and the `UNIQUE_ID` fields which are then used in the syntax of the cursor. If more than one field is specified, the list of field names must be bounded by parentheses within the cursor syntax or a variable containing a list (a Python *tuple*) of field names needs to be specified and then substituted for the `field_names` parameter. A *tuple* is an unordered list of values and is a term commonly used in Python.

NOTE: To access all of the fields, use `"*"` in the `field_names` parameter. To improve data access and cursor performance, it is recommended that only the fields used in the cursor process should be listed in the `field_names` parameter.

The code above uses a list of two field names surrounded by parentheses. The code below shows how a Python list of field names can be used in a cursor's `field_name` parameter.

```
# set a variable assigned to a list (a tuple) of specific
# field names

field_names_list = ["FULLSTREET", "UNIQUE_ID", "LAYER", "CLASS"]

# substitute the field_names_list variable into the
# field_names parameter

with arcpy.da.SearchCursor(streets_shp, field_names_list, query) as
srows:
```

The attribute (column) value of a specific row is obtained by using an index to the specific field listed in the cursor. In the example above the values for a specific street name and the unique ID are obtained by using the following syntax:

```
fullstreet = srow[0]
unique_id = srow[1]
```

The specific row of the cursor is identified by the variable `srow`. The specific column (field) is specified by the index value `[0]` or `[1]`, where 0 references the `FULLSTREET` name attribute (i.e. the first field in the list of field names) and 1 references the `UNIQUE_ID` field (i.e. the second field in the list of field names). Remember, Python indexes values in a list beginning with the number zero (0).

Data Locks

Esri has adopted more Python standards with new software releases. One recent addition is the use of the Python “`with`” statement. Using the Data Access module and cursors the potential exists to cause a lock on data records which may prevent Python to subsequently execute processes on the same data. Implementing the `with` statement when defining and using a cursor helps prevent the locks on data without the need to write specific functions or code to ensure that the data locks are released.

Token Fields

Another concept that can be used with cursors is token (or placeholder) fields. Tokens are special fields that can be used with cursors to access and obtain specific geometry, location, area, length, and object ID fields and values and can be used in subsequent processes such as field calculations and passing information to various web map services. Tokens serve as placeholders for fields that can represent standard values that may be named or defined differently depending on the data type. For example, the object ID may be `OID`, `ObjectID`, `FID`, long Integer, etc. The **`OID@`** token name can be used to obtain the object ID of any data type. In a similar manner, the **`SHAPE@X`** and **`SHAPE@Y`** tokens can be used to obtain the X (longitude/Easting, etc) and Y (latitude/Northing, etc) from the `SHAPE` field. See the ArcGIS Help for a full list of token names/fields.

The rest of this chapter, the demonstration code, and exercises will follow this same logic and structures to illustrate the Insert and Update cursors.

Insert Cursor

The second kind of cursor used with databases is the *insert cursor*. An insert cursor allows a code developer to create new rows for a table or feature class and assign initial values to one or more of the attributes. A prerequisite for using the insert cursor is a table or feature class must exist that contains attributes (columns). The code below illustrates using the `CreateTable` ArcGIS routine to create a file geodatabase table from scratch and use the `InsertCursor` to add rows and populate some of the initial values for the attributes. See the ArcGIS Help for `CreateTable` and `AddField` routines. The **`insert_cursor_sample.py`** script can be found in the **`\PythonPrimer\Chapter06`** folder of the data that accompanies the text. The code has been formatted to fit the page. See the script for the proper formatting.

```
import arcpy, sys, traceback

# set the current workspace (in the case a file
# geodatabase)

arcpy.env.workspace =
'C:\\PythonPrimer\\Chapter06\\Data\\cursors.gdb'

outpath = 'C:\\PythonPrimer\\Chapter06\\Data\\cursors.gdb'

# assign a variable to point to the address table

address_table = 'insert_sample_address_table'

try:

    # check to see if the address table exists
    # if it does, delete it

    if arcpy.Exists(address_table):
        arcpy.Delete_management(address_table)

    # create a new table in the given location, in
    #this case a file geodatabase

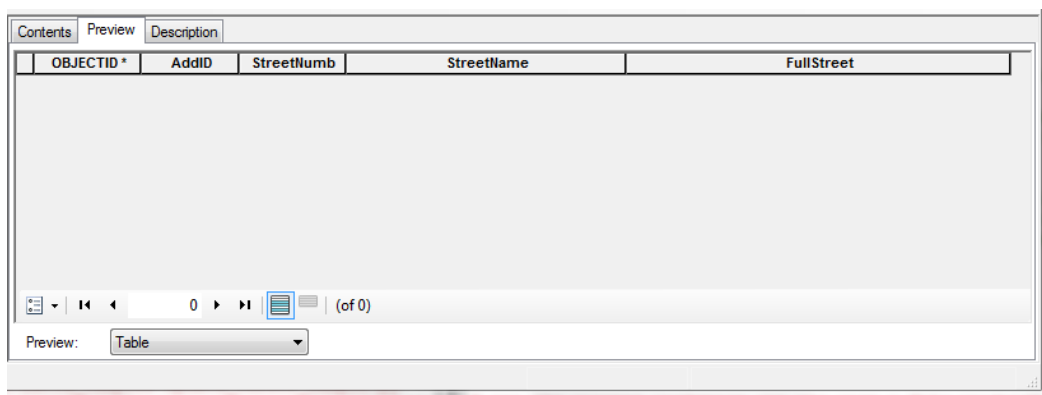
    arcpy.CreateTable_management(outpath,
        address_table)

    # add fields to the table
    arcpy.AddField_management(address_table, 'AddID',
        'LONG')
    arcpy.AddField_management(address_table,
        'StreetNumb', 'TEXT', '', '', 8)
    arcpy.AddField_management(address_table,
        'StreetName', 'TEXT', '', '', 60)
    arcpy.AddField_management(address_table,
        'FullStreet', 'TEXT', '', '', 70)

    print 'Created table and added fields'
```

The code above shows how to create a file geodatabase table from scratch. An `if` statement is used to check to see if a table already exists in the geodatabase. If it does, then it is deleted, since the next step actually creates the table. The `CreateTable` routine is used to create an empty table (**`insert_sample_address_table`**) and then the `AddField` routine is used to create new attributes. Notice the format of the `AddField` routine and the use of data types (e.g. Long and Text). The `AddField` routine requires the field name as well as the data type that will be stored in the field. The number for the text fields sets the width of the text field. See the ArcGIS Help for additional information on defining and formatting attribute fields.

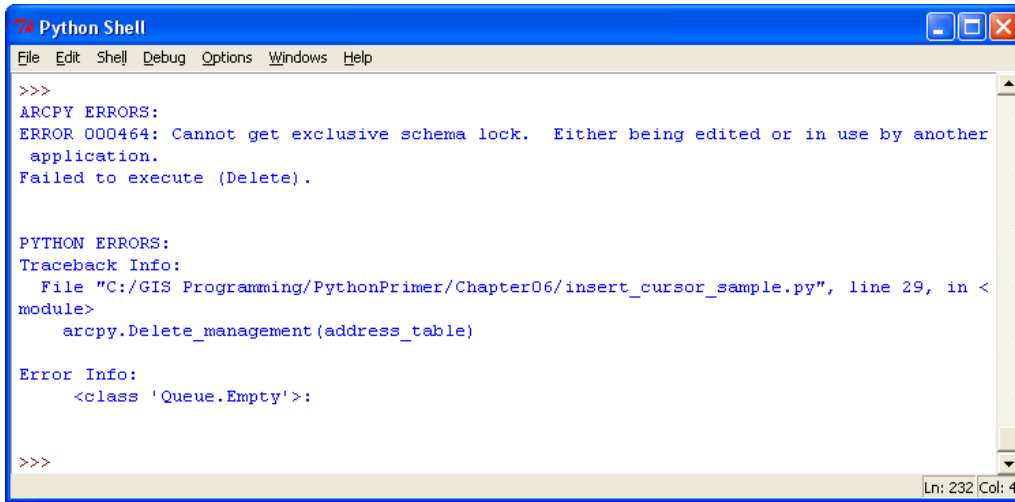
Running the script up to this point creates a new table in the file geodatabase called **`insert_sample_address_table`** with the added fields. If ArcCatalog is opened, the following appears when the user navigates to the location of the new table. If the reader is following along writing their own code, see the Schema Lock section and note below.



In the figure above under the ArcCatalog Preview tab for the table just created note that the attribute fields are present. In addition, the `OBJECTID` field is added. The `OBJECTID` field is automatically added by ArcGIS and must be present for GIS operations to occur within ArcGIS. Tables or feature classes that do not have this field may not be able to perform overlay operations, table joins, or create data from attributes that have Latitude/Longitude (or X/Y fields). At this point in the process no records (or rows) have been added. The insert cursor will be used to do this.

Schema Locks on Data

The reader may have already experienced challenges with deleting data when actively writing a Python script. The code developer may have seen a similar message shown below indicating a schema lock.



```
Python Shell
File Edit Shell Debug Options Windows Help

>>>
ARCPY ERRORS:
ERROR 000464: Cannot get exclusive schema lock. Either being edited or in use by another
application.
Failed to execute (Delete).

PYTHON ERRORS:
Traceback Info:
  File "C:/GIS Programming/PythonPrimer/Chapter06/insert_cursor_sample.py", line 29, in <
module>
    arcpy.Delete_management(address_table)

Error Info:
  <class 'Queue.Empty'>:

>>>
Ln: 232 Col: 4
```

In the case above a Python script is actively being developed and run from Python IDLE. To help troubleshoot syntax and processing problems, the code developer decided to open ArcCatalog to check the results of an operation. Subsequently, the code developer made additional changes in the open Python script, saved the edits, ran the Check Module, and attempted to re-run the script. Because both ArcCatalog and Python were open at the same time and referencing the same data, a lock (schema lock) was put on it. When the Python script attempted to delete the table, the error message was produced.

The remedy for such problem is to make sure all ArcGIS applications are closed completely. If Python is still open, the user can attempt to run the code again.

NOTE: In some cases, Python IDLE and the Python Shell must be completely closed in addition to any ArcGIS application because the Python program can access data. Oftentimes, this kind of problem occurs with cursors, looping functions, and attempting to update values that may or may not be available. The general rule to remedy this problem is to close all ArcGIS applications, Python Shell, and Python IDLE. Make sure to save changes to the Python script before closing. Most of the time a user does not need to shut down or restart a machine. Performing the above tasks should eliminate most schema locks.

Creating and Using the Insert Cursor

Now that a table is available, some records can be added to the table as well as initialize some of the values.

Before adding records (rows), the insert cursor must be defined. A variable `irows` is used to create and reference the insert cursor. The Python “with” statement helps to prevent data locks.

```
with arcpy.da.InsertCursor(address_table, fields_to_update) as irows:
```

`address_table` is a variable representing the input table created above. The input to an insert cursor can be a feature class or a table. The feature class or table must already contain field names and data types so that the insert cursor can use them.

`fields_to_update` is a variable that references a list (a *tuple*) of field names that can be provided to the cursor and used to set initial values to specific fields when the record is created. The code below shows the process of creating and using a cursor to assign a unique value to the `AddID` field. The code has been formatted to fit the page. See the **`insert_cursor_sample.py`** script mentioned above for the proper formatting.

```

print 'Created table and added fields'

# variable to store the attribute values to initialize
# i.e. (provide initial values for)

fields_to_update = ['AddID', 'StreetNumb',
                    'StreetName', 'FullStreet']

# Create insert cursor object

with arcpy.da.InsertCursor(address_table, fields_to_update) as
    irows:

    # counter variable to initialize loop
    counter = 1

    # run the loop until counter is <= to 10
    for irow in xrange(0,10):

        # create a new row and initialize the
        # AddID field with the value of counter

        irows.insertRow((counter, "", "", ""))

        # increment the counter so a new row can be
        # added
        counter += 1

    print 'Added ' + str(counter - 1) + ' rows to table: ' +
        address_table

except:

```

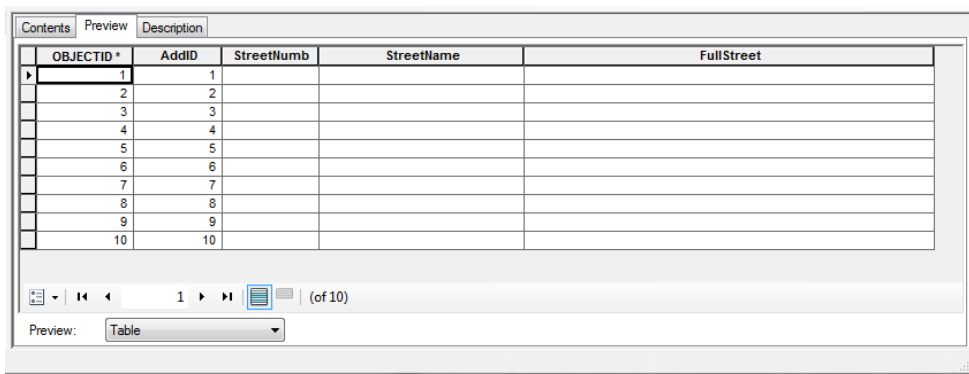
Once the insert cursor is defined, a `for` loop structure is used to insert rows in the new table.

In the example above, the code inserts ten new records to the empty table using the `irows.insertRow()` routine. For each new record (row) the `AddID` field (i.e. the first field in the list) is initialized (i.e. assigned) to the value of the `counter` variable. The other three fields (`StreetNumb`, `StreetName`, and `FullStreet`) are initialized with “blank” values by using a set of double quotes (`""`). The values in the parentheses are ordered in the list of fields provided in the `InsertCursor` line.

NOTE: This example shows a `for` loop to create new rows. A loop is not always required to add (insert) new rows to a table. For example, a search cursor can be set up to read rows from one table and the insert cursor can be created where a new row is inserted into a separate standalone table. In this case the new row and the insertion of the row are performed within the loop `for` the search cursor. From a code development point of view, it is up to the developer to determine what kind of looping structure is required and when to implement different functions. This is part of the challenge the code developer faces when writing scripts. Detailed workflows become very important in these instances.

Once the insert cursor has created new rows and populated any initial field values, the script produces the following results shown below. ArcCatalog can be used to see the table.

If the reader is writing the code while reading this section, the code can be saved and then run. Navigate to the location of the table and open it in ArcCatalog. Make sure to close Python IDLE and the Python Shell before doing so. Refer to the **insert_cursor_sample.py** script in the **\PythonPrimer\Chapter06** folder of the data that accompanies the text.



OBJECTID*	AddID	StreetNum	StreetName	FullStreet
1	1			
2	2			
3	3			
4	4			
5	5			
6	6			
7	7			
8	8			
9	9			
10	10			

Update Cursor

The *update cursor* can be used to update *existing* values in a table or feature class. The update cursor has the same parameters as the search cursor. The primary difference between the search and the update cursor is the search is a “read only” operation, whereas, the update cursor can be used to “write” or update values in a feature class or table. The update cursor also is different than a `CalculateField` operation because the calculate field operation updates all values (or those in a selection) with the same value. The update cursor can uniquely update a given row/column (i.e. cell) with a specific value by iterating through the rows in the cursor. Like the search cursor, the `where_clause` parameter in the update cursor represents a query string; the `sql_clause` parameter refers to key words or phrases that can serve as “prefix” or “postfix” strings when searching databases or geodatabases. This parameter is not used with shapefile or image data.

```
arcpy.da.UpdateCursor(<in_featureclass or in_table>, <field_names>,  
{where_clause}, {spatial_reference}, {explode_to_points},  
{sql_clause})
```

The following example illustrates how values from one table (attribute table) can update values in a separate table using both the search cursor and update cursor. This method is commonly used to read data from one data set and put it into another data set for use in other systems or organizations. This type of activity might be automated to run weekly to produce an updated table that another analyst, user, or external source can use. Refer to the **update_cursor_sample.py** script found in the **\PythonPrimer\Chapter06** folder. The code has been formatted to fit the page. See the script for the proper formatting.

The script begins with both a workspace and an output data path (`fgdpath`) defined.

```
import arcpy, sys, traceback

# set the current workspace (in the case a folder)

arcpy.env.workspace = 'C:\\PythonPrimer\\Chapter06\\Data\\'

fgdpath = 'C:\\PythonPrimer\\Chapter06\\Data\\cursors.gdb\\'

address_shp = 'addresses.shp'
address_fgd = fgdpath + 'update_sample_address_table'

try:

    # list (tuple) of fields to read in the search cursor
    # and written (updated) in the update cursor

    field_list = ["STREETNUMB", "STREETNAME"]
    # set a variable to point to the query string

    query = """"FID" >= 1 AND "FID" <= 10""""

    # Gets a collection of rows from a feature class or
    # table

    with arcpy.da.SearchCursor(address_shp, field_list, query) as
        srows:
```

Notice the path for the workspace. The workspace is a folder that contains the shapefile (**addresses.shp**) from which the search cursor will read data values from the address shapefile's attribute table. A separate data path variable (`fgdpath`) is defined as the file geodatabase location where the updated feature class will reside when the script is executed. This geodatabase contains a pre-existing table (**update_sample_address_table**) to which attribute values will be updated using the update cursor.

After the data path definitions, a query is created to limit the number of records pulled (i.e. accessed) from the **addresses.shp** shapefile. The query returns records with `FID >= 1` and `FID <= 10` from the addresses shapefile using the FID attribute. This happens to match the OBJECTID values for each record in the **update_sample_address_table** file geodatabase table. This example uses the FID and OBJECTID values to illustrate the use of the search and update cursor for reading values from the search cursor (**addresses.shp**) and updating similar rows in

the update cursor (**update_sample_address_table** file geodatabase table). In practice with a case like this, a different kind of query such as only return “residential” addresses vs. “business” addresses would likely be used.

The next step is to start the looping structure to cycle through and read each row in the search cursor.

```
# Gets a collection of rows from a feature class or table

with arcpy.da.SearchCursor(address_shp, field_list, query) as
    srows:

    obj_id = 1

    # read through each row of the search cursor

    for srow in srows:

        # assign streetnum to value of STREETNUMB
        # assign streetname to value of STREETNAME
        # assign fullstreet the concatenation of the two
        # values

        streetnum = srow[0]
        streetname = srow[1]

        # concatenate values for fullstreet
        fullstreet = str(srow[0]) + ' ' + str(srow[1])
```

The **STREETNUMB** (**srow[0]**) and **STREETNAME** (**srow[1]**) attribute values are read and stored in variables (**streetnum** and **streetname**). These two values are then concatenated and assigned to the **fullstreet** variable. Since the attribute values are referenced by a Python list index and are concatenated as a string, each index value requires “casting” to a string data type. Note the **obj_id** variable set to 1. This will be used in the next step.

Creating and Using the Update Cursor

Since both the *search cursor* and *update cursor* function row by row (or record by record), the code developer needs to devise a strategy to get the correct data from one table and put it in the correct location in the update table. In a sense, the rows in each table are synchronized. Working out the logic for this kind of task is challenging and may require the code developer to write out in detail the specific steps that need to occur for each dataset. Also, the code developer can expect some “trial and error” to occur to make sure the correct information is being read from and written to the right location in each data set. Some clues that indicate the logic is not right can include:

1. The same value being written out to each row
2. Only the first or last value being written out to each row
3. Empty values appearing where data is expected to be written to the output row values
4. Out of range errors, which indicate that the looping mechanisms are not iterating or incrementing properly
5. `Queue.Empty` error messages. These typically reference queries that do not have the correct information in it, improperly constructed query strings, or the `field_names` list values are not ordered correctly, or are missing field names that are directly referenced in the cursors
6. Queries that use incremented values (such as the use of `FID` and `obj_id` above) return values that do not exist or go beyond the physical range of the table (e.g. a table may only have 100 records, but the query indicates there are 101 records).

If errors occur, it is helpful to systematically work through each piece of code rather than try to work out the entire problem at once.

The following code shows how the search and update cursors can be used together to read values from one feature class and written to an output standalone table. After variables and data paths are defined, a search cursor is implemented to read a single row (record) from the address shapefile. Once this is completed the update cursor is implemented within the search cursor to write the values of the row read from the search cursor. Note the location of the

update cursor within the search cursor loop. The code has been formatted to fit the page. See the **update_cursor_sample.py** script file for the proper formatting.

```
import arcpy, sys, traceback

# set the current workspace (in the case a folder)
arcpy.env.workspace = 'C:\\PythonPrimer\\Chapter06\\Data\\'

fgdpath = 'C:\\PythonPrimer\\Chapter06\\Data\\cursors.gdb\\'

address_shp = 'addresses.shp'
address_fgd = fgdpath + 'update_sample_address_table'

try:

    # list (tuple) of fields to read in the search
    # cursor and written (updated) in the update
    # cursor

    field_list = ["STREETNUMB", "STREETNAME"]

    # set a variable to hold the query string

    query = """"FID" >= 1 AND "FID" <= 10"""

    # Gets a collection of rows from a feature class
    # or table

    with arcpy.da.SearchCursor(address_shp, field_list, query) as
        srows:

        obj_id = 1

        # read through each row of the search cursor

        for srow in srows:

            # assign streetnum to value of STREETNUMB
            # assign streetname to value of STREETNAME
            # assign fullstreet the concatenation of
            # the two values

            streetnum = srow[0]
            streetname = srow[1]
            fullstreet = str(srow[0]) + ' ' + str(srow[1])
```

```
update_fields = ["STREETNUM",
                 "STREETNAME", "FULLSTREET"]

# Get a collection of rows for the update
# cursor based on a query
# This update cursor will only return a
# single record, which is desired.

query = """"OBJECTID" = "" + str(obj_id)

# create the update cursor
with arcpy.da.UpdateCursor(address_fgd,
                           update_fields, query) as urows:

    # cycle through the rows (in this case
    # only 1) to actually update the row
    # in the cursor with the values
    # obtained from the search cursor

    for urow in urows:
        urow[0] = streetnum
        urow[1] = streetname
        urow[2] = fullstreet
        urows.updateRow(urow)

obj_id += 1

print 'Finished Updating Rows in ' + address_fgd

# delete the cursor variables to prevent # schema
# locks

del urow, urows, srow, srows

... # other code is processed here
```

The query statement just before the update cursors uses the OBJECTID attribute in the **update_sample_address_table** file geodatabase table (`address_fg`). Notice that the first record has an OBJECTID value of 1 and subsequent records are ordered sequentially.

The query is used as a parameter in the update cursor statement. Notice that the variable (`address_fg`) points to the file geodatabase table (**update_sample_address_table**) as the dataset for the update cursor.

A `for` loop is implemented to cycle through the rows (in this case a single row) of the update cursor. The program only needs to update a single record at a time, so the above structure provides this ability. Without accessing a single record, all of the update rows would continue to be overwritten with each iteration of the loop for `urows` and end up with the last value of the search cursor updating all of the records in the update table.

Within the `urows` loop the *StreetNum*, *StreetName*, and *FULLSTREET* fields in the **update_sample_address_table** table are updated with the values from the search cursor (**addresses.shp**). The reader might find it helpful to look back at the variables that are set from the search cursor. (Technically, the variables `streetnum` and `streetname` are not required, but are used to make the code a little easier to read and follow. The `srow[0]` and `srow[1]` could have been used to assign values to `urow[0]` and `urow[1]`, respectively). To actually set the updated values in the table for the given row, the following line is used.

```
urows.updateRow(urow)
```

To ensure that the next update row is queried properly, the `obj_id` variable is incremented by one. Notice also that the `obj_id` variable is not indented as part of the `urows` loop, but is indented as part of the `srows` loop. The code must update the `obj_id` variable before it is used again in the update cursor line. Since the code is only querying a single record, the code developer can actually increment the `obj_id` value within either the `urows` or the `srows` loop. Generally it is more appropriate to increment the `obj_id` as part of the `srows` loop because the initial process of the code is to iterate over each record of the search cursor and not update all of the rows at the same time for the update cursor. The table below shows the results of the *StreetNum*, *StreetName*, and *FULLSTREET* attributes after the update cursor is completed.

Contents Preview Description					
	OBJECTID *	AddID	StreetNum	StreetName	FULLSTREET
▶	1	1	912	11TH	912 11TH
	2	2	908	11TH	908 11TH
	3	3	1009	J	1009 J
	4	4	1030	I	1030 I
	5	5	921	10TH	921 10TH
	6	6	1013	J	1013 J
	7	7	1029	J	1029 J
	8	8	921	11TH	921 11TH
	9	9	1000	I	1000 I
	10	10	1000	J	1000 J

1 (of 10)

Preview: Table

As shown above, the cursors are deleted to free up space and to assist with removing potential data locks.

This section illustrated the common implementations of the search, insert, and update cursors. These are valuable structures for being able to iteratively and independently read through, create, and update records in database tables, feature classes, and individual data file formats supported by ArcGIS. Being able to effectively use cursors can assist the code builder in developing automated processes where data needs to be extracted from one database and put into another. Many of these types of data commonly require tens, to hundreds, to millions of records. These types of data and processes are good candidates for automation and can run during off hours (at night or non-peak hours). The examples and exercises provided have been building on the concept of process automation and will continue to be developed in the subsequent chapters. *Workbook III*, Chapter 11 focuses on script automation.

Table Joins

A common need for many GIS users is bringing together data from a variety of sources. In some cases the different datasets are maintained separately from one another. For example a biologist may collect and maintain spatial data and some attribute characteristics for certain species of animals. Part of the information the biologist may need is data related to water temperature of nearby lakes which is maintained separately by a water agency. The water agency has an identification field in the lakes table called *LakeID*. When the biologist created the species feature class (e.g. a point feature class of species) one of the attributes she added was the *LakeID*. Since the biologist is not the maintainer of the water data, there is no real need to maintain lake related information as part of the animal species data; however, the biologist is interested in the lake data because it contains useful information for the biological

analysis the biologist performs. The two datasets can be related to one another through a “dynamic relationship” by “virtually connecting” them via a database mechanism called a *table join*. Table joins allow different datasets that are separately maintained to connect to one another so that values from the respective tables can be used in attribute queries, reading values from the respective tables, updating data, or summarizing information from the “joined” tables.

The reader can follow along in this chapter by reviewing the **Lakes** feature class and **Lake_Info** table in the **cursors.gdb** file geodatabase provided in the **Chapter06\Data** folder. The **table_joins_sample.py** script can be found in the **Chapter06** folder. Excerpts are shown throughout this section.

Two forms of these “virtual relationships” exist in ArcGIS, 1) table joins and 2) table relates. Both types of relationships require a common attribute to exist in both datasets. A table join is most often used when a one-to-one relationship exists between the two data sets. A table relate provides the ability to perform more complex relationships between data sets where a one-to-many or many-to-many relationship exists. For example, a parcel may have many addresses within it (one parcel, many addresses) or an apartment complex may have multiple buildings that cross parcel lines and also have many addresses per building (many buildings to many parcels).

This chapter will focus only on table joins since the fundamentals are similar for both types of relationships. The reader can find more information on joins and relates in the ArcGIS Help. Readers are recommended to review all of the ArcGIS Help topics under Tables if they are unfamiliar with the different kinds of table operations available in ArcGIS. The ArcGIS Help covers tables in general, table design, joining, relating, as well as a special type of join, the “spatial join.” Specific information about the Spatial Join routine can be found in the ArcGIS Help under **Analysis—Spatial Join** since Spatial Join is a geoprocessing tool found within the Analysis toolbox and is often used in Python scripts when multiple spatial overlays (i.e. combining multiple data sets with overlapping geographies) are required. For example each address in an address point feature class can be assigned the parcel number when the addresses fall within the parcel polygon feature class.

Before developing code or determining if a join function should be used the analyst should review each data source to determine what the attributes are and if there are any common attributes between the two that might be used to join the two data sets together. The following shows the attribute tables from the **Lakes** feature class and the **Lake_Info** table from the **Chapter06\Data\cursors.gdb** file geodatabase.

OBJECTID	Shape	AREA	PERIMETER	HYDRO24CA	HYDRO24CA1	TYPE	HNAME	Shap
3	Polygon	336207	2556.76	4348	4	L	BASS LAKE	8
5	Polygon	12105.6	470.1	4365	66	L	BASS LAKE	1
1	Polygon	45009600	141633	4064	74	L	FOLSOM LAKE	4E
2	Polygon	41307	1003.27	4280	5	R	HINKLE RESERVOIR	3
4	Polygon	1960410	21568.8	4352	109	L	LAKE NATOMA	7C
6	Polygon	35517.3	896.588	4370	30	R	RESERVOIR	
7	Polygon	33562.4	730.349	4406	8	R	WILLOW HILL RESERVOIR	2

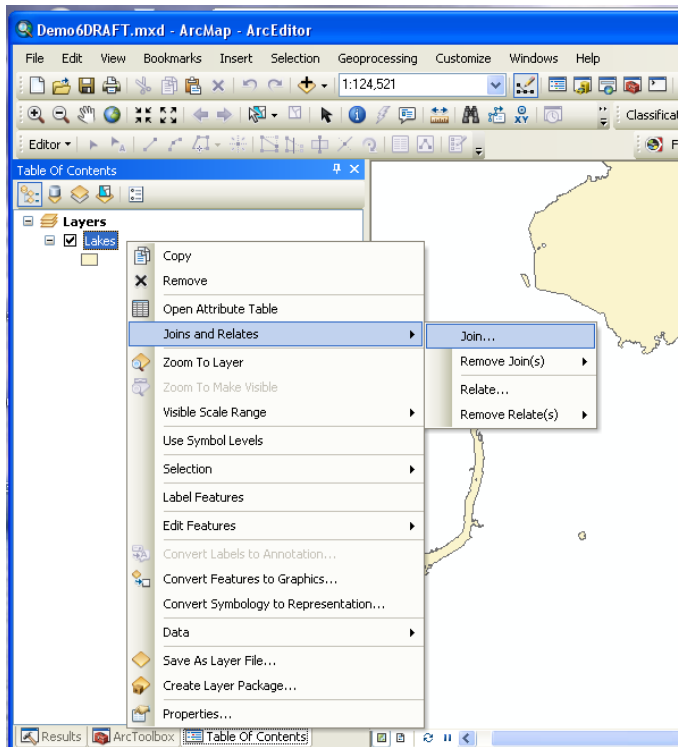
Lakes feature class attribute table.

OBJECTID	LakeID	Lake_Name	Temp_F
1	74	Folsom Lake	65
2	109	Lake Natoma	72
3	5	Hinkle Reservoir	73
4	66	Bass Lake	63
5	8	Willow Hill Reservoir	75

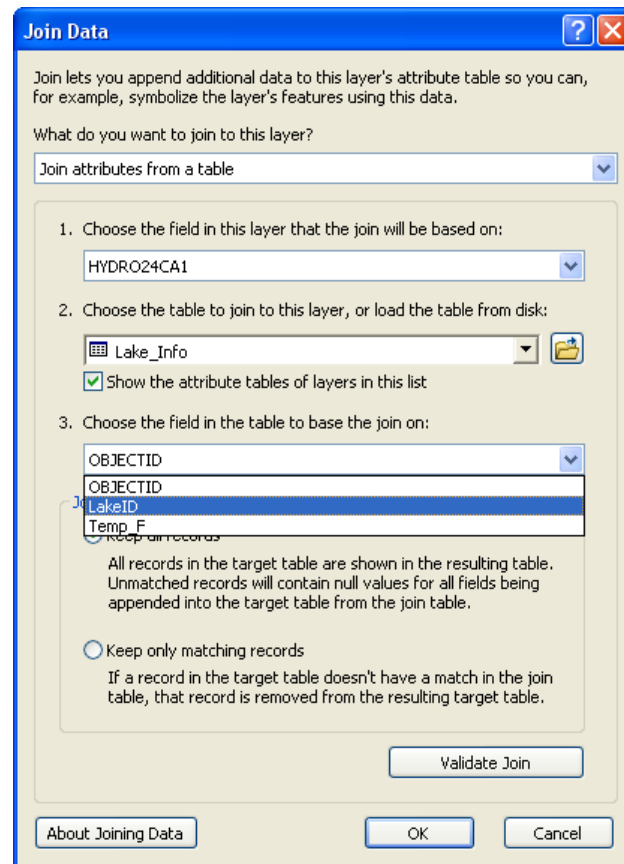
Lake_Info attribute table.

Upon reviewing the two data sets one sees that a lake temperature value only exists in the **Lake_Info** attribute table and not in the **Lakes** feature class. Also, the two attribute tables have different numbers of records (the **Lakes** feature class has seven records; the **Lake_Info** has five records). One will also notice that Bass Lake is represented by two separate features in the **Lakes** feature class; whereas, the **Lake_Info** table only contains a single temperature value for that feature. In addition, the analyst sees that a common field exists in both attribute tables, although the name of that field is different. In the **Lakes** feature class the field is called *HYDRO24CA1* whereas, in the **Lake_Info** table, the attribute with the same values is called *LakeID*. These two attributes also have the same data type (a numeric value). These respective attributes will be used to perform a table join.

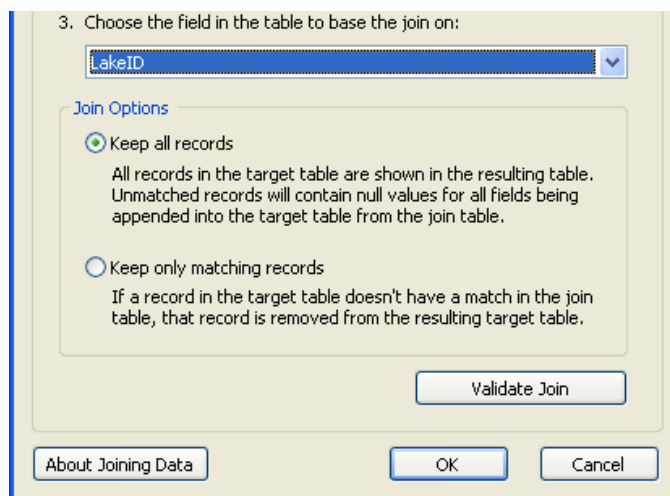
In ArcMap, a table join can easily be performed by right clicking on the feature class (actually a feature layer) or table (actually a table view) in the Table of Contents and then selecting **Joins and Relates—Join**. Remember when feature classes or tables are added from disk to ArcMap, they become feature layers or table views, respectively. The following pages review the process to perform a join manually within ArcMap. Later, the Python scripting methods will be reviewed that accomplishes the same task.



A dialog box then appears that allows the user to select the matching attribute fields from each attribute table to perform the join. The Join dialog box below shows the *HYDRO24CA1* attribute from the **Lakes** feature class and the *LakeID* from the **Lake_Info** table as the respective “join attributes” from each data set.

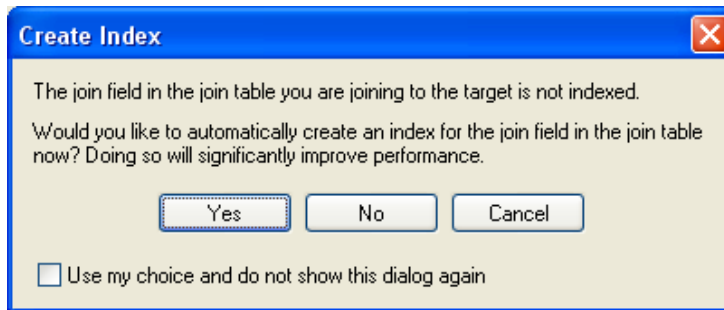


Notice that “Keep all records” is checked. This is the default. However, some users may not want to see the unmatched records during a join because they are most likely irrelevant.



Attribute Indexes

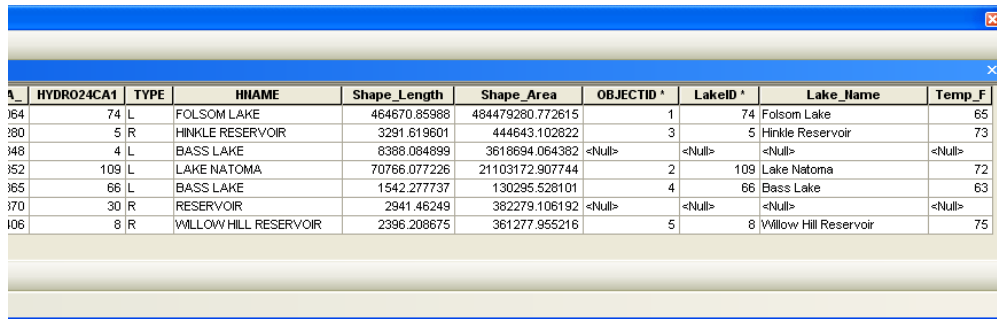
When the user clicks OK in the above dialog box, a message appears asking if the user wants to create an attribute index. An index is a database construct that allows quick access to data values. One can think of an attribute index as an index in a book. Key topics in a book index are sorted in alphabetical order. A reader can consult the index, find the topic and locate the page number the topic is on, turn to the page in the book and begin reading about that topic. In the same manner, an attribute index creates a sorted set of values on one or more attributes and usually assigns a number to the sorted list. The index is sometimes referred to as a look up table. The user never “sees” the look up table, but it essentially contains a number and the sorted attribute value. When a “join” is performed, the index serves as a more efficient way to “connect” values from different tables.



Recommendation: It is highly recommended that attribute indexes are created and used when “joining” one or more tables, especially those with large numbers of records. Indexes will need to be created for each attribute that an analyst expects to use when performing table joins and queries that involve joined tables.

For example, a city can have a master address database with hundreds of thousands of addresses. These addresses often relate to parcel or street centerline data of which each of these datasets can have hundreds of thousands of records. Attempting to maintain updates to each of these datasets and relationships between them can be very time consuming if attribute indexes are not used in the process.

In this example, after the user clicks “Yes” from above, the attribute tables from the different datasets are “joined.” The user can open the attribute table for the **Lakes** feature class to see the “connected” attribute tables. The image below shows portions of the joined data sets.



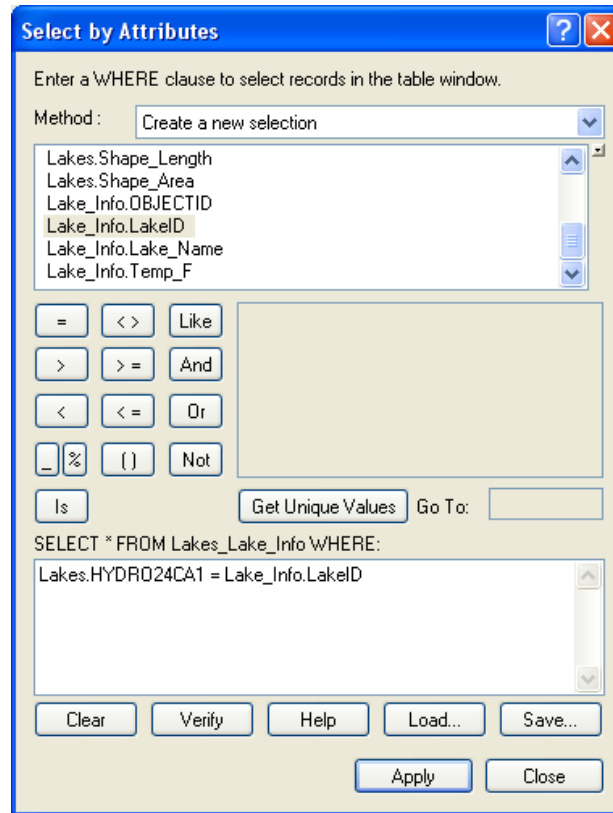
	HYDRO24CA1	TYPE	HNAME	Shape_Length	Shape_Area	OBJECTID *	LakeID *	Lake_Name	Temp_F
164	74	L	FOLSOM LAKE	484670.85988	484479280.772615	1	74	Folsom Lake	65
180	5	R	HINKLE RESERVOIR	3291.619601	444643.102822	3	5	Hinkle Reservoir	73
148	4	L	BASS LAKE	8388.084899	3618694.064382	<Null>	<Null>	<Null>	<Null>
152	109	L	LAKE NATOMA	70766.077226	21103172.907744	2	109	Lake Natoma	72
165	66	L	BASS LAKE	1542.277737	130295.528101	4	66	Bass Lake	63
170	30	R	RESERVOIR	2941.46249	382279.106192	<Null>	<Null>	<Null>	<Null>
106	8	R	WILLOW HILL RESERVOIR	2396.208675	361277.955216	5	8	Willow Hill Reservoir	75

Joined attribute tables. *OBJECTID* and *LakeID* represent the beginning of the attributes from the *Lake_Info* table. All of the attributes to the left represent data fields from the *Lakes* feature class. The reader may want to do the join in ArcMap to actually see the results.

Notice that a record in the **Lake_Info** table matches records in the **Lakes** feature class where *HYDRO24CA1* and *LakeID* are the same. Note the <null> values in the *LakeID* attributes. The <null> indicates that no match was found. These show up in the resulting joined table because the user chose to show all records in the join options versus only matching records.

Notice that the *OBJECTID* and the *LakeID* attributes each have an asterisk (*) next to the attribute name. The asterisk indicates that an index exists for this attribute. The *OBJECTID* in all tables and feature classes are created automatically by ArcGIS.

At this point the user can create attribute queries, such as with the Select Layer By Attribute routine, using the fields from the **Lakes** feature class and the joined attribute table from the **Lake_Info** table. For example the attribute query shown below “selects” records where the Lake *HYDRO24CA1* attribute is equal to the **Lake_Info** *LakeID*.



The analyst should notice that with “joined” attributes, the query looks a little different from non-joined tables. Joined table queries must include the feature class or table name in addition to the attribute. The above query statement looks like this.

```
<table or feature class>.<field_name> =
    <joined table or feature class>.<field_name>
```

```
Lakes.HYDRO24CA1 = Lake_Info.LakeID
```

From a programming point of view, this is important to know because besides joining tables, often a subsequent step is to query the data to select features or rows or to use a query in a cursor routine. The query syntax may be a little more challenging with joined data sets and often require troubleshooting by the code developer.

Programming and Using Table Joins

This section describes how to code the manual steps shown above and then use the join in a subsequent process. Refer to the **Chapter06\table_joins_sample.py** script as needed.

To use joined tables in Python scripts, three elements need to be created:

1. Attribute Index – (not required, but good for large datasets)
2. Feature Layer or Table View
3. Table Join

The attribute index is created to efficiently perform the table join. The index is not required, but is beneficial with large datasets. The *feature layer* or *table view* is required for the table join to occur and so it can be used in feature selections by attributes or location or in cursors. Table joins cannot use “feature classes” or “tables.” Feature layers and table views can be created at any time before the `AddJoin` routine is used to create the table join. The author tends to create the indexes first and then then make the feature layer or table view. See the ArcGIS Help for **Add Join**. The table join needs to be identified so the script and other processes can use and perform a join.

Creating an Attribute Index

The first step to creating and using a table join is to set up an attribute index on the attribute that is expected to be used in the join. The `AddIndex` routine is used to create the attribute index. See the ArcGIS Help on **Add Attribute Index** for more details.

The required parameters for the Add Attribute Index are:

1. Table name
2. Field(s) participating in the index
3. Index name (optional, but may be helpful in scripting routines)

In the example above, the add index syntax may look like this. The code snippet has been formatted to fit the page.

```
# create indexes for the feature classes and tables
arcpy.AddIndex_management(lake_fc, 'HYDRO24CA1',
    'Hydro_Index', 'NON_UNIQUE', 'NON_ASCENDING')

arcpy.AddIndex_management(lake_table, 'LakeID',
    'Lake_Index', 'NON_UNIQUE', 'NON_ASCENDING')
```

Two indexes are shown above, one for a feature class (*Lakes*), the other for a table (*Lake_Info*). The first parameter is the feature class or table name (shown as a variable in the code). The second parameter represents the attribute field the index will be created on. The third parameter is a name of the index which is any name the developer chooses. The last two parameters indicate if the index will represent unique values and if the index will be ascending or descending. Most of the time, the parameters show above can be used. Indexes are created on *feature classes* and *tables*, not *feature layers* and *table views*.

Creating Feature Layers or Table Views

Feature layers and table views can be created as described in Chapter 5 and need to be created before the `AddJoin` routine can be implemented. The code below shows how a feature layer and table view are created before being used in the `AddJoin` routine. Note a check is created to see if the feature layer or table view exists (and then is deleted it if it does) so that the script can be run multiple times if needed.

```
# Create a feature layer and table view
# so the AddJoin routine can run

if arcpy.Exists(lake_fl):
    arcpy.Delete_management(lake_fl)

if arcpy.Exists(lake_tv):
    arcpy.Delete_management(lake_tv)

# Make Feature Layer
arcpy.MakeFeatureLayer_management(lake_fc, lake_fl)

# Make Table View
arcpy.MakeTableView_management(lake_table, lake_tv)

# Create the Join
arcpy.AddJoin_management(lake_fl, "HYDRO24CA1",
    lake_tv, "LakeID", "KEEP_ALL")
```

Creating the Table Join

Once these elements are created, they can be used appropriately in the `AddJoin` routine. In addition to the feature layer and table view, the other required parameters for the `AddJoin` are the respective “join fields” that is used to join the two data sets together. In the example, the **Lakes** feature layer uses the *HYDRO24CA1* field and the **Lake_Info** table uses the *LakeID* field. Note that the `AddJoin` routine uses the “KEEP_ALL” value to keep all of the records from the input feature layer, even if there is no match in the **Lake_Info** table. Unmatched records are not “deleted” or “eliminated” since a join is performed in memory and does not physically change the data.

Using and Accessing Information in Joined Data

Once the join is complete, the values from both attribute tables can be used for subsequent processing. The code snippet below shows the `ListFields` routine to print out a list of fields to the Python Shell as well as set up a search cursor to read specific elements from the joined data. ArcGIS list routines will be discussed in the next chapter. Notice the `field_list` variable is assigned to a Python list of “joined” field names. Since variables have been previously defined in other parts of a script (not shown) for the feature class (`lake_fc`) and table (`lake_table`) names, they can be used in the Python list of fields as well as in the query parameter for the search cursor. The **table_joins_sample.py** script and **Demo6d** provides more coding details. The code has been formatted to fit the page.

```

# Create the Join
arcpy.AddJoin_management(lake_fl, "HYDRO24CA1", lake_tv, "LakeID",
    "KEEP_ALL")

# print out a list of fields so the code developer
# can get an idea of what the field names look like

fields = arcpy.ListFields(lake_fl)

for field in fields:
    print field.name

# list (tuple) of fields to read in the search cursor

# the list of fields that include the join
# featureclass/table names

field_list = [lake_fc + '.HYDRO24CA1', \
    lake_fc + '.HNAME', \
    lake_table + '.LakeID', \
    lake_table + '.Temp_F']

# Create a query to only process the rows where
# HYDRO24CA1 and LakeID are the same

query = lake_fc + '.HYDRO24CA1 = ' + lake_table +
    '.LakeID'

# Create a cursor to read the values from the joined
# table Notice the input is the feature layer (used in
# the AddJoin above)

with arcpy.da.SearchCursor(lake_fl, field_list, query)
    as srows:

    for srow in srows:

        # Read values from the joined table

```

The `ListFields` routine may be useful for a code developer to see a list of fields, since joined fields look different and require some syntax changes when developing queries (such as selecting data or creating cursors). The program continues to read values from both the feature layer and the table view as shown below. These are printed to the Python Shell, but they could easily have been written to a separate table or used to update values in another feature class, table or in other subsequent processes within the code.

```

with arcpy.da.SearchCursor(lake_fl, field_list, query) as srows:

    for srow in srows:
        # Read values from the joined table
        # Note the feature class name and the table name
        # are used

        HydroID = srow[0] # lake_fc + '.HYDRO24CA1'
        LakeName = srow[1] # lake_fc + '.HNAME'
        LakeID = srow[2] # lake_table + '.LakeID'
        temp_F = srow[3] # lake_table + '.Temp_F'

        # print the values out to the Python Shell
        # these values can be used in update cursors or
        # populating rows in other tables or feature
        # classes

        print lake_fc + ' HYDRO24CA1 is: ' + str(HydroID)
        print lake_fc + ' Lake Name is: ' + LakeName
        print lake_table + ' LakeID is: ' + str(LakeID)
        print lake_table + ' Temperature is: ' +
            str(temp_F) + '\n'

    # remove the join, since it is no longer used
    arcpy.RemoveJoin_management(lake_fl, lake_table)

    # delete cursors to free up memory and help eliminate
    # data locks
    del srow, srows

```

The search cursor uses the `field_list` index position (`srow[0]`, `srow[1]`, `srow[2]`, etc) to obtain the value for the specific row and column.

Note: For ArcGIS 10.0, the syntax to obtain a value from a joined field is

```
aValue = srow.getValue(<field_name>)
```

where the `<field name>` syntax is

```
<feature_class or table_name>.<field_name>
```

or `lake_fc + '.HYDRO24CA1'` using the example above and

```
aValue = srow.getValue(lake_fc + '.HYDRO24CA1')
```

The ArcGIS 10.0 version of cursors and code can be found under the **Chapter06\legacy** folder in the data package that can be downloaded from the author's website.

Summary

This chapter has shown a variety of cursor methods and the table join that can be used by code developers to access specific values from feature classes or tables (even joined tables) and use those values for other purposes. These operations can be very powerful and useful when working with large databases, performing frequent updates, and summarizing or combining data from different databases or systems. Often these structures are found in automated scripts that run during off-hours or off-peak computer system loads. Properly constructing both the cursor and the processing logic to successfully process the records are key concepts the reader should take away from this chapter. In many cases, designing the proper processing logic will continue to be a primary challenge when using cursors. Once this is overcome, a Python script can become very useful to an organization managing large datasets and databases.

ArcGIS/Python Help References

- Data Access module
- Cursors (Search, Insert, Update)
- Python lists and tuples
- Using Tokens
- CreateTable
- CopyRows
- CopyFeatures
- AddField
- Joins (Add, Remove)
- Indexes
- Feature Layers and Table Views

Chapter 6 Demos

The demos for this chapter are broken into four parts:

1. Demo 6a – Search Cursor
2. Demo 6b – Insert Cursor
3. Demo 6c - Search and Update Cursor
4. Demo 6d – Table Joins

The demos follow the procedures mentioned above and use the ArcGIS 10.1 and later Data Access cursors. The data for these demos can be found in **\PythonPrimer\Chapter06\Data**. The completed **Demo6** scripts can be found in the **\Chapter06** folder. The reader may find it helpful to view the feature classes and tables in ArcCatalog and can also review data in ArcMap before starting the demos. The ArcGIS 10.0 versions can be found in the **Chapter06\legacy** folder.

The concepts illustrated in these demos are:

ArcGIS Concepts

Search Cursor
Insert Cursor
Update Cursor
Create Tables
Add/Remove Join
Add Fields
Data paths
Queries
Data locks
Data Access module

Python Concepts

`for` loop
`while` loop
Indentation
Casting numbers to strings
`os` module

Demo 6a: Search Cursor

This demo uses the **Sacramento_Streets.shp** file that can be found in the **\PythonPrimer\Chapter06\Data** folder. Open ArcMap or ArcCatalog to see the various attributes and values of **Sacramento_Streets.shp** file as needed.

NOTE: The reader's path may be different than shown below and depending on where the data is put.

This demonstration illustrates the construction and use of a search cursor to read values from an attribute table using a query to read only the highway records.

1. Add the `arcpy`, `sys`, `os`, and `traceback` modules. Recycle the `except :` block from a previous script. Add some commentary describing the script.
2. Set a workspace to the **\Chapter06\Data** folder shown above.
3. Create a variable to point to the name of the shapefile

At this point, the script should look similar to this.

```
import arcpy, sys, os, traceback

# set the current workspace (in the case a folder)
arcpy.env.workspace = 'C:\\PythonPrimer\\Chapter06\\Data\\'

# assign a variable to point to the street shapefile
streets_shp = 'Sacramento_Streets.shp'

try:
```

Since only highways will be used in this cursor, a query variable is set with the following query:

4. Create a query variable to select only highways (i.e. "CLASS" = 'H'). Notice the "triple double quotes"

```
query = """CLASS" = 'H'"""
```

5. To limit the cursor to only the FULLSTREET and UNIQUE_ID fields, create a variable to point to a Python list of these fields:

```
field_list = ["FULLSTREET", "UNIQUE_ID"]
```

6. Create the search cursor using the `streets_shp` variable, the list of fields, and the query variable.

```
# Get a collection of rows from a feature class or
# table

with arcpy.da.SearchCursor(streets_shp, field_list, query) as
    srows:
```

Now that the cursor is defined, a looping structure can be created to iterate over each of the records in the cursor. Make sure to indent the `for` loop one tab.

7. Create a `for` loop to iterate over the rows of the cursor

Since a search cursor simply reads values from a table, create two new variables to hold the values of the FULLSTREET and UNIQUE_ID fields. The Data Access cursors use the field list index position to access a specific column. `srow[0]` references the first value in the `field_list` list of values which is FULLSTREET and `srow[1]` to reference the second value in the `field_list`, UNIQUE_ID). Python references elements in a Python list using index values and begin with zero to reference the first element in the list. Shown below is a way that values for a specific row (`srow`) and column (`[0]` or `[1]`) can be obtained and assigned to a variable which can then be used in a subsequent process (in this case, printing the values).

```
for srow in srows:    # for each row in the cursor

    # assign a variable for the value for FULLSTREET
    # assign a variable for the value for UNIQUE_ID

    fullstreet = srow[0] # FULLSTREET
    unique_id = srow[1]  # UNIQUE_ID

    # prints the value of the variables
    # i.e. the value of srow.[0] and srow.[1]

    print fullstreet + ' ' + str(unique_id)
```

For this demo, the code developer simply prints the values to the Python Shell.

8. Create a `print` statement that prints the full street name and the `unique_id` value. Remember that when printing numbers with a print string, that the `str()` must contain the value. See above.
9. Make sure to include the `except:` block of code mentioned above. The code can be found in the **Demo6a.py** script or in the **Chapter01** folder of the data package. Save and check the module. Fix any problems encountered. See the **Demo6a.py** script, if needed.
10. Save and run the script. The Python Shell should display the values read from the **Sacramento_Streets.shp** file. Only highway FULLSTREET names and their respective UNIQUE_ID values should appear in the printed output. A sample of the output is shown below.

```
BUS 80 EB 76824
I 5 NB 77245
I 5 SB 77246
>>>
```

Demo 6b: Insert Cursor

This demonstration will illustrate the use of the insert cursor. In addition, the user will learn how to create a table from scratch and add some fields. The insert cursor will be used to create new rows and populate initial values to one of the attributes. It is recommended that write access is available to the folder where the file geodatabase is located.

This demo uses a pre-existing file geodatabase (**cursors.gdb**) to create the **Demo6b_insert_address_table**. The geodatabase can be found in the **\PythonPrimer\Chapter06\Data** folder. Refer to the **Demo6b.py** script as needed.

NOTE: The reader's path may be different than shown below depending on where the data exists.

1. Start the script by adding the `arcpy`, `sys`, `os`, and `traceback`, modules. The `os` module is not required unless the `os.path.join` routine is used (see below). Add some commentary describing the script. Recycle the `except:` block from a previous script.
2. Add a workspace (`arcpy.env.workspace`) and a workspace variable (`outpath`) to point to the file geodatabase. Both of these will point to the full path to the **cursors.gdb** file geodatabase. Note: a workspace to a file geodatabase includes the folder plus the name of the file geodatabase including the extension (`.gdb`).
3. Add a variable (`address_table`) to point to the name of the new table (yet to be created) called **Demo6b_insert_address_table**. There is no file extension on the table, since it resides within the file geodatabase. Tables and feature classes within file, personal, or SDE geodatabases do not have extensions.

The code should look similar to the following:

```
import arcpy, sys, os, traceback

# set the current workspace (in the case a file geodatabase)

arcpy.env.workspace =
'C:\\PythonPrimer\\Chapter06\\Data\\cursors.gdb'

outpath = 'C:\\PythonPrimer\\Chapter06\\Data\\cursors.gdb'

# assign a variable to point to the address table
address_table = 'Demo6b_insert_address_table'

try:
```

Since this demo illustrates creating a new table and adding attributes, the `CreateTable` function must be used.

4. Add the `CreateTable` routine to the script.

```
try:

    # check to see if the address table exists
    # if it does, delete it

    if arcpy.Exists(address_table):
        arcpy.Delete_management(address_table)

    # create a new table in the given location, in this
    # case a file geodatabase

    print outpath
    print address_table

    arcpy.CreateTable_management(outpath, address_table)
```

5. The `CreateTable` routine requires an output path and a table name. Use the `output_path` (which is the same as the workspace, in this case) and the `address_table` variable for the table name. Use these two variables in the `CreateTable` routine.

In this script the workspace and the location of the new table point to the same location. Oftentimes the output location is different from the input location. Using a separate variable to represent the location of the output would be a good option.

Since the script may be run multiple times, it is good practice that if the developer knows a table or feature class will be replaced with a new feature class with the same name, the use of the `Exists` function can be used to test to see if the table already exists. If it does, the `Delete` routine can be used to delete the table. If it is not deleted, the code will break, since it will be trying to create a table with the same name of the table that already exists.

6. Add the `Exists` and `Delete` statements above. Notice that an `if` statement is used to check to see if the table exists.

Now that an empty table has been created, some attributes (fields) are required before rows are added. The `AddField` ArcGIS routine is used to perform this operation. Consult the ArcGIS Help for `AddField` to find out more details on using different data types and how to write the syntax. Most of the time in ArcGIS, the fields will be strings with a specified length, integers (long or short), or doubles (floating point numbers) values.

7. Use the `AddField` routine to add the following fields with the specified data type. Each added field uses the same general syntax.

AddID - Long

StreetNum – Text, length 8 characters

StreetName – Text, length 70 characters...

```
arcpy.CreateTable_management(outpath, address_table)

# add fields to the table
arcpy.AddField_management(address_table, 'AddID', 'LONG')
arcpy.AddField_management(address_table, 'StreetNum', 'TEXT',
    '', '', 8)
arcpy.AddField_management(address_table, 'StreetName', 'TEXT',
    '', '', 70)
...
```

Note the format of the parameters for the `AddField` routine. Some parameters have (") to represent place holders for optional values that are not needed so that the order of the parameters remain the same.

NOTE: If the workspace for the input data and the output path location for the output data are different, the following syntax is one way to modify the code for the `Exists`, `Delete`, and `AddField` routines. Remember to import the `os` module so the `os.path.join` routine can be used to combine the `outpath` and the `address_table` into one string that will be used in the `AddField` routine. Also see that the `CreateTable` routine did not change because the required parameters include a “folder or workspace name” and the “name of the table” as separate parameters.

```

# input data location
arcpy.env.workspace =
    'C:\\PythonPrimer\\Chapter06\\Data\\MyInput.gdb'

# output data location (different from input)
outpath = 'C:\\PythonPrimer\\Chapter06\\Data\\MyOutput.gdb'

address_table = 'Demob6b_insert_address_table'

# variable to combine the output path and table name
# for use in the Exists, Delete, and AddField routines

outpath_table = os.path.join(outpath, address_table)

if arcpy.Exists(outpath_table):

    arcpy.Delete_management(outpath_table)

# create a new table in the given location, in this
# case a file geodatabase

arcpy.CreateTable_management(outpath, address_table)

# add fields to the table
arcpy.AddField_management(outpath_table, 'AddID', 'LONG')

```

Now that a blank table has been created and has some attribute fields, the insert cursor can be implemented. If a pre-existing table contained attribute fields, the above steps are not needed.

8. Create a variable that contains a Python list of fields that will be used in the insert cursor.

```

# create a list of fields to update when using the insert cursor
field_list = ['AddID', 'StreetNum', 'StreetName']

```

9. Add the insert cursor routine which requires the table name and the list of fields as parameters.

```

# Create cursor object
with arcpy.da.InsertCursor(address_table, field_list) as irows:

```

Once the insert cursor is created, a looping structure is required to insert the respective number of rows and provide initial values to attributes, if needed. A `for` loop or a `while` loop is suitable for this process. In this example, the `while` loop is used. A `counter` variable can be used to initialize the loop and establish a certain number of records (rows) in the table.

10. Add the `while` looping structure and `counter` variable. Make sure to indent below the “with” statement when creating the `while` loop. See **Demo6b.py** for the proper formatting.

```
# set counter variable to initialize loop
counter = 1

# process the loop until counter is <= to 10
while counter <= 10:

    # actually inserts the row into the table
    irows.insertRow((counter, "", ""))

    # increment the counter to iterate the while
    # loop

    counter += 1
```

Within the loop the actual new rows are created and initialized. The `insertRow` routine is passed the values for each field in the `field_list`. The inner set of parentheses is required syntax when passing values for a list of fields. In the example above, the value of `counter` will set the initial value of the `AddID` field. Every time the loop iterates, `counter` is increased by 1 and hence so does the value of `AddID`. The other two attribute fields (`StreetNum` and `StreetName`) are set with no value (i.e. `""`).

The final part of the **Demo6b** script includes the `del` (delete) Python function to remove the cursor variable from memory which can also help to eliminate data locking problems. A `print` statement is also added to indicate the number of inserted rows.

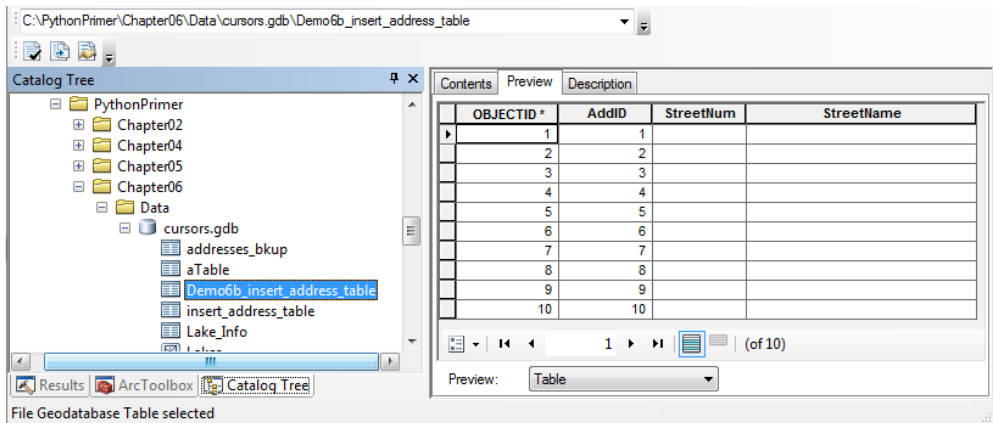
```
# delete the cursor variable to prevent schema locks

del irows

print 'Added ' + str(counter - 1) + ' rows to table: ' +
address_table
```

When the script is completed, 10 rows are added to the address table which was created from scratch. Notice how the `while` loop is structured and the primary components of creating and initializing rows are used in addition to incrementing the `counter` variable by one each time the `while` loop executes. Note also that the `print` statement uses “`counter - 1`” instead of only `counter`. The reason for this is that the `counter` variable is used in the `while` statement to check to see if `counter` is greater than 10. If it is, then the loop stops. At the last iteration of the `while` loop, `counter` is 11. To report out the correct number of rows added, `counter` must have 1 subtracted from it when used in the `print` statement.

The resulting table should look like the following illustration after the insert cursor script is completed. Use ArcCatalog to view the resulting table. Make sure to close the Python IDLE and Shell windows before looking at the table in ArcCatalog or use **View—Refresh** (or F5) to refresh the **Demo6b_insert_address_table** within the **cursors.gdb** file geodatabase.



The screenshot shows the ArcCatalog interface. The Catalog Tree on the left displays the folder structure: PythonPrimer > Chapter06 > Data > cursors.gdb > Demo6b_insert_address_table. The main pane shows a preview of the table with the following data:

OBJECTID *	AddID	StreetNum	StreetName
1	1		
2	2		
3	3		
4	4		
5	5		
6	6		
7	7		
8	8		
9	9		
10	10		

The status bar at the bottom indicates "File Geodatabase Table selected".

Demo 6c: Search and Update Cursor

This demo uses the **addresses.shp** file that can be found in the `\PythonPrimer\Chapter06\Data` folder. Open ArcMap or ArcCatalog to see the various attributes and values of **addresses.shp** file as needed. In addition, a file geodatabase (**cursors.gdb**) is provided that contains the **Demo6c_update_address_table** that will be updated with the update cursor. Refer to the **Demo6c.py** as needed.

This demonstration illustrates the construction and use of a search cursor that contains the query (“where clause”) parameter to obtain 10 records from the **addresses.shp** file attribute table where `FID >= 1` and `FID <= 10`. For each of the 10 records in the shapefile the `STREETNUMB` and `STREETNAME` values are “read” using the search cursor and then are used with an update cursor to “update” the *StreetNum* and *StreetName* attributes in the **Demo6c_update_address_table** table in the file geodatabase.

1. Add the `arcpy`, `sys`, `os`, and `traceback` modules. Recycle the `except :` block from a previous script. Add some commentary describing the script.
2. Set a workspace to the data folder shown below for the shapefile and create a separate variable to hold the data path to the file geodatabase (**cursors.gdb**) using the `os.path.join` routine. Create a separate variable for the full path to the addresses table in the file geodatabase using `os.path.join`. Create a separate variable to the name of the *addresses.shp* file.

NOTE: The reader’s path may be different than shown below and depending on where the data is located.

```
import arcpy, sys, os, traceback

# set the current workspace (in the case a folder)
arcpy.env.workspace = 'C:\\PythonPrimer\\Chapter06\\Data\\'

workspace = arcpy.env.workspace

fgdpath = os.path.join(workspace, 'cursors.gdb')

address_shp = 'addresses.shp'
address_fgd = os.path.join(fgdpath,
'Demo6c_update_address_table')

try:
```

Note the script shows a “workspace” and a separate variable pointing to the data path to the file geodatabase using `os.path.join`. Only one current workspace can be set at a time. Notice that the workspace is set to the path where the shapefile is located. Also note the variable used to store only the name of the shapefile versus the variable used to store both the path and the name of the address table in the file geodatabase. The `address_fgd` represents a string that points to the entire path including the name of the file geodatabase name and the addresses table which exists within the file geodatabase. Only a variable is required to store the name of the *address.shp* file because the shapefile is located in the workspace.

Because no new feature classes or tables are being created, a workspace really is not needed and the code developer could have created a variable to hold just the path (or the path and file name) for the shapefile.

3. Next two variables are created; one to set the query statement for the search cursor and a second to set to the list of fields that will be passed to the search cursor. This demo will create a cursor that contains the rows of the **address.shp** file with FID values greater than or equal to 1 and FID values less than or equal to 10.

```
# set a variable to point to the query string
query = """FID" >= 1 AND "FID" <= 10"""

# set a variable to a Python list of field names for
# the search cursor these field are in the address.shp
# file

field_list = ['STREETNUMB', 'STREETNAME']
```

4. Create the search cursor using the `address_shp` variable that references the **address.shp** file, the list of fields (`field_list`), and the `query` variable as the input parameters to the search cursor.

```
# Get a collection of rows from a feature class or
# table

with arcpy.da.SearchCursor(address_shp, field_list, query) as
    srows:
```

5. Create a `for` loop structure to iterate over the records (features in this case) in the search cursor. Make sure to indent just after the “with” statement.
6. Create variables to hold the values of the *STREETNUMB* and *STREETNAME* attributes for the specific row in the cursor. Remember, these attributes are “indexed” starting with zero to the list of fields in `field_list`. The `obj_id` variable is used in the next step.

```

# this section of code is indented after the "with"
# line above

obj_id = 1    # used later in the update cursor query

# iterate through each row in the search cursor

for srow in srows:

    # assign a variable for the value of STREETNUMB
    # assign a variable for the value of STREETNAME

    streetnum = srow[0] # STREETNUMB
    streetname = srow[1] # STREETNAME

```

Once the *STREETNUMB* and *STREETNAME* attribute values are obtained for a row in the **addresses.shp** file, an update cursor can be constructed so that a row in the **Demo6c_update_address_table** file geodatabase table can be accessed to update the *StreetNum* and *StreetName* attributes within this table. Only a single row needs to be accessed in the **Demo6c_update_address_table** file geodatabase table. A query can be constructed to only access the proper row by using the *OBJECTID* field from the **Demo6c_update_address_table** table. In this example it is assumed that the Object IDs (*FID*) in the **addresses.shp** file match the Object IDs (*OBJECTID*) of the **Demo6c_update_address_table** file geodatabase table.

7. Set up the query and field list variables for the update cursor.

```

# Get a collection of rows for the update cursor
# based on a query
# This update cursor will only return a single
# record, which is desired.

query = """"OBJECTID" = "" + str(obj_id)

u_field_list = ['StreetNum', 'StreetName']

```

To create the proper syntax for the query, a variable is created and assigned a value of 1 (e.g. `obj_id = 1`) as shown above. This variable is placed just before the search cursor `for` loop that iterates over the search cursor rows. (*Actually, the `obj_id` variable can be defined anywhere before the update cursor is defined.*)

Since the idea for the update cursor is to update only a single row in the **Demo6c_update_address_table** file geodatabase table at a time, a specific row must be accessed. To do this a query must be set up to access the specific row (in this case the appropriate *OBJECTID* attribute value for a respective row). A variable (`obj_id`) can be used to dynamically access the given row. In this example the first row accessed is where the *OBJECTID* = 1 (the first row of the **Demo6c_update_address_table** file geodatabase table). Using a variable to store a value that can change with each iteration of the looping structure is desirable, since without a query to limit the update cursor to access a single row, the update cursor would iterate through all of the rows and update each value with the same value.

8. Create the update cursor using the **Demo6c_update_address_table** file geodatabase table variable (`address_fgd`), the `u_field_list`, and the `query` from above.

```
with arcpy.da.UpdateCursor(address_fgd, u_field_list, query) as  
    urows:
```

Notice the input data set for the update cursor is the **Demo6c_update_address_table** file geodatabase table and not the **addresses.shp** file feature class.

NOTE: The user may want to comment out the query statement and not use a query parameter in the update cursor to see what effect the query has when testing the demo code.

9. Create a `for` loop structure for the update cursor. Make sure to indent after the “with” statement. In this example since there is only a single record that is accessed in the query, the `for` loop only processes one time.

```

with arcpy.da.UpdateCursor(address_fgd, u_field_list, query) as
    urows:

    # cycle through the rows (in this case only 1)
    # to actually update the row in the cursor
    # with the values obtained from the search cursor

    for urow in urows:
        urow[0] = streetnum
        urow[1] = streetname
        urows.updateRow(urow)

    # increments counter to use in the next query just
    # before the update cursor

    obj_id += 1

print 'Finished Updating Rows in ' + address_fgd

del urow, urows, srow, srows    # free up memory

```

10. The `urow[0]` and `urow[1]` are set with the variables `streetnum` and `streetname` that are set to values read from the search cursor (**addresses.shp**). To update (i.e. “write to disk” or save) the actual row in the update cursor (which points to the **Demo6c_update_address_table** file geodatabase table), the `updateRow` routine is used and passed the specific row (`urow`, which contains the `streetnum` and `streetname` values). Alternatively, the following syntax could have been used to update the row and NOT use the `urow[0]` and `urow[1]` lines.

```
urows.updateRow((streetnum, streetname))
```

11. Note that the variable `obj_id` is incremented by 1 and is at the same indentation level as the line of code as the “with” statement for the update cursor (and also at the same indentation level as the lines within the `for` loop block for the *search cursor*. The purpose of this is to properly increment the *OBJECTID* attribute value (in the **Demo6c_update_address_table** file geodatabase table) so that the *search cursor* row matches that of the *update cursor* row and the correct row in the update cursor is updated properly (i.e. the row in the file geodatabase **Demo6c_update_address_table** table matches the same row in the **addresses.shp** attribute table, where **addresses.shp** *FID* = **Demo6c_update_address_table** *OBJECTID* attribute values).

12. A final `print` statement is written to let the user know the process is completed and has updated the addresses table in the file geodatabase. The cursors are also deleted to free up memory and prevent data locks. See the **Demo6c_update_address_table** for the results of this script.

Demo 6d: Joining Tables

Using the data described in Chapter 6, this demo will illustrate the steps needed to perform a table join between a feature class and a standalone table. NOTE: This method can also work with two feature classes or two tables, provided that table views and/or feature layers are properly constructed.

The **Demo6d.py** script can be found in the `\PythonPrimer\Chapter06\` folder. The data used for this demo can be found in the `\PythonPrimer\Chapter06\Data\cursors.gdb` file geodatabase. The **Lakes** feature class and **Lake_Info** table will be used in this demo.

1. Set up a workspace and variables for the feature class, feature layer, table, and table view. See below.

NOTE: The reader's path may be different than shown below and depending on where the data is located.

```
import arcpy, sys, os, traceback

# set the current workspace (in the case a folder)

arcpy.env.workspace =
'C:\\PythonPrimer\\Chapter06\\Data\\cursors.gdb\\'

# this is the Lakes feature class in the cursors.gdb

lake_fc = 'Lakes'

# this is the Lake_Info table in the cursors.gdb

lake_table = 'Lake_Info'

# this is a string that represents the Lakes feature layer

lake_fl = 'Lakes FL'

# this is a string that represents the Lakes_Info table view

lake_tv = 'Lakes TV'

try:
```

2. Create a list of indexes for both the feature class and the table to check if a specific index exists. The `ListIndexes` routine is used. If the index is found, then use the `RemoveIndex` routine to remove (delete) it. Refer to the **Demo6d.py** script which contains some additional in-line comments and explanation. Some of the code has been formatted to fit on the page. See the **Demo6d.py** script for the exact syntax.

```
try:

    # list the indexes for the Lakes feature class

    indexes = arcpy.ListIndexes(lake_fc)

    for index in indexes:
        # if the index name already exists in the list
        # of indexes, remove it

        if (index.name == 'Hydro_Index'):
            arcpy.RemoveIndex_management(lake_fc,
                                          'Hydro_Index')

    # list the indexes for the Lake_Info table; remove
    # the index if it already exists

    indexes = arcpy.ListIndexes(lake_table)

    for index in indexes:

        if (index.name == 'Lake_Index'):
            arcpy.RemoveIndex_management(lake_table,
                                          'Lake_Index')

    # create indexes for the feature classes and
    # tables the indexes help operations on joined
    # tables perform quicker

    arcpy.AddIndex_management(lake_fc, 'HYDRO24CA1',
                              'Hydro_Index', 'NON_UNIQUE', 'NON_ASCENDING')

    arcpy.AddIndex_management(lake_table, 'LakeID',
                              'Lake_Index', 'NON_UNIQUE', 'NON_ASCENDING')
```

3. Once the indexes are removed (provided they previously existed), run the `AddIndex` routine to create a new index. This is performed twice, once on the **Lakes** feature class and once on the **Lake_Info** table. Refer to the documentation above or the ArcGIS Help

for specific comments on the parameters. The *HYDRO24CA1* is the field in the feature class which will have an index; the name of the index is '*Hydro_Index*' which is just a string. The *LakeID* is the field in the table that will have an index; the name of the index is '*Lake_Index*'. The '*NON_UNIQUE*' and the '*NON_ASCENDING*' parameters are the defaults for the `AddIndex` routine.

4. Since the join requires a feature layer and a table view (and not a feature class or a table), these must be created. The next steps show the `MakeFeatureLayer` and the `MakeTableView` to create the feature layer and the table view for the **Lakes** feature class and the **Lake_Info** table, respectively.

```
# Create feature a feature layer and table view
# so the AddJoin routine can run
# check to see if the feature layer and table view
# already exist, if so, delete it

if arcpy.Exists(lake_fl):
    arcpy.Delete_management(lake_fl)

if arcpy.Exists(lake_tv):
    arcpy.Delete_management(lake_tv)

# Make Feature Layer from the Lakes feature class
arcpy.MakeFeatureLayer_management(lake_fc, lake_fl)

# Make Table View from the Lake_Info table
arcpy.MakeTableView_management(lake_table, lake_tv)

# Create the Join using the common attribute
# between the two data sets Lakes feature class
# (HYDRO24CA1); Lake_Info table (LakeID)

arcpy.AddJoin_management(lake_fl, "HYDRO24CA1",
    lake_tv, "LakeID", "KEEP_ALL")

# print out a list of fields so the code developer
# can get an idea of what the field names look
# like this is really not needed, but can help in
# troubleshooting programming problems

fields = arcpy.ListFields(lake_fl)

for field in fields:
    print field.name
```

5. Once the feature layer and table view are created, the join can occur by using the AddJoin routine. Notice the feature layer (*lake_fl*) will have the table view (*Lake_tv*) joined to it using the *HYDRO24CA1* and the *LakeID* fields. All of the records from the resulting join will be shown using the *KEEP_ALL* parameter. See above.

6. When developing code for joining tables, it might be a good idea to list out the field names resulting from the join. This can be useful because the field names change slightly when performing a join and take the form:

```
<feature class or table name>.<field name>
```

The `arcpy.ListFields` routine can be used to access the fields from a feature class or table. To print out a list of the fields to the Python Shell a `for` loop can be used to print the field names. Lists will be discussed in Chapter 7. This step is not required for using fields and values from joined data, but it may be helpful for troubleshooting purposes. See the above code.

7. After the data sets have been joined, specific records and values can be accessed and used. A search cursor is implemented to retrieve a set of records from the data. In this example, since some **Lake_Info** *LakeID* attribute values do not match the **Lakes** *HYDRO24CA1* attribute values, the code developer only wants to obtain records that have actual values in both the feature attribute table (**Lakes**) and the standalone table (**Lake_Info**), a query is used to limit the records returned from the search cursor.

The query string is:

```
query = lake_fc + '.HYDRO24CA1 = ' + lake_table + '.LakeID'
```

Note that the `lake_fc` and `lake_table` variables are used to construct the query. The programmer could have “hard coded” the specific feature class and table names, but if the names of the feature class and table change while maintaining the same attribute names, the code remains flexible for this possibility.

This is another place that code developers are often challenged and one reason why printing out the joined field names may be useful. Listing out the field names can help the code developer write the correct syntax for the query. The query string follows the syntax shown above for joined field names. In this case, the query will retrieve records where the *HYDRO24CA1* field value from the **Lakes** feature class equals the *LakeID* field from the **Lake_Info** table. The user may find it useful to print the query statement to the Python Shell when troubleshooting code.

8. Since a search cursor is used to read values from the joined table, a list of fields is required. The only fields that are required for processing need to be in the list of fields. The “\” shown below is a Python construct that provides the ability to divide a long line of Python code into smaller pieces. This can make code writing more readable.

```
# the list of fields that include the join
# feature class/table names

field_list = [lake_fc + '.HYDRO24CA1', \
              lake_fc + '.HNAME', \
              lake_table + '.LakeID', \
              lake_table + '.Temp_F']
```

9. The search cursor can now be created using the syntax below.

```
with arcpy.da.SearchCursor(lake_fl, field_list, query)
    as srows:
```

10. A for loop can be used to cycle through the search cursor and read values from the joined data. Remember to indent just after the “with” statement. Since the field list already contains the feature class or table name, the programmer simply references the list index to obtain the specific attribute value. The comments show which feature class or table and the respective field name are referenced by each field list index.

```
for srow in srows:

    # Read values from the joined table

    HydroID = srow[0] # lake_fc + '.HYDRO24CA1'
    LakeName = srow[1] # lake_fc + '.HNAME'
    LakeID = srow[2] # lake_table + '.LakeID'
    temp_F = srow[3] #lake_table + '.Temp_F'
```

11. Once the records have been retrieved by a cursor, specific values can be accessed and used. In this case, the values are simply printed to the Python Shell, but can easily be used with other cursor methods or processes.

```
print lake_fc + ' HYDRO24CA1 is: ' + str(HydroID)
print lake_fc + ' Lake Name is: ' + LakeName
print lake_table + ' LakeID is: ' + str(LakeID)
print lake_table + ' Temperature is: ' +
    str(temp_F) + '\n'
```

12. Once the records have been accessed and processed the join is no longer used in the script and can be removed by using the `RemoveJoin` routine. The `RemoveJoin` is not indented as part of the `for` loop, but is in line with the “with” statement of the search cursor. If this were not the case, the join would be removed after the first iteration of the loop, which is not desired.

```
# remove the join, since it is no longer used
arcpy.RemoveJoin_management(lake_fl, lake_table)
```

The reader may notice that the `RemoveJoin` routine is using a feature layer and a table and not a table view. The reason for this is the `RemoveJoin`’s second parameter (“join name”, a string) requires the name of the table, not the name of the table view while the first parameter (input feature layer or table view), requires the feature layer. In this case the **Lake_Info** table name is “Lake_Info”; whereas, the table view name is “Lakes TV”. If the table was “aTable.dbf”, the join name would be “aTable”. See the ArcGIS Help for the `RemoveJoin` routine.

13. The cursor pointers are deleted to help eliminate data locks using the `del` routine.

```
del srow, srows
```

Exercise 6 - Using Cursors and Table Joins

Exercise 6 will bring together a number of elements already experienced in the book. Part of Exercise 6 provides the opportunity for an analyst to do some research to find some of the answers for specific tools that are required as well as the parameters that are needed. Much of the code is already provided. This exercise will have the reader add a few key lines of code regarding some variables required to perform the `AddJoin` routine (already provided in the code), creating a couple of Python lists, and writing the key lines for the insert cursor. Optionally, the “joined” data sets can be written to a separate output feature class. Refer to the *Python Primer* text as well as ArcGIS Tool Help to identify these tools and parameters. Use the `\PythonPrimer\Chapter06\Exercise6.py` script for the exercise. Read through the entire exercise before beginning.

The **Exercise6.py** script contains comments that begin with

```
# !!!! <comment> !!!!!
```

to indicate the sections in the code where changes need to occur. The comments will also be indented where appropriate to help the reader place the code in the correct location.

Augment the **Exercise6.py** script with the following conditions.

- Manually create a file geodatabase in the **Chapter06\MyData** folder. This geodatabase will be used to store the output table and optional feature class.
- Using **Exercise6.py** as a starting point, instead of writing the joined values to the Python Shell, write the values to a new output table within the file geodatabase created above. The geodatabase will be created in ArcCatalog. Creating and populating the new output table with records will be done programmatically. See below.
- Use an insert cursor method to actually write out the joined values to the new table.
- Optional** - In addition to the table, write the joined features out to a new feature class within the same script using the `CopyFeatures` routine. Do not use a cursor to do this.

Recommendation

After making the file geodatabase, copy the **Chapter06** folder to a different location as a “back up,” just in case data or files are deleted. Also make sure to write the Python script in a separate folder from **\PythonPrimer\Chapter06\Data** or **\PythonPrimer\Chapter06\MyData**. If a workspace (instead of a feature class or table) is deleted by accident and the workspace is a folder (such as **\Chapter06\Data**) any files in this folder may be deleted (any kind of file). It is good practice to write and process the script in a different location than where data is written to, especially if data is to be deleted.

STEP 1 – Create a File Geodatabase for the Output

1. In ArcCatalog manually create an empty file geodatabase in the **MyData** folder. The geodatabase will have a **.gdb** extension.
2. Add a variable to the **Exercise6.py** script that is set to the file geodatabase location just created. For example,

```
outpath = 'c:\\PythonPrimer\\Chapater06\\MyData\\out_fgdatabase.gdb'

# Note, no trailing '\\' and this is NOT a workspace, since
arcpy.env.workspace is not used
```

STEP 2 – Create a New Output Table

1. Create a variable that is set to an output table name. Remember, the table will be stored in the file geodatabase and thus will not have a file extension (such as .dbf). For example,

```
out_table = 'my_table'
```

2. Create a variable that uses the `os.path.join` routine to combine the output path and the output table name created above. This variable of the “joined” path and table will be used in the check to see if the output table exists, the `AddField` routine, and the `Insert Cursor`
3. Create another variable that is set to an output feature class name.
4. Create a variable that uses the `os.path.join` routine to combine the output path and the output feature class name. (This should look very similar to the output path and table name. This will be used for the optional `CopyFeatures` routine below.
5. Within the `try` block add a section to check to see if the table exists and if so delete it. The variable used to combine the output path and table name will be used here. For example,

```
if arcpy.Exists(outpath_table):  
    arcpy.Delete_management(outpath_table)
```

6. Add the `CreateTable` routine to create the new table. Pay close attention to the parameters. The variables created above should be used as the parameters to create the output table.

STEP 3 – Add Fields to the New Table

Use the `AddField` routine to add the following fields with the specified data type. Again pay close attention to the “table” name. The variable pointing to the combined output path and table name will be used. The field definitions are shown below. The `AddField` routine will need to be used four separate times, one for each attribute field.

- a. **LakeFC_ID** - short
- b. **Lake_Name** – text, 50
- c. **Lake_Info_ID** – long
- d. **Lake_Temp** – short

NOTE: Pay close attention when adding the field names and using them in the field list for the insert cursor. If the field names do not match exactly in the code, a “Query.Empty” error may appear when performing the insert cursor.

STEP 4 – Create a Python list of the “added fields”

After the `AddField` routines are added to the script, define a variable that points to a Python list of the field names used above. The Python list will look similar to this:

```
aList = ['field1', 'field2', 'field3', etc]
```

This list will be used in the Insert Cursor below.

STEP 5 – Set up and Implement the AddJoin

This set of code is already provided and sets up the data and performs the `AddJoin` routine between the feature class (**Lakes**) and the table (**Lake_Info**). This step includes creating indexes, “making” a feature layer and table view in addition to performing the `AddJoin` routines.

A `for` loop and set of print statements print out the “joined” data fields

STEP 6 – Create the Insert Cursor

Write the code for an insert cursor. The variable created for the combined output path and output table will be used as the table in the insert cursor. The Python list created above will be used as the “list of fields” parameter in the insert cursor.

STEP 7 – Create a Python list of the “joined” attribute fields

A separate Python list variable (`field_list`) is already provided on the “joined” data which will be used in the Search Cursor. Notice that this step is indented after the Insert Cursor code created above.

STEP 8 – Create a query for the Search Cursor

A query statement is already provided that uses the “joined” fields created above. Notice the format of the query. This is similar to the syntax discussed earlier in the chapter.

STEP 9 – Create the Search Cursor

The search cursor and for loop are provided. Notice the “`lake_fl`” (feature layer variable already provided in the code) is used as the data set that the search cursor will be applied to. The variable `field_list` already provided and mentioned above is used as the “list of fields” parameter. The `query` variable created above is used as the query (“where clause”) parameter.

The `for` loop starts the process of the search cursor. Variables are already set up and assigned to the different row/field in the search cursor. Notice the `srow[x]` reference that points to a particular row and column in the search cursor. Several print statements are provided so that the values can be reviewed in the Python Shell.

The table below shows the cross reference of the respective data set (feature class or table), the specific data field referenced in the `field_list` Python list, and the variable used to store the specific values from the search cursor.

Data Set	Field from Joined Data	Output Field
Lakes (feature class)	Lakes.HYDRO24CA1	HydroID
Lakes (feature class)	Lakes.HNAME	LakeName
Lake_Info (table)	Lake_Info.LakeID	LakeID
Lake_Info (table)	Lake_Info.Temp_F	Temp_F

STEP 10 – Insert the new row and values to the output table

Once the values from the feature layer (i.e. the lakes feature class), they can be “inserted” into a new row in the output table. This step is already provided. Both the creation of a new table record and the assignment of specific data values are accomplished in one line of code.

The format will be similar to the following:

```
irows.insertRow((value1, value2, value3, value4))
```

`irows` is the reference to the insert cursor

`((value1, value2, value3, value4))` will be replaced with variables created within the search cursor (already provided in the code).

The output table should look like this.

OBJECTID	LakeFC_ID	Lake_Name	Lake_Info_ID	Lake_Temp
1	74	FOLSOM LAKE	74	65
2	5	HINKLE RESERVOIR	5	73
3	109	LAKE NATOMA	109	72
4	66	BASS LAKE	66	63
5	8	WILLOW HILL RESERVOIR	8	75

STEP 11 – Optional steps to create the feature class

1. In addition to the above insert cursor, the joined data can be written to a unique output feature class.
2. Outside of the search and insert cursors (i.e. at the same indentation level of the insert cursor add a check to see if the output feature class exists. If it does, delete it. Use the variable created above that uses the output path and the feature class names.
3. Use the `CopyFeatures` routine to copy the joined data to the output feature class. The input will be the feature layer used in the `AddJoin` routine. This line will be added before the `RemoveJoin` routine.
4. A print statement can be added to indicate the `CopyFeatures` routine is complete.
5. Add the variable (e.g. `irows`) created above for the insert cursor to the `del` line.

The output feature class should have 7 records.

Chapter 6 Questions

These questions cover the content in Chapter 6, the demonstrations, and the exercise.

1. Briefly describe what each cursor does and an example of when it might be used.
 - a. Search
 - b. Insert
 - c. Update
2. What is the purpose for creating attribute indexes?
3. What feature or table type does a table join require?
4. Describe the parameters required for a table join.
5. Describe how the attribute names change after the table join.
6. Describe how the data can be accessed in a joined table.

From the Exercise

1. What tool is used to create a new table?
2. What tool is used to create new attributes in the table?
3. Name the cursor type used to write out records in the exercise.
4. Describe where in the code the following occur:
 - a. Where is the insert cursor created?
 - b. Where are records updated?

Optional

5. If features were written out to a feature class, what tool was used?
6. What do the attributes in the feature class look like? How are the attribute names different from the respective feature class or table attribute names used in the join?

Chapter 7 Describing Data, Lists, and Raster Processing Basics

Overview

Chapter 7 focuses on programmatically obtaining information for different kinds of data sets. This kind of process is often used on folders or geodatabases that can contain numerous types of data. A useful way of processing such data is to use lists. The fundamental Python list was described earlier. The types of lists discussed here are those found within the `arcpy` module (such as `ListFeatureClasses`, `ListRasters`, and `ListDatasets`, among others). The ArcGIS lists operate in the same manner—create a list, then iterate over the list using a `for` loop.

Chapter 7 also discusses some basic raster (image) processing fundamentals using Spatial Analyst and the Spatial Analyst (`sa`) module. This chapter only introduces raster processing so the programmer can understand how to access images and image bands (i.e. the individual components of an image, since image data can have one or more bands). Chapter 7 will not cover complicated custom “algorithms” (math expressions written in computer code) or the Python `numpy` module that is often used for statistical, matrix algebra, and higher end mathematical functions. The reader is encouraged to consult other sources to learn more about the `numpy` module and more complicated numerical routines.

Describing Data

When working with a variety of data formats (feature classes, tables, images, geodatabase, CAD, networks, etc.), especially when writing Python scripts, a code developer may need to access properties that can be used to make decisions for processing. The `Describe` ArcGIS routine provides this ability and will work with all of the formats supported by ArcGIS. For a full account of the data properties refer to the ArcGIS Help in **Geoprocessing—The ArcPy site package—Functions--Describing data**.

To access data properties the `Describe` routine is used which is able to accept many different kinds of properties depending on the data type. Before specific properties can be acquired from the data set, the `Describe` routine must be used to “describe” or obtain properties about a specific dataset. To do this a variable is assigned to the results of the `Describe` routine of the data set. For example, to access specific properties of an image data set from a satellite sensor, the `Describe` syntax may look like the following:

```
in_image = 'c:\\images\\landsat.img'
desc_img = arcpy.Describe(in_image)
```

Once the `Describe` routine is defined for a given dataset, specific properties of the data set can be accessed and used. In this simple example below, the spatial reference, number of sensor bands, and image format are printed to the Python Shell.

```
import arcpy, sys, traceback

arcpy.env.workspace = 'C:\\PythonPrimer\\Chapter07\\Data\\'

in_image = 'tm_sacsub.img'

# Create describe object
desc_image = arcpy.Describe(in_image)

try:

    spat_ref = desc_image.spatialReference.name
    num_bands = desc_image.bandCount
    img_format = desc_image.format

    print 'Spatial Reference: ' + str(spat_ref)
    print 'Number of Bands: ' + str(num_bands)
    print 'Image format: ' + img_format
```

Since an image is being used as the parameter for the `Describe` routine, properties about images can be retrieved (such as the number of bands or format). See ArcGIS Help in **Geoprocessing—The ArcPy site package—Functions--Describing data—Describe properties—Raster dataset properties**. In addition, more general properties can be obtained, such as the spatial reference. See ArcGIS Help in **Geoprocessing—The ArcPy site package—Functions--Describing data—Describe properties—Dataset properties**. Dataset properties are those properties that apply to almost all geospatial datasets such as spatial reference, data type, and spatial extent, among others. Code developers will need to become familiar with the type of data they are using in their scripts and then consult the `Describe` properties that pertain to their needs.

Using `Describe` properties can be used in code to make decisions (such as with the use of the `if` conditional statement). For example, a conditional statement can be used to check to see if a spatial reference is associated with the data set. If one does not exist, then a `print` statement can be used to report this back to the user (or possibly printed to a “log file”). Log files will be discussed in Chapter 8). If a spatial reference does exist, then geoprocesses can be

added to perform one or more tasks on the data set. The example below shows a conditional statement that checks for a spatial reference. If a spatial reference exists, then another conditional statement is written to see if the image has six bands. If it does, then a statement is written to process an algorithm on the image (the Normalized Difference Vegetation (NDVI), in this case). NDVI is an algorithm that has been developed to quantify how much healthy biomass exists in a remotely sensed image. The reader should also note that to perform the NDVI, the Spatial Analyst extension is used. The script has been formatted to fit the page. See **Demo7a.py** for the actual script showing the proper format.

```

try:

    spat_ref = desc_image.spatialReference.name
    num_bands = desc_image.bandCount
    img_format = desc_image.format

    print 'Spatial Reference: ' + str(spat_ref)
    print 'Number of Bands: ' + str(num_bands)
    print 'Image format: ' + img_format

    if spat_ref <> 'Unknown':

        if num_bands == 6:

            # Process Normalized Difference Vegetation
            # Index (NDVI)

            # (Raster Band 4 - Raster Band 3) / (Raster
            # Band 4 + Raster Band 3)

            # See Demo7a for the variable definitions for
            # band3 and band4 that goes here

            print 'Starting NDVI...'

            NDVI = Float(Raster(band4) - Raster(band3)) / \
                    Float(Raster(band4) + Raster(band3))

            print 'Saving ' + NDVI_image

            if arcpy.Exists(NDVI_image):
                arcpy.Delete_management(NDVI_image)

            NDVI.save(NDVI_image)

        else:

            print 'This image does not have 6 bands, \n' +
                  'the algorithm will not be processed'

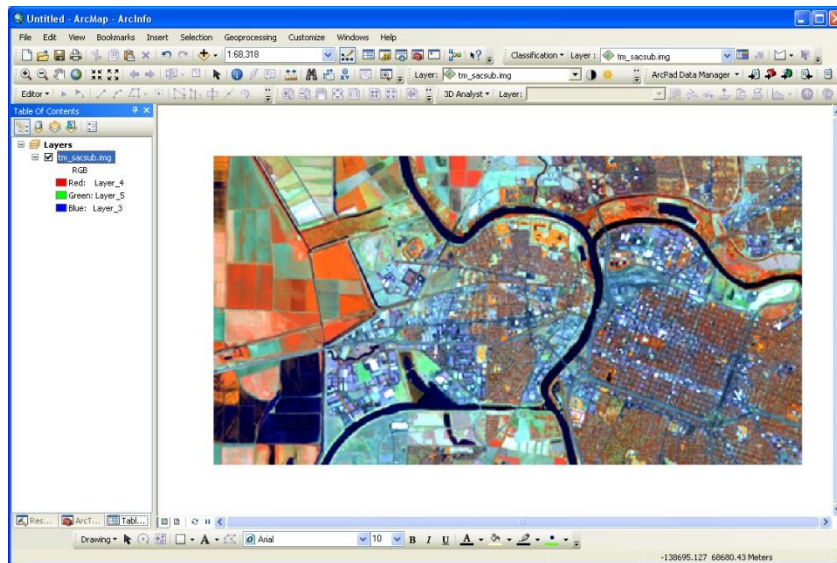
    else:

        print in_image + ' does not have a spatial reference'

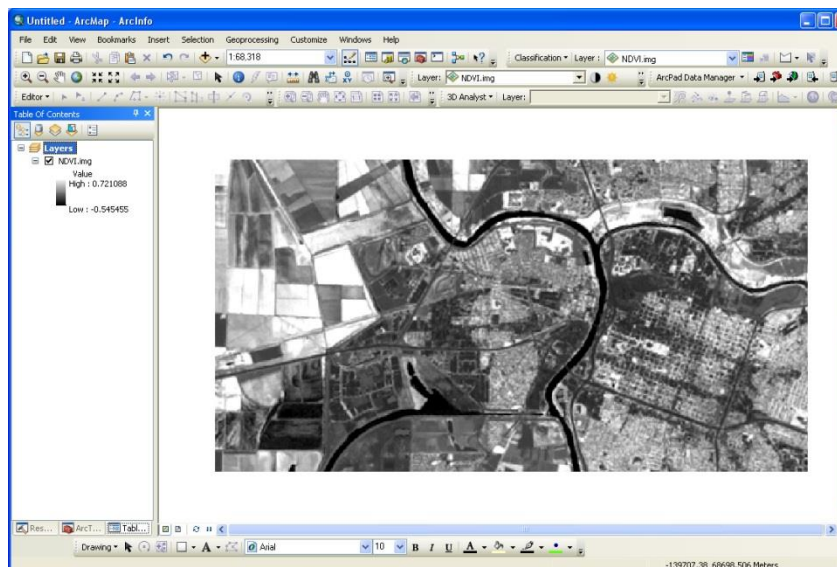
except:

```

The original image (**tm_sacsub.img**) and the resulting image from the script (**NDVI.img**) are shown below. Both images can be found in the **Chapter07** folder.



Original Landsat Image subset (tm_sacsub.img).



NDVI image result generated by the Python script (NDVI.img).

Once the algorithm is processed, the resulting image is then written to an output file (**NDVI.img** using the `NDVI_image` variable and the raster save routine).

If the image data set does not have a spatial reference a print statement is provided to let the user know that one does not exist. In addition, if the image does not have six bands, another print statement is used to let the user know that the NDVI algorithm will not process the image.

Listing Data

Lists are used as primary structures to perform iterative tasks and are at the heart of being able to execute “batch processes” (i.e. performing numerous geoprocessing tasks in sequence). To process a list, one must be created. A number of listing routines exist in ArcGIS and can be used in Python scripting. Typically, a list involves developing a specific list of data types. Once a list is created, then individual elements of the list can be processed. For example, a list of shapefile feature classes can be created and then iteratively processed to convert them into file geodatabase feature classes. See the ArcGIS Help for more details on the different kinds of list routines.

The example below illustrates the use of the `ListFeatureClasses` routine to convert shapefiles to file geodatabase feature classes. See **Demo7b.py** for the actual script.

The general syntax for the `ListFeatureClasses` routine is:

```
arcpy.ListFeatureClasses({wild_card}, {feature_type},  
                        {feature_dataset})
```

1. *wild_card* – indicates a set of characters to help limit the list (e.g. all feature classes ending with extension “*.shp”). The “*” indicates to make a list of “all” shapefiles in the workspace.
2. *feature_type* – indicates to limit the list to the type of feature (point, line, polygon, etc). See the ArcGIS Help documentation for a full list
3. *feature_dataset* – indicates that the list can be limited to feature classes found in a feature dataset. Default is empty and will return only standalone feature classes (i.e. those not found in a feature dataset)

Notice that all of the parameters for the `ListFeatureClasses` routine are optional. In many cases, the code developer may choose to create a list of all feature classes in a given workspace. In this case, the following syntax becomes:

```
arcpy.ListFeatureClasses()
```

The following script shows an example of using the `ListFeatureClasses` routine to copy shapefiles from a workspace to a separate folder containing a file geodatabase. Refer to the **Demo7b.py** script to show the proper syntax format. The script below has been modified to fit the page.

```
import arcpy, os, sys, traceback

arcpy.env.workspace = "c:\\PythonPrimer\\Chapter07\\Data\\"

workspace = arcpy.env.workspace

fgdb_path = "c:\\PythonPrimer\\Chapter07\\Mydata\\"

fgdb = os.path.join(fgdb_path, "Ch07_fgdb.gdb")

fclist = arcpy.ListFeatureClasses("*.shp")

for fc in fclist:

    print str(fc.split(".")[0]) # split the feature class root name
                                # and the extension ("shp")
                                # at the dot "."

    shaperoot = fc.split(".")[0] # assign a variable for
                                # cleaner code

    if arcpy.Exists(os.path.join(fgdb, shaperoot)):
        arcpy.Delete_management(os.path.join(fgdb, shaperoot))

    print "Deleted " + os.path.join(fgdb, shaperoot)

    print 'Copying ' + fc + ' to ' + fgdb
    arcpy.CopyFeatures_management(fc, os.path.join(fgdb,
        shaperoot))
```

After defining the workspace and output folder for the file geodatabase, the `ListFeatureClasses` routine is used to create a list of all the shapefile feature classes found in the workspace. A `for` loop is then used to process each shapefile and copy it to the file geodatabase location. Notice the use of both the `os.path.join` and the Python `split` routine. The `split` routine can be used to separate the root file name from the extension and then use it to create the feature class in the file geodatabase using the `os.path.join` routine. In the example above, the `split` routine creates a Python list of strings that represent the characters to the left or right of the “.” (dot). The `[0]` references the first element in this string (in this case the shapefile root name). The shapefile root name can then

be used in other code and parameters such as for copying shapefile feature classes to a file geodatabase. Notice how the shapefile root name is used to create a full path to the output file geodatabase feature class (since file geodatabase feature classes do not have a file extension). This is an efficient way of using variables and Python syntax to perform geoprocessing tasks on data using the functionality of the `arcpy` module and other Python syntax.

With a relatively small script that uses a list routine, converting feature classes from one format to another becomes more automated. The script's generic form indicates that this it can be implemented many different times with only a few small changes in the code (primarily, the workspace location of the input feature classes that make up the list and the output file geodatabase location). In addition, the number of feature classes can vary within the folder, so the script can operate on one or even hundreds or thousands of feature classes with the same script.

Raster Processing Basics using the Spatial Analyst Module

GIS data can take two primary forms, *vectors* and *rasters*. Most of us are familiar with points, lines, and polygons. Some of us are familiar with rasters (or images) and may need to do some basic image processing or perform spatial analysis on images. Image processing can occur on multi-band images derived from remote sensors such as aerial imaging systems or satellites. Image processing can also occur on thematic data stored as rasters such as land cover, soils, elevation, slope, etc.

Image data is stored in pixels (picture elements) and are organized into rows, columns, and bands. An image can have multiple bands where most of the time the individual bands on a remotely sensed image represent a specific wavelength of light energy captured on the sensor. The individual pixels store the “brightness value” that represents the relative intensity of reflected energy for a given wavelength captured on the sensor. Thematic rasters are often single band images where the pixels represent different values of a given theme. For example, individual pixels in a land cover raster can represent different land over types. Pixel values are numbers (often integers, but can be floating decimal point numbers, too) where the numbers represent the value of the wavelength or thematic value.

As has already been seen, rasters have properties that can be accessed and the Spatial Analyst (`sa`) extension can be enabled to perform various raster processing tasks (such as “Band Math” using individual bands to create custom algorithms on a multi-band image data set) or by using one of the Spatial Analyst geoprocessing tools such as the `Hillshade` algorithm to create a hill shade data set from a digital elevation model (DEM). Multiple raster algorithms can be processed sequentially to perform spatial analysis modeling efforts such as hydrological or

ecological models). For geoprocesses not found within the Spatial Analyst (`sa`) or Geostatistical Analyst (`ga`) extensions, Python `numpy` arrays and other Python modules can be used. The Geostatistical Analyst and `numpy` arrays are beyond the scope of the book. Readers are encouraged to consult the ArcGIS Help or other Python resources for more information.

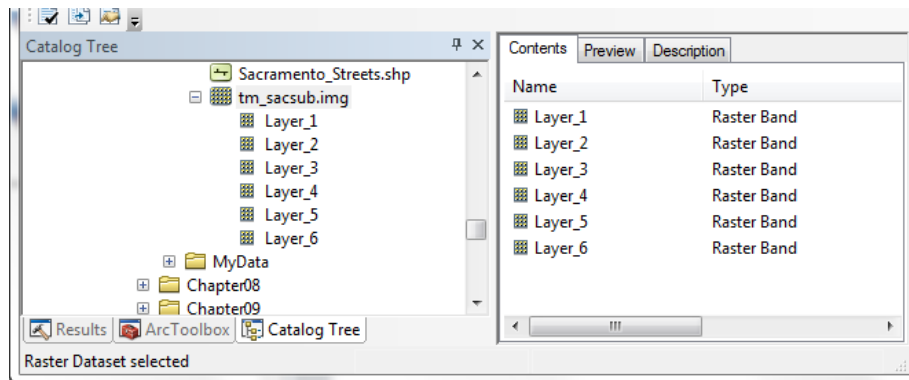
The NDVI (Normalized Difference Vegetation Index) has already been shown. In order to actually perform this algorithm, the Spatial Analyst extension needs to be “checked out” and the `arcpy` module (`sa`) needs to be imported. Checking out the spatial analyst extension allows Python to use the raster processing functionality contained within this extension (and is similar to checking the Spatial Analyst box under the **Customize—Extensions** in ArcGIS). Note that Spatial Analyst functionality cannot be used in Python without the purchase and installation of the extension on the system running the script. Spatial Analyst is an additional extension to the base functionality of ArcGIS.

```
import arcpy, sys, os, traceback
from arcpy.sa import *

# Check out the Spatial Analysis Extension
arcpy.CheckOutExtension("spatial")
```

Raster Data Organization

Before working with image data with Python scripting and `arcpy`, knowing the image data structure is important so that Spatial Analyst geoprocessing routines can be implemented correctly. Image data is often accessed by choosing the image file name to add it to ArcMap or view its properties in ArcCatalog. If an image contains multiple bands (such as an aerial or satellite image) the user can click “twice” on the image to access the individual bands. When doing so the data preview looks like the following:



Notice the individual “bands” are labeled “Layer_1”, “Layer_2”, “Layer_3”, etc. To access or use individual bands in geoprocessing scripts, both the root image name and the layer name need to be referenced.

Accessing and Using Image Bands

One way to access and use the individual bands is to use the `Describe` routine.

```
arcpy.env.workspace = "C:\\PythonPrimer\\Chapter07\\Data\\"
in_image = "tm_sacsub.img"
# Create describe object
desc_image_ = arcpy.Describe(in_image)
# the name of the image (i.e. tm_sacsub.img)
print desc_image.name
# a variable pointing to the specific band (i.e. "Layer_3")
# in the image
band3 = arcpy.Describe(os.path.join(desc_image.name, "Layer_3"))
```

Doing so allows the programmer to obtain information about a specific band, such as the number of rows and columns.

```
print "Rows and Columns are: " + str(band3.width) + " and " +
str(band3.height)
```

Another way to access individual image bands can be to simply assign them to variables. This can be a useful method if no other properties about the bands are needed.

```
band3 = os.path.join(desc_image.name, "Layer_3")
band4 = os.path.join(desc_image.name, "Layer_4")
```

After accessing the required bands, algorithms (math expressions expressed in programming code), can be used with them. The “\” indicates the line of code continues on the next line. The “/” is the division symbol for this algorithm.

```
NDVI = Float(Raster(band4) - Raster(band3)) / \
      Float(Raster(band4) + Raster(band3))
```

Summary

Up to this point, the examples have shown only a few actual Python coding structures (e.g. variable names, conditional statements, and looping structures). The code developer may find it useful to implement other Python structures like those shown in this chapter (e.g. `split`). This will require the developer to research and understand other Python structures and test them in the respective code. ArcGIS Help may provide some clues or hints to these other structures. Also, once coding elements are found and successfully developed, they can become part of the code developer's "code repository" or "portfolio" of script structures and processes that can be used in other programming routines.

As illustrated in this chapter describing data provides some additional specific access to data properties that can be used to make decisions as well as using lists to iterate through groups of data so that common repetitive tasks can become more automated through the use of scripting. The `Describe` and `List` functions allow scripts to become more dynamic so that processing can be implemented on different data types and for different needs.

Understanding the raster structure and how to access and use image and image band information provides the fundamentals to perform the Spatial Analyst geoprocessing tasks or develop custom image processing algorithms that are not part of the core ArcGIS functionality.

Chapter 7 Demos

Demo7a and **Demo7b** illustrate the tasks discussed above using the `Describe` and `ListFeatureClass` routines as well as implement the raster data structure.

The concepts illustrated in the demos and exercises are:

ArcGIS Concepts

Describe
ListFeatureClasses
ListRasters
CopyFeatures
Exists and Delete
Spatial Analyst (sa) module
Raster processing
Buffer
Building queries using variables as values
SelectByAttribute
ExtractByMask
IsoClusterUnsupervisedClassification
Save Rasters
Build Image Pyramids

Python Concepts

for loop
if...else conditional statements
Python lists
Indentation
Casting numbers to strings
os module
os.path.join()
replace()
split()

Demo 7a: Describe Image Properties and Perform Image Processing

The following demonstration shows how to create and use the `Describe` routine for an image and then set some variables for some of the properties of the image. The demonstration code checks to see if a spatial reference exists for the image; if it does, then a second check is performed to see if the image has six bands. If the image has six bands, then the Normalized Difference Vegetation Index (NDVI) algorithm is performed. The Spatial Analyst extension is used since the algorithm uses band math to perform the raster (image) function. The data can be found in the `\PythonPrimer\Chapter07\Data` folder. An example of the NDVI output image is provided (`NDVI_out.img`). The images and their properties can be viewed in ArcMap.

This demo uses the `tm_sacsub.img` ERDAS Imagine formatted image. This image is a subset of a Landsat TM (Thematic Mapper) satellite image for a portion of Sacramento, CA. NOTE: The workspace may need to be changed for this demo to perform correctly on the user's system.

1. Import the modules required for the program. In addition import the Spatial Analyst module, since this script uses band math, which requires the Spatial Analyst extension. For this demo it is not important to know why this syntax is used with the Spatial Analyst (`sa`) Python module.

```
import arcpy, os, sys, traceback
from arcpy.sa import *
```

2. Check out the Spatial Analyst extension. Checking out the extension is the method to use so that the functionality of the extension can be accessed within a script. The `CheckOutExtension` routine performs the same function as “checking” the box under the **Customize—Extensions** ArcGIS menu option.

```
arcpy.CheckOutExtension("spatial")
```

3. Set up a workspace to a folder that contains the Landsat image

```
arcpy.env.workspace =
    'C:\\PythonPrimer\\Chapter07\\Data\\'
```

4. Set up some variables, one for the input (**tm_sacsub.img**) and another for the NDVI image (**NDVI.img**) result.

```
in_image = 'tm_sacsub.img'
NDVI_image = 'NDVI.img'
```

5. Set up the `Describe` routine for the Landsat image. Review the `Describe` routine in the ArcGIS Help for more details describing raster data.

```
desc_image = arcpy.Describe(in_image)
```

The code should look like the following. The syntax has been modified to fit the page. See the **Demo7a.py** for the correct syntax.

```
import arcpy, os, sys, traceback
from arcpy.sa import *

# Check out the Spatial Analysis Extension
arcpy.CheckOutExtension("spatial")

arcpy.env.workspace = 'C:\\PythonPrimer\\Chapter07\\Data\\'

in_image = 'tm_sacsub.img'
NDVI_image = 'NDVI.img'

# Create describe object
desc_image = arcpy.Describe(in_image)

try:
```

Next create some variables for different properties of the image.

6. Create variables for some of the image properties. See the ArcGIS Help for more information on the specific properties available for raster (image) data.
 - a. Spatial reference
 - b. Number of bands
 - c. Image format

```
spat_ref = desc_image.spatialReference.name
num_bands = desc_image.bandCount
img_format = desc_image.format
```

7. Some print statements are provided to show that values are actually obtained from the different image properties. These are just to illustrate some of the `Describe` values.

```
print 'Image Name: ' + desc_image.name
print 'Spatial Reference: ' + str(spat_ref)
print 'Number of Bands: ' + str(num_bands)
print 'Image format: ' + img_format
```

8. A `for` loop is added to show some properties of individual bands and to set a couple of variables representing specific image bands that will be used later in the code. The `for` loop begins by looping through bands 1 to `num_bands+1` (i.e. in `range (1, num_bands+1)`). The loop begins with `band = 1` and then loops over all of the image bands (1-6). The “ending” value for the loop must be `num_bands + 1` so that the loop continues to process through `band = 6`. A simple snippet of code can illustrate this point. The reader can try this at the Python Shell. The numbers 1-6 will print in the Python Shell.

```
num_bands = 6
for band in range (1, num_bands + 1):
    print band
```

Notice the use of `os.path.join` to join the image name (`desc_image.name` and the “layer” (i.e. name of the individual band) for the `band_name` and the respective `band` variables). Notice also the “layer_” and the `str(band)` strings are concatenated together to create the actual “layer” name (e.g. “layer_3”).

```
for band in range (1, num_bands+1):

    band_name = arcpy.Describe
    (os.path.join(desc_image.name, "layer_" + str(band)))

    print "Rows and Columns for Band" + str(band) + " is " + \
    str(band_name.width) + " and " + str(band_name.height)

    if band == 3:

        band3 = os.path.join(desc_image.name, "layer_" + str(band))
        print band3

    if band == 4:

        band4 = os.path.join(desc_image.name, "layer_" + str(band))
        print band4
```

9. The next section of code is “out dented” to the same indentation level as the `for` loop. An `if` statement is added to check if a spatial reference value exists. A second conditional statement checks to see if the number of bands equals 6. Six is the number of functional bands often found with Landsat TM data. If the image contains a spatial reference and six bands, then the NDVI algorithm will be implemented. If not, `else` statements will be written to report to the Python Shell that the image either does not have a spatial reference or does not have six bands.

```
if spat_ref <> 'Unknown':

    if num_bands == 6:

        # Process Normalized Difference Vegetation Index (NDVI)

        # (Raster Band 4 - Raster Band 3) / (Raster Band
        # 4 + Raster Band 3)

        print 'Starting NDVI...'

        NDVI = Float(Raster(band4) - Raster(band3)) / \
            Float(Raster(band4) + Raster(band3))

        print 'Saving ' + NDVI_image

        if arcpy.Exists(NDVI_image):
            arcpy.Delete_management(NDVI_image)

        NDVI.save(NDVI_image)

    else:

        print 'This image does not have 6 bands, \n'
        + 'the algorithm will not be processed'

    else:

        print in_image + ' does not have a spatial
        reference'

except:
```

10. The specific NDVI algorithm is added within the second `if` block (`if num_bands==6`). The syntax for the algorithm takes advantage of the variables created earlier for each band. This makes the code easier to read and provides less opportunity for the code developer to make typo errors.

NDVI is a common algorithm used with multi-spectral satellite imagery that measures the quantity of healthy biomass (shown below). The algorithm can be found in most remote sensing and digital image processing texts. NIR refers to the near infrared wavelength collected by the Landsat satellite sensor.

$$(\text{Landsat NIR Band 4} - \text{Landsat RED Band 3}) / (\text{Landsat NIR Band 4} + \text{Landsat RED Band 3})$$

11. The `save` line saves the NDVI result to a new image (using the variable defined above (`NDVI_image`)).
12. Two `else` statements are written to let the user know that 1) the image does not have six bands and will not be processed and 2) the image may not have as spatial reference associated with it.
13. The `except` block is added. The exception code is the same code that has already been used in many of the scripts so far.

Demo 7a covered a number of Python and `arcpy` topics in addition to the illustration of processing raster (image) data and using the Spatial Analyst module and extension. The reader is encourage to explore additional topics with the Spatial Analyst module and geoprocessing functionality

Demo 7b: List and Use Feature Classes and Images

This script implements two of the listing routines found in ArcGIS, `ListFeatureClasses` and `ListRasters`. The first List routine will be used to convert shapefiles to file geodatabase feature classes. The second list will check the organization of images stored in a folder. If the image contains six bands, then the `IsoClusterUnsupervisedClassification Spatial Analyst` routine will be implemented and store the output image in a file geodatabase. See **Demo7b.py** commentary for more details on each of these constructs. Feature classes and images are processed from the `\Chapter07\Data` folder and the pre-existing output file geodatabase (**Ch07_fgdb.gdb**) used for the output feature classes and images are stored in the `\Chapter07\MyData` folder. The code has been formatted to fit the page. See the **Demo7b.py** script for the proper formatting.

1. Import the proper modules. The `os` and `sa` modules are added because they are required in the script.

```
import arcpy, os, sys, traceback
from arcpy.sa import *
```

2. Set up a workspace for the input path and variables for the pre-existing output file geodatabase and output image.

```
arcpy.env.workspace = "c:\\PythonPrimer\\Chapter07\\Data\\"

fgdb_path = "c:\\PythonPrimer\\Chapter07\\Mydata\\"

fgdb = os.path.join(fgdb_path, "Ch07_fgdb.gdb")
outUnsuperImg = os.path.join(fgdb, "outUnsuper")
```

3. Set up the `try` block and create the list of feature classes using the `ListFeatureClasses` routine to process only shapefiles in the folder by using the `"*.shp"` wildcard parameter.

4. Set up a `for` loop to process each of the feature classes in the list.

```
try:

    fcclist = arcpy.ListFeatureClasses("*.shp")

    for fc in fcclist:
```

The list above creates a Python list of shapefiles that contain the *“.shp”* extension. Since the output feature classes will be named the same as the *“root name”* of the shapefiles, only the characters before the *“.shp”* are needed. The Python `split` routine can separate the root name from the extension and can be used for further processing. The following is shown to illustrate this. Note this portion of the script is part of the `for` loop created above.

```
print str(fc.split(".")[0]) + " " + str(fc.split(".")[1])

print fc.split(".")[0] # split the file root name and
                      # extension ("shp")
                      # at the dot "."

shaperoot = fc.split(".")[0] # assign a variable for
                           # cleaner code
```

Although the two print statements are not required in the script, they show the use of the `split` routine. When the loop processes each shapefile in the Python list created above, the `split` routine breaks apart the *“root name”* from the *“.shp”* extension by using the *“.”* (dot) character and creates a Python list containing the root name and the extension as separate items in the list. To access the *“root name,”* a Python index is used. Remember that Python lists start their indexes at *“0.”* To access the *“root name,”* `fc.split(".") [0]` is used since the root name is the first element in the Python list created after the `split` routine is implemented. The first print statement above prints out the two separate Python list components that result from the `split` routine. The second only prints out the *“root name”* so that the reader can see this more clearly.

Finally, the `shaperoot` variable is assigned the *“root name”* of the shapefile and is used later on in the script. Assigning the `split` routine results to a variable makes for cleaner code development and less prone to typo errors.

5. Create an `if` statement to check to see if the feature class exists in the file geodatabase. The `os.path.join` routine is used to join the pre-existing output file geodatabase path and the `shaperoot` variable in the `Exists`, `Delete`, and the `CopyFeatures` routine used in the next step. The code developer can take advantage of the variables created above plus the Python `os` module to build and use data paths in different parts of the code. Note the code has been formatted to fit the page. See the **Demo7b.py** script for the proper code format. This code occurs within the `for` loop block created above.

```
shaperoot = fc.split(".")[0] # assign a variable for
                             # cleaner code

if arcpy.Exists(os.path.join(fgdb, shaperoot)):

    arcpy.Delete_management(os.path.join(fgdb, shaperoot))

    print "Deleted " + os.path.join(fgdb, shaperoot)

print 'Copying ' + fc + ' to ' + os.path.join(fgdb, shaperoot)

arcpy.CopyFeatures_management(fc, os.path.join(fgdb, shaperoot))
```

The code snippet above checks to see if the output feature class exists in the existing file geodatabase. If it does, then the feature class is deleted, otherwise, the `CopyFeatures` routine is used to copy the shapefile feature class to the output file geodatabase as a feature class.

6. The second part of the script is to process file-based images stored in a folder. The code is indented to match that of the `for` loop created above. A list of images is created using the `ListRasters` routine. The number of images in the folder is printed to the Python Shell. Note the syntax has been formatted to fit the page. Consult the **Demo7b.py** script for the proper syntax structure.

```
imglist = arcpy.ListRasters()

print "Number of images in workspace: " + str(len(imglist))
```

7. Another `for` loop is created to process the list of images. The `Describe` routine is used to access some of the image properties which will be used later in the code. Some print statements are written to print some of the properties to the Python Shell.

```
for img in imglist:

    desc_img = arcpy.Describe(img)

    print "Image Name: " + desc_img.name
    print "Image Format: " + desc_img.format
    print "Number of bands: " + str(desc_img.bandCount)
```

8. Next, an `if` statement is written to check if the number of bands is greater than 1. If it is then use a `for` loop to iterate through each band of the image, access some properties about the specific band, and print these out to the Python Shell.

```
if desc_img.bandCount > 1:

    for band in range (1, desc_img.bandCount + 1):

        band_name =
        arcpy.Describe(os.path.join(desc_img.name,
        "layer_" + str(band)))

        print "Band " + str(band) + " cols: " +
        str(band_name.width) + \
        " rows: " + str(band_name.height)
```

The band count from the `Describe` routine is used to construct the `for` loop. Notice that the `in range (1, desc_img.bandCount + 1)` is used. This syntax tells Python to assign the variable `band` with the value of 1. For each iteration of the `for` loop, the number automatically increments by 1. The `bandCount + 1` is required so that `band` can be assigned a value up to the total number of bands in an image. To do so, the number of bands (`bandCount`) must be incremented by 1. If the `bandCount` was not incremented by one, then the loop would only iterate for values of one to one less than the number of image bands, which is not desirable. A simple Python snippet can be written at the Python Shell to illustrate this. Write the script multiple times by including and excluding the `" + 1"` after `bandCount` to see the differences.

```
bandCount = 6
for band in range (1, bandCount + 1):
    print band
```

9. An `if` statement is written after the code above to check to see if the image contains 6 bands. If it does, then check out the Spatial Analyst extension and run the `IsoClusterUnsupervisedClassification` routine. This is an automated image processing routine that segments a multi-band image into specific categories that have similar spectral characteristics.

```
if desc_img.bandCount == 6:

    num_classes = 10 # the number of classes to create

    arcpy.CheckOutExtension("Spatial")

    print "Running Unsupervised Classification..."

    UnSuperClass =
    IsoClusterUnsupervisedClassification(desc_img.name,
    num_classes)

    if arcpy.Exists(outUnsuperImg):

        arcpy.Delete_management(outUnsuperImg)

    UnSuperClass.save(outUnsuperImg)
```

The `IsoClusterUnsupervisedClassification` routine is implemented and then the output image is written to the file geodatabase. Remember a variable was created at the top of the script to store the output image for this routine.

```
outUnsuperImg = os.path.join(fgdb, "outUnsuper")
```

An `else` statement is set up to just print the rows and columns of the image if it contains only one band. The `else` statement has the same indentation level as the `if desc_img.bandCount > 1:` line.

```
else:

    print "cols " + str(desc_img.width)
    print "rows " + str(desc_img.height)
```

The standard exception block is added to complete the script. See **Demo7b.py** for the complete script.

Exercise 7 - Batch Clip Images Using a Feature Class

In this exercise both the `Describe` and `List` routines will be used to “batch clip” a number of images (rasters) using the Spatial Analyst extension using the `ExtractByMask` routine. Refer to the demos to see how the Spatial Analyst `arcpy` module is imported and how the Spatial Analyst extension is checked out. Read through all of the instructions before starting the exercise. The **Exercise7.py** script is provided to start the script development.

NOTES

When working on this exercise, the following should be noted.

1. Make sure to use feature layers where necessary.
2. The output path name is likely needed for the Save routine to save the clipped image.
3. Use print statements to help troubleshoot and check query syntax. Use ArcMap to help provide clues to query syntax and that the proper polygon is selected.
4. Make sure to use `Exists` and `Delete` properly to help remove pre-existing feature layers and file names.
5. Make sure one step works before working on another when developing this script.

The following files are required for this example and can be found in **\PythonPrimer\Chapter07\Data\images**. Refer to the **Exercise7.mxd** map document. The workspace will reference the **\PythonPrimer\Chapter07\Data\images** folder.

The images to clip are:

RectifyAA14.tif
RectifyAA18.tif
RectifyBBB19.tif
RectifyQ14.tif

SID_grid.shp – polygon shapefile of grid tiles that will be referenced and queried from within the program. The shapefile is located in the workspace identified above.

The output to this program will be placed in a folder like:

\PythonPrimer\Chapter07\MyData

Each resulting image will have the format (TIF files) with the word “Clipped” at the beginning of each file name:

ClippedAA14.tif
ClippedAA18.tif
ClippedBBB19.tif
ClippedQ14.tif

The following Python syntax will be useful when creating queries, intermediates, and the final output format. “tif” is a variable that represents an image on disk.

```
tif_root = tif.replace('Rectify', '').split('.')[0]
```

The above Python syntax replaces the characters in ‘Rectify’ and replaces them with “nothing” (i.e. ‘’), then in the same line splits the remaining characters from the file extension (in this case “.tif”) and then assigns the remaining characters to the variable `tif_root`. For example, **RectifyAA14.tif** becomes `tif_root = 'AA14'`.

The code developer can try this out at the Python Shell.

```
tif = 'RectifyAA14.tif'
tif_root = tif.replace('Rectify', '').split('.')[0]
print tif_root
```

and can also try this with a Python list (formatting has been modified to fit the page):

```
imglist = ['RectifyAA14.tif', 'RectifyAA18.tif',
           'RectifyBBB19.tif', 'RectifyQ14.tif']

for tif in imglist:
    tif_root = tif.replace('Rectify', '').split('.')[0]
    print tif_root
```

For each of the image files, the result of the above syntax should show a similar result:

AA14
AA18
BBB19
Q14

The script will need to perform the following.

STEP 1

Define a variable to point to an output data path (folder). For example, 'C:\\PythonPrimer\\Chapter07\\MyData\\'.

STEP 2

Make a list of the rasters that only include the TIF type. The conditions will be: '**Rectify***' and be type '**.TIF**' in the `ListRasters` routine. **NOTE:** In ArcGIS Help it uses the key format word '**TIFF**'. The `ListRasters` routine needs to use the word '**TIF**'.

STEP 3

Use a loop to iterate through each element of the raster list to perform the following:

Describe the image and print the following to the Python Shell.

1. Spatial Reference
2. Number of Bands
3. Image format

STEP 4

Create a variable that uses the syntax above that contains the root name of the images that begin with 'Rectify'. You should use the syntax as it is shown above for this step.

STEP 5

Create a query variable that uses the "SIDSHT_ID" attribute from the **SID_grid.shp** file and uses a variable (e.g. `tif_root`) that will hold the "root name" (e.g. AA14). The `tif_root` variable will be used on the right side of the query. NOTE: Review the values of the SIDSHT_ID field and note that the values in this field contain the "root name" value of the TIF file (but without the "Rectify" or the ".TIF" strings).

For example, a query will need to be designed such that the syntax has the form:

"SIDSHT_ID" = 'AA14'

but without "hard coding" the value 'AA14'. The right side of the equal sign needs to be a variable, yet use the proper syntax. One of the challenges is to make sure the syntax is written correctly for the query. Review the previous chapters and demos if needed.

STEP 6

Use a `SelectLayerByAttribute` routine with the query created above to select a feature (i.e. one of the image tiles (rectangles)) from the **SID_grid.shp** file. This selection will only select a single polygon from the `SID_grid.shp` file, which is desired). Remember the `MakeFeatureLayer` routine is required before the `SelectLayerByAttribute` routine can be properly implemented. The variables `SID_grid_Shape` (feature class) and `SID_Layer` (feature layer) have been defined at the top of the code for use in the step.

STEP 7

Buffer the selected tile (polygon) by **5 feet**. Create a generic `buffer.shp` file name (and can be created at the top of the script). The buffer variable is just a string that will be used in the `Buffer` routine. Since the buffer shapefile is created as the output and will be stored in the workspace, all that is needed is a variable referencing a string (i.e. the string of the shapefile name).

```
buffer_shape = "buffer.shp"
```

A separate buffer polygon will be created for each iteration of the loop structure. Make sure to add the `Exists` and `Delete` routines before performing the `Buffer` routine to make sure the previous buffer is deleted before the `Buffer` routine recreates the buffer polygon.

STEP 8

1. Create a variable that is defined to the full path of the output image (created at the top of the script) and begin with the characters `'Clipped'` and the variable `tif_root` that references the root name of the rectified image created earlier in the script. The output image file should end with the characters `'.tif'`. Use the `os.path.join` routine to create the full path to the output image folder that includes the output TIF file name. The output name should look like this for the output:

ClippedAA14.tif

2. Clip the image with buffered polygon. The step will use the Spatial Analyst `ExtractByMask` routine. This tool provides the ability to clip a raster by a feature class (or another raster image).
3. Save the resulting image from the `ExtractByMask` routine. Refer to the ArcGIS Help for `ExtractByMask` and the Chapter07 demos for syntax regarding saving an image. Make sure to use the `Exists` and `Delete` routines before implementing the `save` routine for the output image.

Extra

Determine what the intermediate file(s) are and clean them up (i.e. delete these files). Make sure to NOT delete the input or output, so be careful.

Build Pyramid Layers for the output images. This was not covered in this chapter, but the reader can research it.

Chapter 7 Questions

1. Reading through the chapter and the ArcGIS Help, what are some of the uses for the `Describe` routine?
2. What are some of the benefits of using the `List` routines? What do these operations allow scripting to accomplish?
3. How is an image band accessed when using Python and ArcGIS?
4. How is an image saved using Python?
5. What is the use of the Python string `split` routine? Give an example.
6. How is the Python `replace` routine used in Exercise 7?

Chapter 8 Custom Error Handling and Creating Log Files

Overview

The reader can see that being able to track and identify errors is very helpful when developing code. So far the demo and exercise programs have relied on a single error block that serves as a default error handler. From a developer or single user's point of view, this may be sufficient; however, if the script will be used by a broader audience, then using more specific error messages may be appropriate. Being able to track and log print messages and errors can be helpful, especially if the scripts are complex or when they are automatically executed (as will be illustrated in *Workbook III*, Chapter 11). Creating log files, custom print statements, and error handlers can assist the programmer as well as the user log, monitor, and track unexpected problems with data or code. Log files and the use of date and time stamps in Python routines can assist in the troubleshooting process.

Custom Error Handlers

The reader has already seen and used the `try:` and `except:` blocks to perform some basic error handling. A number of additional error handling methods exist that can expand the error handling capability of a Python script. Some useful methods are:

- a. Using of multiple `try: except:` blocks
- b. Using nested `try: except:` blocks
- c. Using Python functions for different error messages
- d. Creating Python classes for different error messages

Of these methods *A Python Primer for ArcGIS* focuses on the last method, since the ArcGIS Help documents and sample code tend to use this method and are readily available to the programmer. For a more extensive discussion on exception or error handling, consult a Python text or the **Python.org** website.

The reader has likely witnessed a variety of error messages and codes, some of which are not very intuitive for trouble shooting and can confuse an end user when an error is encountered. A code developer can add additional error handling code and messages that provide more constructive and informative messages.

To keep existing code logic simple, often a single `try:` and `except:` block are used for general or default error handling purposes. Following this same logic, error “classes” can be created that specifically target potential trouble spots in the code. For example, consider the following possibility.

A code developer performs a `SelectLayerByAttribute` routine that uses a query. One potential pitfall of this routine is the query could have the incorrect syntax or may include values that return no selected features. If subsequent code attempts to use these selected features (such as in the `SelectLayerByLocation` routine), the code will process, but will still end up with no features. Some additional error handling can be added to stop the program and provide a useful message to both the code developer (likely for troubleshooting during code development) and for the end user.

Creating and Using an Error Handling Class

A Python “class” is a computer programming construct for creating and using “objects.” An object is a group of code that has a certain functionality that can be called from and used in a script at different places. One can think of a class as a function, however, classes differ from functions in that classes can have properties and methods and have a set of hierarchical organization that are not found in functions. A function might be written to perform a calculation operation. The program may want this routine to execute in different places throughout the program. Since this chapter focuses on error handling classes and functions are not thoroughly discussed, the reader is encouraged to consult the **Python.org** site or text for a more in depth discussion on classes and functions and how they can be created and used for expanding the abilities of a Python script. An example is provided in the Chapter08 folder as well as the **Demo8** script.

To create an error class, the general structure is shown below:

```
class <name of class> (Exception):  
    pass
```

The word `class` is a Python keyword to define a Python class structure. The `<name of class>` is any name the code developer chooses (without spaces). `(Exception)` is another keyword that references a Python class to catch exceptions in code and is placed within parentheses. The `pass` keyword (which is indented on the next line of the `class`) tells Python to “do nothing” and acts as a placeholder. The error classes are always defined at the top of a Python script and just after the import modules so the class can be reference at any point in the Python script. In this case, the name of the class will refer to an exception block such as in this example:

```
import arcpy, sys, traceback

class NoWorkspace(Exception):
    pass

try:

    arcpy.env.workspace =
    'C:\\PythonPrimer\\Chapter08\\xData\\'

    if not arcpy.Exists(arcpy.env.workspace):

        raise NoWorkspace

    print 'Completed Program.'

except NoWorkspace:

    print 'The workspace does not exist. Check the
    workspace location.'

except:
    # Normal exception code goes here.
```

In this relatively simple example, a workspace is defined. A conditional statement is set up to test to see if the workspace does NOT exist. If the workspace does not exist, then an exception is raised using the `raise` keyword. In this case, a class is used to define a specific class for exceptions. Since the `pass` keyword is used (i.e. no real functionality occurs), the `except NoWorkspace:` block is called and processes the print statement. In the code above, the path stated above does not exist (note `xData` is not an existing folder) and so the code results in an error.

In addition to the above, also note that the `except:` block is used as well, which serves as a “catch all” for other programming errors.

Writing a custom error handler class helps the code developer to write more specific error messages, the error class can be re-used throughout the code, and additional print statements are not required within the main block of code to provide messages back to the Python prompt when the code fails for the specific checks (e.g. an existing workspace or data path). So, in this example, if the code developer decides to change workspaces, the same kind of check can be performed and call the same error handler.

When developing code, error handlers are often written as a “second phase” of code development. Not until the initial code is written and tested do code developers realize places

in the code for error handling. Also the type and quantity of error handlers depends on the end user and audience of the script. A script written and used primarily by the code developer may not warrant specific error handlers. If a script will be deployed so that others can use it, then more specific error handlers will likely be useful.

Using Log Files to Collect Messages

Another useful option for code developers is to print messages to a location other than the Python Shell. If the code developer manually runs a script at a specific time and wants to monitor the messages reporting back from a script, then printing messages to the Python Shell is sufficient. However, most of the methods described in *A Python Primer for ArcGIS Workbook* series can be automatically run (i.e. without human involvement) and so having a method other than printing messages to the Python Shell is often useful. Printing messages to a file (i.e. a file logging messages and hence the term “log file”) is useful from an implementation point of view because the log files can be created during the implementation of the script and can be kept in a repository (i.e. a folder on computer disk) so that they can be audited at any time. Creating a log file is straight forward. Writing messages to the log file only requires a small change in the print syntax. A couple of example scripts are provided in the **Chapter08** folder.

Creating and Using a Log File

To create a log file for writing messages to it, the following is required.

- a. Create a variable for the log file and its path
- b. Open the log file for writing
- c. Close the log file after writing the messages to it

The following script shows the fundamental elements of creating and using the log file.

```
import arcpy, sys, traceback, time, datetime

CURDATE = datetime.date.today()

# The following location (path and folder) must pre-exist on the disk
logpath= 'c:\\PythonPrimer\\Chapter08\\logfiles\\'

# use just the name log.txt for logging messages
logfile1 = logpath + 'log.txt'

# check to see if the log.txt file exists
if arcpy.Exists(logfile1):
    arcpy.Delete_management(logfile1)

log1 = open(logfile1, 'a')

# prints message to Python Shell
print 'Start logging for log.txt...'

# prints message to the log.txt file
print >> log1, 'Start logging for log.txt...'

# Close the log files
print 'Close the logfile'
log1.close()
```

The above script shows how to create the log file and write to it. Since this book focuses on using Python with ArcGIS the `Exists` and `Delete` functions are used to check whether the log file exists and if it does, deletes it. Alternatively, Python only methods are available to perform file management. The reader can check a Python text or the **Python.org** site for more details.

The first two lines of code set up the path location for the log and then it is used to create the log file. The next line (`log1 = open(logfile1, 'a')`) actually opens the file for writing (`'a'`, to append) meaning that the messages created during the current run will be added to a single file.

NOTE: If the `Exists` and `Delete` routine are used to delete the log file before it is re-created, all of the previous messages will be deleted. The code developer will want to consider how log files will be maintained. A naming strategy may need to be developed to keep track of print messages for multiple executions of the script (such as how many times per day or how many log messages are written to a single file over a number of days). The date and time stamp Python methods mentioned below can assist with developing this strategy.

A variable (e.g. `log1`) is typically used to point to the open file so that print messages can actually be sent to it. Once the file is opened any print statement can be sent to it until the file is closed. To actually write messages to the script notice that the print statement uses `'>>'` to “redirect” the print statements to the log file using the following syntax. The `log1` variable and the message are separated by a comma (`,`).

```
print >> log1, 'Start logging for log.txt...'
```

In the above script, the message contains two print statements that are almost identical. The first statement prints to the Python Shell, whereas, the second statement is written to the log file.

Once the script is finished writing messages to the log file, it can be closed by using the following syntax. Note that `log1` is the variable that points to the actual log file.

```
log1.close()
```

Adding the Date or Time to a File Name or Message

An often useful addition to a log file or message is providing the date and time. To monitor scripting progress, the code developer may want to know the date and time a process starts or completes. To use the `date` and `time` Python functions the `time` and/or `datetime` modules need to be imported. The date and timestamps in print statements can help code developers troubleshoot issues when the scripts are scheduled to automatically run (for example, during off-peak hours). The date can be used as part of the log file name so that unique log files can be generated for a given day.

To access the current date of the computer system, the following syntax can be used.

```
CURDATE = datetime.date.today() # format YYYY-MM-DD
```

The reader can refer to a Python text or the **Python.org** site for more date formatting options.

To obtain the current date and time and use it as a string (for example as a time stamp in a print statement), the following syntax can be used.

```
# date and time format for '%c' is: MM/DD/YY HH:MM:SS
time.strftime('%c')
```

By slightly modifying the print statements in script above, the following syntax takes advantage of the date and time.

```
# prints message to Python Shell

print 'Start logging for ' + 'log' + str(CURDATE) + '.txt'

# prints message with a date and time stamp to the log file # and also
uses the current date in the log file name

print >> log1, time.strftime('%c') + ' Start logging for ' + 'log' +
str(CURDATE) + '.txt'
```

Summary

This brief chapter provided some basic Python structures and strategies to work with errors and keep track of print messages so that the programmer and end user of the script have more meaningful ways of troubleshooting code, data, and logic problems. The demo and exercise provide some additional opportunities for the reader to develop these strategies.

Chapter 8 Demo Create Custom Error Messages

This demo will add a couple of error handling messages to the **Demo5b.py** script used in **Demo5b**. A copy of this script can be found in the `\PythonPrimer\Chapter08` folder. Two kinds of errors will be created: 1) check will be to see if any features have been selected for the `SelectLayerByAttribute` routine and 2) check to see if any features are selected from the `SelectLayerByLocation` routine. If either case exists, then an empty feature class will be generated, which is not desirable and will cause the program to stop. This demo will illustrate the use of creating error “classes”. A number of different options exist with Python; however, ArcGIS preferentially uses the class method. See comments within the demo code for options to test the error messages. The **Demo8.py** script contains the code from the **Demo5b.py** script including the custom error handling code used in this illustration.

The concepts illustrated in the chapter and demos are:

ArcGIS Concepts

- GetCount
- Spatial Analyst extension
- CheckOutExtension

Python Concepts

- `class`
- `except`
- Unique error handling
- Cast numbers and values to strings
- `if` statements
- Open, close, write to files
- `time` and `datetime` modules and formatting
- Redirect print statements to log files

1. Start by defining an error class. Two will be defined as shown below.

```
import arcpy, sys, traceback

class NoAttributeFeatures(Exception):
    pass

class NoLocationFeatures(Exception):
    pass
```

2. Next, toward the bottom of the code, add the specific exception blocks. In this case there are two, since two classes are defined above.

```
print "Completed Script"

except NoAttributeFeatures:
    print 'There are no selected features by attribute.\n'\
        'Review the query syntax.'

except NoLocationFeatures:
    print 'There are no selected features by location.\n'\
        'Review the feature layers and the selection\
        type.'

except:

    # standard error handling code goes here
```

3. Within the body of the code, add an `if` statement to check to see if the selected feature count is zero. If it is, then “raise” an error using the name of the specific error message. In this case, since the number of selected features is required, the `GetCount` routine is used and will also be used in the `if` statement. Also note that the result value from the `GetCount` is cast as a string (using `str()`) to work properly with the `if` statement.

```
# 3. Select by attributes using the query

# Select features by attribute using query
arcpy.SelectLayerByAttribute_management(street_layer,
"NEW_SELECTION", query)

result = arcpy.GetCount_management(street_layer)

if str(result) == '0':

    raise NoAttributeFeatures
```

4. A second `if` statement will be used to check to see that some polygon features are selected.

```
# 4b. Select Parcels

result1 = arcpy.GetCount_management(parcel_layer)

print 'Total number of parcels: ' + str(result1)

arcpy.SelectLayerByLocation_management(parcel_layer, "INTERSECT",
street_layer, search_distance, "NEW_SELECTION")

# this line is added to "clear" the selected features
# and may not actually occur in a real script
# comment out this line to test the script without
# producing the NoLocationFeatures error

arcpy.SelectLayerByAttribute_management(parcel_layer,
"CLEAR_SELECTION")

# obtain the number of records after the polygon
# selection is cleared
# i.e. "no" polygon features are selected

result2 = arcpy.GetCount_management(parcel_layer)

print 'Total number of parcels after selection: ' + str(result2)

# test to see if the count of features is the same
# both before and after the selection routines; if
# they are, raise the error message indicating
# that "no" polygon features are selected

if str(result1) == str(result2):

    raise NoLocationFeatures
```

To test the first error message, deliberately change the query string to a street “CLASS” that is not available (e.g. “CLASS” = ‘X’). After making this change, execute the script to see that the first error message is written to the Python Shell.

To test the second error message, the reader will note that a `SelectLayerByAttribute` routine with the `"CLEAR_SELECTION"` selection type is provided so that the `NoLocationFeatures` error message can be tested. If all of the polygon features are "cleared" from the selection, then the `GetCount` result for `result2` will report the total number of features in the feature class, thus indicating that "no" polygons are selected.

NOTE: If no features are selected, `GetCount` reports the total number of features of the feature class even though there are no selected features.

A test (refer to the second `if` statement shown after the `result2` variable) can be performed to see if the total number of parcels is the same both before and after the selection routines. The `result1` variable reports the total number of parcels in the feature class. If they are the same, this will indicate that "no" polygon features are selected, thus "raising" the `NoLocationFeatures` error. Refer to the **Demo8.py** script to review and test the entire script.

The reader can comment out the line containing:

```
arcpy.SelectLayerByAttribute_management(parcel_layer,  
    "CLEAR_SELECTION")
```

This will bypass the `NoLocationFeatures` error and execute the `CopyFeatures` routine for the polygon layer.

Exercise 8 - Create Custom Error Messages and Log Files for a Script

Using the code developed for Exercise 7, add the following options to the code. NOTE: The input paths for the image data will continue to point to `\PythonPrimer\Chapter07\Data` and the output will continue to point to `\PythonPrimer\Chapter07\MyData`. The `logfiles` folder will be added to the `\MyData` folder. Use the `os.path.join` to create the path (for example, `c:\pythonprimer\chapter07\MyData\logfiles\`). Make sure the path ends with a trailing “backslash.” Your path may be different depending on where the data is stored on your system. See the `Exercise7_Batch_Clip_Solution.py` script which can be found in the `\Chapter08` folder that can be used in this exercise.

1. Add a unique error message handler to monitor an error when the number of selected features is zero.
2. Add a second unique error message handler to monitor an error when the image does not have a spatial reference (i.e. spatial reference is `'Unknown'`). `'Unknown'` is the string name used to indicate that a spatial reference does not exist with a data set.
3. Create a folder under **MyData** called **logfiles**.
4. Create a unique log file that uses the date in the log file name.
5. Write `print` statements for some of the steps to the log file as well as to the Python Shell. For example, if you have a print statement that indicates a process is starting or finishing, write this same print statement to the log file. Use the time stamp string method shown in the chapter. You may want to add a variable for the current date and time. See the chapter and the demo. For example, if the print statement is already created, add additional statements that will be redirected to the log file (so that one message is written to the Python Shell and a second message is written to the log file).

NOTES

1. Make sure to import the `time` and `datetime` modules to the script, otherwise the date and time values cannot be used properly. Also make sure to cast the dates and/or times appropriately. See the chapter for more details.
2. Make sure to add all of the required code pieces for creating error handlers.

Chapter 8 Questions

1. What are reasons why code developers would want to create unique error messages?
2. What are log files useful for?
3. What is the benefit of using date and time stamps in log files and within `print` statements?

Chapter 9 Mapping Module

Creating maps and map books can be a time consuming task for a GIS analyst. Until ArcGIS 10 many analysts created a map document by adding specific map elements such as a title, legend, North arrow, and scale bar among others. As any cartographer can attest, creating a map takes skill in using the software to perform the tasks, but also needs an “artistic eye” to produce a map that is appealing to the audience and communicates the intended message. For some cases, spending significant time creating and designing a map for a single geographic area may be warranted. However, if the same kind of map is required for different geographic locations where only the surrounding map elements change (such as the title, subtitle, legend items, date, map sheet number, etc.), making these minor changes can be very time consuming and often involve the analyst creating many separate map documents or manually changing the map elements for a single map document, and then printing or generating individual PDF documents.

Past versions of ArcGIS provided some tools and sample code to assist with the map production efforts of analysts (such as map production sample scripts and third party tools to semi-auto generate map sheet boundaries and map sheet labeling schemes). Creating useful map production tools required the use of Visual Basic for Applications or Visual Studio to create graphical user interfaces as well as special geographic data for making the map production process worthwhile.

Beginning with ArcGIS 10, a new Python module, the `mapping` module, is available to customize map documents as well as provide the ability to “automate” map production. Using Python scripting reduces the coding overhead (i.e. the number of lines of code) often found with Visual Basic or Visual Studio. In addition, the `mapping` module also contains methods to generate PDF documents or print to available printers. The introduction of the `mapping` module is one of the recent major improvements to the ArcGIS functionality. Additional improvements with automating map production are the introduction of *Data Driven Pages* and ArcGIS reference material for building map books. See the ArcGIS Help topics “**Data Driven Pages.**”

The reader may find these resources useful when developing a map production strategy and some of them require some additional interaction with ArcMap and data preparation. The focus of this chapter is to illustrate how to use the fundamental `arcpy mapping` module methods, properties, and routines to modify some of the map elements that may be present on a map (or map book) since the GIS analyst may need to write custom code to develop automated map sets that are not available in Data Driven Pages.

Overview

Two “views” of geographic data exist in ArcGIS: 1) Data View and 2) Layout View. In Data View, geographic data is viewed, symbolized, and managed without respect to a specific map page size or layout. Data View is typically used when an analyst needs to conduct geographic analysis, review results, and perform other analytical tasks on the data. In Layout View, the geographic data is presented on a map page with a specific size and layout. The Layout View provides the cartographer a (*What You See is What You Get, WYSIWIG*) view of the data to facilitate map presentation. The cartographer or analyst will spend time in the Layout View refining the map elements, symbolizing the map with colors, line types, shading as well as the other map elements such a font size, font type, and placement of the elements (on the side, at the bottom, within the map area itself, etc.).

Map Elements

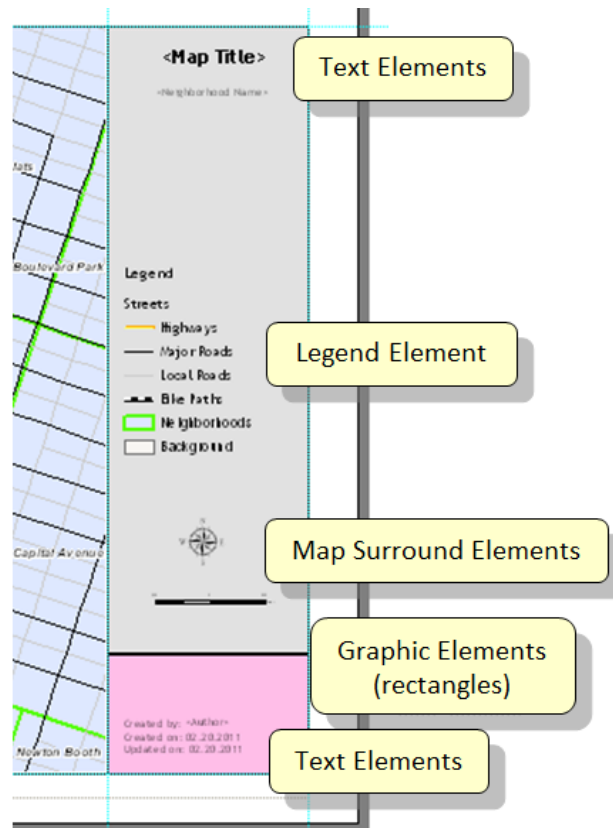
A typical ArcGIS map consists of the following basic components.

- a. *Map document* – the specific map file
- b. *Map frame* – the area that contains geographic data
- c. *Layers* – data layers found in the table of contents
- d. *Legend* – usually a list of data layers and their symbols shown in the map frame
- e. *Title* – the theme of the map
- f. *Subtitle* – (sometimes optional, depending on the map)
- g. *North arrow* – to indicate which direction is North in the map
- h. *Scale bar* – scale to indicate the level of detail and distance measurements
- i. *Title block* – an area where a cartographer can place information about the map, such as an author, map date, map sheet numbers, disclaimers, map data sources, notes, or other documentation

The following figure shows a typical map with the elements listed above.



The figure below shows most of the layout elements above and their associated element type.



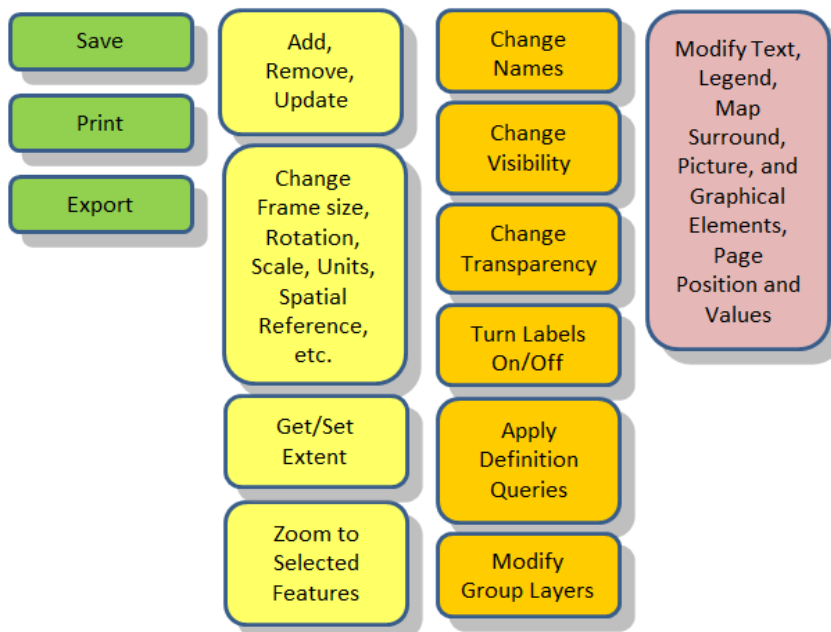
Map Element Relationships

The general relationship among the elements of a map is shown below. The chart indicates some of the fundamental map layout elements and their associated operations. For a full account of the map layout elements available in the mapping module, see the ArcGIS Help documents under **Mapping module**.

Map Element



Operations



The primary element is the ArcMap document. Map documents contain data frames and layout elements and can also be saved, printed, and exported. Layers are typically related to a data frame and can be added, removed, and modified (for example, the symbology can be changed). In addition, data frames can be modified (such as the rotation, scale, units, and dimensions of the map frame on the map layout). The extent of the map frame can be

obtained or set explicitly or changed by zooming to selected features. Specific layers can have their appearance changed (such as the visibility, transparency, and labeling) and definition queries can be applied to the map layout. Finally, the layout elements (such as text, graphics, pictures, and legends) can have their position and values set. *A Python Primer for ArcGIS* covers some of the fundamental `mapping` module routines that are used in many map production processes. The ArcGIS Help contains a more comprehensive list of properties and methods associated with the `mapping` module.

Prerequisites

Before working with the `mapping` module and manipulating it programmatically, a map document must exist on computer disk and contain at least one map frame. Typically, this map frame will have some data layers and typically include some map elements that are likely to change with each unique map produced. Essentially, this map document can be thought of as a template for an automated map production process. The elements that will change (such as the map title, subtitle, date, legend, North arrow, scale, and various text elements) will also need to be uniquely named through their respective element properties.

Map Design

1. Create a map template (.mxd)
 - a. Determine map size
 - b. Add data frame(s)
 - c. Set up and symbolize Layers
2. Add and name map elements (as required)
 - a. Title/sub title
 - b. North arrow, scale bar, legend, title block

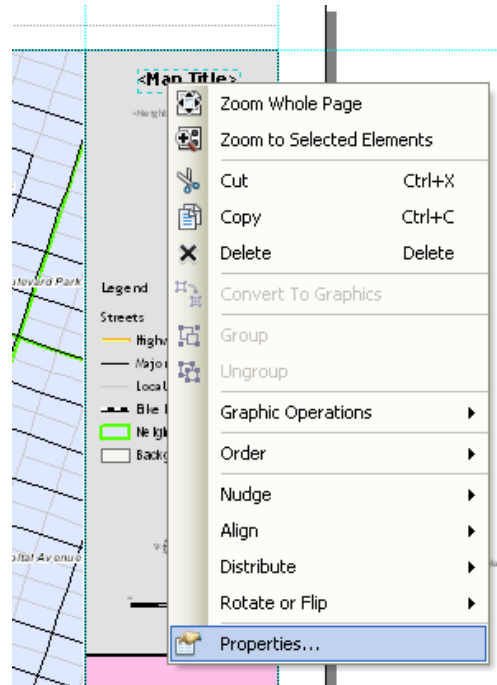
Create a Map Template

Fundamentally, before working programmatically with a specific map document (and learning the programming fundamentals), it is helpful to create a “template” that contains all (or most of) the data layers required for the map, the page layout (map size and orientation), the map frame, and the organized map layout. This “template” is not the traditional ArcMap template (.mxt), but rather a template (.mxd) that contains a number of common map elements that can be manipulated through scripting methods as part of an automated map production task. Only layers can be programmatically added to the map document, so essentially all of the map

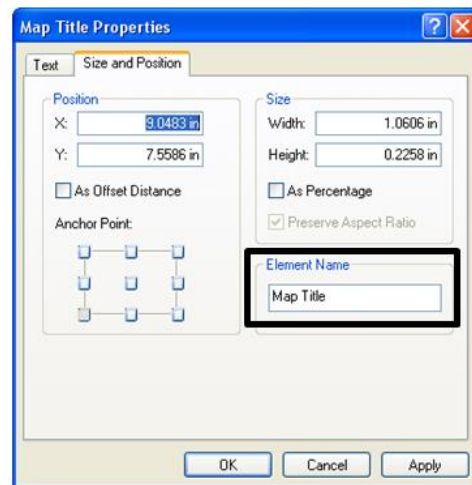
elements exist in the map document that will be programmatically modified. The examples in this chapter will illustrate a number of these programming methods using an existing map document with data layers and typical map layout elements.

Name Map Layout Elements

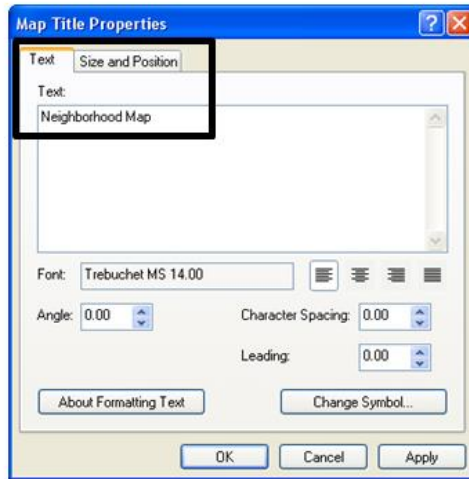
In addition to creating a map template, the cartographer or analyst will need to assign unique “names” to each layout element that is to be manipulated by a Python script. This is performed within ArcMap as part of the original map design. Map layout elements can be named by accessing the element’s properties (right-click on a specific layout element and choose Properties). The following figure shows an example.



Click on the Size and Position tab to access the layout Elements Name property. In this case, the “name” of the selected layout element is called “Map Title.” When the Element Name is changed, the Property dialog box title is also renamed.



Note the Element Name is different than the actual text shown on the map. The text shown on the map is found under the Text tab in element’s properties. Also, depending on the type of layout element, different tabs and properties are available.



The unique name for each layout element is used by the `mapping` module to gain access to specific elements and programmatically manipulate them. If a unique layout element name does not exist, then it cannot be changed through Python scripting methods.

Also, data layers should have meaningful names in the table of contents. If multiple data frames exist on a single map, they should also be uniquely named. Providing names to layers and data frames can assist with developing Python script to make changes to the map layout. NOTE: Accessing specific data layers and data frames does not require unique names, but it is good practice to use unique names.

Mapping Module Class Properties and Methods

Before discussing specific Python methods to manipulate the map layout the reader should note that many of the coding operations related to the ArcMap document is to access and modify properties and methods from specific mapping *classes*. A *class* is a programming term that means a collection of properties, methods, and functions are related to one another. For example a driver may have a car (class) that is a certain make and model, has four wheels, and is white (properties). In addition, the car may have methods that the driver can use to determine the how well the car functions. For example, the driver may have a method called Fuel Gauge that reports the current level of fuel in the car. It may also have an Odometer method to report the total distance the car has driven or a Trip Odometer to report the mileage of the current trip.

From an ArcGIS perspective all of the properties and methods related to each other can be found in their respective classes. Data Frame properties and methods can be found in the Data

Frame class. Data frame properties can include the name, scale, rotation among others. These properties can be used to determine or set the data frame name, current scale, or the data frame rotation. Data Frame methods include actions such as `zoomToSelectedFeatures` that can be implemented to perform an action (in this case, zoom the data frame to the extent of the selected features). Layers and specific layout elements also have their own respective classes that a code developer can use to obtain and manipulate values for each part of a map document. The full list of the `mapping` module classes (which include descriptions of the properties and methods) can be found under the **Mapping Module** Help in ArcGIS.

Mapping Module Functions

In addition to properties and methods, the `mapping` module also has a variety of functions to gain access to different parts of the ArcMap document and its map elements (data frames, layers, layout elements) and for exporting and printing map documents. Mapping module functions are implemented in similar ways to other ArcGIS functions and geoprocessing tools. A `mapping` module function performs an action to do something with a data frame, layer or layout element. A number of list routines exist that can be used to iterate through layers (for example, to change the visibility in the map) or layout elements (for example, to change the value of text elements). The ArcGIS function help is divided between **Managing Documents and Layers** (i.e. those related to the ArcMap document contents) and **Exporting and Printing**.

Implementing Map Documents, Data Frames, Layers, and Layout Elements

Now that some of the map document, data layer, and map layout elements fundamentals have been discussed, this section illustrates how to work with these elements programmatically using Python and the `mapping` module. The examples shown below can be followed in the **Demo 9a Mapping_Module_Overview.py** script and the **Mapping_Module_Overview.mxd**. Refer to specific `mapping` module classes, properties, methods, and functions in the ArcGIS Help as needed.

The general workflow when working with the `mapping` module is:

1. Import the `mapping` module
2. Access a specific map document (.mxd)
3. Access a data frame from a list of data frames (often only one)
4. Access and modify layers within the map (as needed)
5. Access and modify layout elements in the map (as needed)
6. Print, export, or save the map

Import the `mapping` Module

To begin programming Python scripts to work with map documents, the `mapping` module must be imported.

Notice the `'from arcpy.mapping import *'` line below. Using this syntax tells Python that the entire functionality of the `mapping` module will be imported and helps to shorten some of the `mapping` module related syntax (such as not needing to add the syntax `"arcpy.mapping"` before writing each of the `mapping` routines).

```
import arcpy, os, sys, traceback, datetime
from arcpy.mapping import *
```

Accessing an ArcMap Document

Next, the code developer needs to access the pre-existing map document. Notice that a variable is assigned the path (folder, not a workspace) for the location of the map document. To access a specific ArcMap document, a path and map document file name is required, not a workspace).

```
datapath = 'C:\\PythonPrimer\\Chapter09\\'

# output for PDFs

mappath = datapath + 'MyData\\Maps\\'

mxd = MapDocument(datapath + 'Mapping_Module_Overview.mxd')
```

The specific ArcMap document (datapath + Mapping_Module_Overview.mxd) is assigned to the variable mxd.

NOTE: The specific mapping module functionality is accessible by `import arcpy`, however, without using `from arcpy.mapping import *`, all of the mapping module specific classes, functions, and methods would need to include “`arcpy.mapping`” in front of the specific class, function, or method. For the example, without `from arcpy.mapping import *`, the syntax would be written like this for the map document:

```
mxd = arcpy.mapping.MapDocument(datapath +
'Mapping_Module_Overview.mxd')
```

Once the map documented is referenced, some map properties can be obtained. Below, see some of the map properties that are printed to the Python Shell.

```
print 'Map Document Title: ' + str(mxd.title)
print 'Map Document Author: ' + str(mxd.author)
print 'Map Document Summary: ' + str(mxd.summary)
print 'Map Document Description: ' + str(mxd.description)
print 'Map Document last Date Saved: ' + str(mxd.dateSaved)
print 'Is the Map Document Relative Path checked? : ' +
str(mxd.relativePaths)
```

A programmer can use the map properties to make checks on map documents and determine if further changes and modifications are required. For example, a conditional statement can be written to check the last saved date of the map and then make further updates to it if they are needed.

Accessing a Data Frame

Next, a specific data frame can be accessed using the `ListDataFrames` function. The syntax below tells Python to find the data frame named “Layers” from the first data frame in the **Mapping_Module_Overview.mxd**. The `ListDataFrames` function returns a Python list which begins with an index value “0” or the first position in the list. A Python list contains a list of specific elements, in this case data frames. Elements in a Python list are accessed by their position in the list, which begins with position (0). The first (and only element) in this example is the only data frame in the map and thus the syntax is:

```
dataframe = ListDataFrames(mxd, "Layers") [0]
```

To access a specific element in a Python list, its index position must be used. In the example below, since the code developer already knows the contents of the specific map document and that it contains only a single map frame, the map frame Python list position (i.e. 0) can be “hard coded” and assigned to the variable `dataframe`. Like the map document properties shown above, data frame properties can be accessed. Some of them are shown below. The code has been formatted to fit the page. See the **Demo9a_Mapping_Module_Overview.py** script for the proper format.

```
dataframe = ListDataFrames(mxd, "Layers") [0]

# Report some properties about the map frame named
# "Layers"

print 'Map Frame Map Units: ' + str(dataframe.mapUnits)
print 'Map Frame Scale: ' + str(dataframe.scale)

# See ArcGIS Help under 'spatialreference' (all one
# word) for more information

print 'Map Spatial Reference: ' +
str(dataframe.spatialReference.name)

print 'Map Frame Anchor Point X Position (page units): '
+ str(dataframe.elementPositionX)

print 'Map Frame Anchor Point Y Position (page units): '
+ str(dataframe.elementPositionY)

print 'Map Frame Width (page units): ' +
str(dataframe.elementWidth)

print 'Map Frame Height (page units): ' +
str(dataframe.elementHeight)

# Get the extent of the data frame and assign it to a
# variable

mapExtent = dataframe.extent
```

Accessing Layers and Layer Properties

To access the layers in a map, the following syntax can be used:

```
TOCLayers = ListLayers(mxd)
```

where `TOCLayers` is a variable that references the list of layers in the Table of Contents in the map document.

The `ListLayers` function is used to access a list of data layers in the ArcMap document. The resulting list is a Python list and operates similarly to the `ListDataFrames` function. In this case the Python position `[0]` is not used because a `for` loop is implemented to iterate through

all of the layers in the list (that occurs within the map document). Note that this `ListLayers` function is not referencing a specific data frame, so the resulting list will include all of the layers in the table of contents. The `ListLayers` routine does have an optional data frame parameter that can be used to provide a list of layers associated with a specific data frame. The `for` loop iterates through the list to perform additional operations on different layers. The example above shows how a list of layers is created from a map document (assuming that only one data frame exists in the map document).

The above script could also have been written by referencing the data frame parameter.

```
TOCLayers = ListLayers(mxd, '', dataframe)
```

`mxd` – variable for the map document

`''` – optional wild card parameter (not used in this example)

`dataframe` – variable for the data frame

Layer properties are accessed in a similar manner as map document or data frame properties. The general form of accessing a layer's property is:

```
Layer.property
```

Since many layers exist in the table of contents, a `for` loop is used. Within the loop, the layer properties can be accessed, such as for the layer name:

```
# loop through the layers
for TOCLayer in TOCLayers:

    print 'Layer Name: ' + str(TOCLayer.name)

    print 'Longname: ' + str(TOCLayer.longName)
```

See the `Layer` class in the ArcGIS Help under **Mapping module** for a full list of layer properties.

The example above also shows how a layer can be accessed within a group layer using the `longName` property. A group layer in ArcMap often contains a set of layers that may relate to a specific theme or purpose. For example, a “Roads” group layer can contain different road layers such as Highways, Surface Streets, and Trails. Similarly, a “Detail Map” group layer, may contain Streets, a City Boundary, and Parcel layers. The general syntax to access a group layer is:

```
Layer.longName = '<GroupLayer>\<LayerName>'
```

where the `GroupLayer` and `LayerName` are separated by a “\”. In the **Demo9a_Mapping_Module_Overview.py** script, the `Neighborhoods` layer is accessed this way:

```
TOCLayer.longName = 'Detail Map\Neighborhoods'
```

Using Layer Properties in the Script

Typically, once layer properties are retrieved, the code developer wants to do some processing with the values (besides just printing them out to the Python Shell). The use of the property depends on the property values, when the value is obtained and how the coding structure is designed. In the snippet below a conditional statement is used to check to see if the `longName` is the “Detail Map\Neighborhoods” layer. If it is, then the layer’s `transparency` property is set to 50% (i.e. `TOCLayer.transparency = 50`) or if the group name is the “Detail Map\Streets”, then the labels are turned on. Likewise, other properties can be manipulated.

```
if TOCLayer.longName == 'Detail Map\Neighborhoods':  
    TOCLayer.transparency = 50 # 50% transparency  
  
if TOCLayer.longName == 'Detail Map\Streets':  
    TOCLayer.showLabels = True # turn labels on
```

The reader should recognize the progression or hierarchy of accessing layer properties. To access a specific layer’s properties, the map (and optionally the data frame) must be known so that a specific layer can be obtained; only then can layer properties be accessed and used for various purposes. As shown above, the script accesses a specific ArcMap document (`Mapping_Module_Overview.mxd`). Since the map document only contains one data frame, the code developer only needs to use the map document as the parameter in the `ListLayers` routine (i.e. `ListLayers(mxd)`). Once the list is obtained, specific layer properties can be accessed and modified as needed. The same will be true when the code developer wants to manipulate the layout elements of a map, which will be shown below.

Accessing and Changing Layout Elements

To manipulate the layout elements a similar process is used. The layout elements are accessed through the use of the `ListLayoutElements` function. Since several different kinds of layout elements exist, one of the specific layout element types is often used in the `ListLayoutElements` function. The specific element types are:

- TEXT_ELEMENT – text elements in the map
- LEGEND_ELEMENT – a legend, if used in the map
- DATAFRAME_ELEMENT – any data frame in the map
- GRAPHIC_ELEMENT – graphic elements (rectangles, circles, and other shapes)
- MAPSURROUND_ELEMENT – north arrows, scale bars
- PICTURE_ELEMENT – image files (e.g. logos)

The snippet below shows how the text elements (element type is `'TEXT_ELEMENT'`) of a map can be accessed and used.

```
tElements = ListLayoutElements(mxd, "TEXT_ELEMENT")

print 'Processing text elements...'

for tElement in tElements:

    if tElement.name == 'Map Title':

        tElement.text = 'Test Map'
        tElement.elementPositionX = 8.7

    if tElement.name == 'Print Date':

        tElement.text = str(CUR_DATE)
```

Shown above, the `ListLayoutElements` function is used to obtain only the “text” type elements from the map document (`ListLayoutElements(mxd, 'TEXT_ELEMENTS')`). All of these happen to be part of the title block in the ArcMap document (see the **Mapping_Module_Overview.mxd** associated with this chapter). A `for` loop can be used to iterate through many of the text elements and make the required changes using the specific text element properties. Remember that the layout elements use the “Element Name” described above so that Python and the mapping module can access specific layout elements. Only those elements with unique element names can be modified programmatically. Elements that are not expected to be modified programmatically do not need to have a unique element name.

The above script shows that the text element with the element name “*Map Title*” will have its “Text Property” assigned to the string “Test Map.” The text will also be placed at the page layout X position at 8.7 inches from the left edge of the page. Also, the text element with element name “*Print Date*” will have its text property assigned to the current date of the computer system. The script has a variable called `CUR_DATE` that is assigned the current computer system’s data. This variable is defined towards the top of the script. Note in the **Demo9a_Mapping_Module_Overview.py** script that some special formatting is provided so that the date will have the format MM.DD.YYYY.

```
CUR_DATE = datetime.date.today().strftime('%m.%d.%Y')
```

Export Map to PDF or Print Map to Printer

A final step to modifying the map is to export or print the map to a printer or PDF file. Several export options exist for map documents, but only the export to PDF is described here. For more details see the ArcGIS Help under **Exporting and Printing Maps**.

To export or print the map the `ExportToPDF` or the `PrintMap` function is used. The script below shows the two different implementations. The `Print Map` functions are commented out so that the end user can change the settings depending on the local printer or network.

```
# Check to see if PDF exists, if it does, delete it

if arcpy.Exists(mappath + 'test_map.pdf'):
    arcpy.Delete_management(mappath + 'test_map.pdf')

print 'Writing PDF file...'

# Create the PDF document

ExportToPDF(mxd, mappath + 'test_map.pdf')
print 'Created : ' + 'test_map.pdf'

# Alternatively, the map can be printed to a local printer
# This is commented out and can be changed by the code
# developer

# PrintMap(mxd) # prints map to default printer

# print to networked printer
# PrintMap(mxd, '\\\\network_location\\printer_name')
```

Notice that the path of the map (the `mappath` variable) and a file name (`'test_map.pdf'`, “hard coded” in this example) is used in the `ExportToPDF` function to save the map to a PDF file on disk. The map document (the `mxd` variable above) will be exported to the path and file name in the second parameter of the `ExportToPDF` function. Note that the `mappath` variable is defined toward the top of the script. The top portion of the script is shown below.

```
datapath = 'C:\\PythonPrimer\\Chapter09\\'
mappath = datapath + 'MyData\\Maps\\' # output for PDFs
mxd = MapDocument(datapath + 'Mapping_Module_Overview.mxd')
```

Saving Map Documents

ArcMap documents can be saved after changes have been made to them using Python by using either the `save()` or the `saveACopy()`. The `save()` method performs the same operation as **File—Save** in ArcMap, while `saveACopy()` performs the same operation as **File—SaveACopy** in ArcMap. The `save` method will save any changes made with the Python script. The `saveACopy` method will save the changes to a new ArcMap path and file name (defined in the script) and an optional ArcGIS version value (e.g 10.0, 9.3, 9.2, etc). The default is the current version of ArcGIS, but other previous versions can be used in the version parameter. The `save` method may not be desirable, if the changes made using the script are used for map document production where PDF files or hard copy prints are made. By not saving the changes to the map template used in the map production routine, the map remains in its original state. In the case of creating and executing a script for map production, the script makes changes to the map as needed and then uses the export or print routine to create the PDF files or print the maps to a printer, so using the `save` or `saveAsCopy` are really not required. The script below shows the `save` and `saveACopy` methods as being “commented out” so a programmer can decide to use them or not.

```
# Save Changes
# Commented out so that the user can perform this
# operation if desired.

# mxd.save() # performs the same operation as File-
#           # Save in ArcMap

# mxd.saveACopy(<path and file name of different MXD>,'10.0')
```

Working with Data Frame, Layer, and Layout Element Methods

So far this chapter has focused on the fundamental map elements and some of the properties associated with them. In addition to the various *properties* each of the map elements may have, several *methods* are also available for data frames, layers, and layout elements. The section above regarding Saving Map Documents discussed and illustrated two common methods associated with the map document (`save` and `saveACopy`). This section will focus on some of the common methods associated with data frames, layers, and layout elements such as changing the zoom extent.

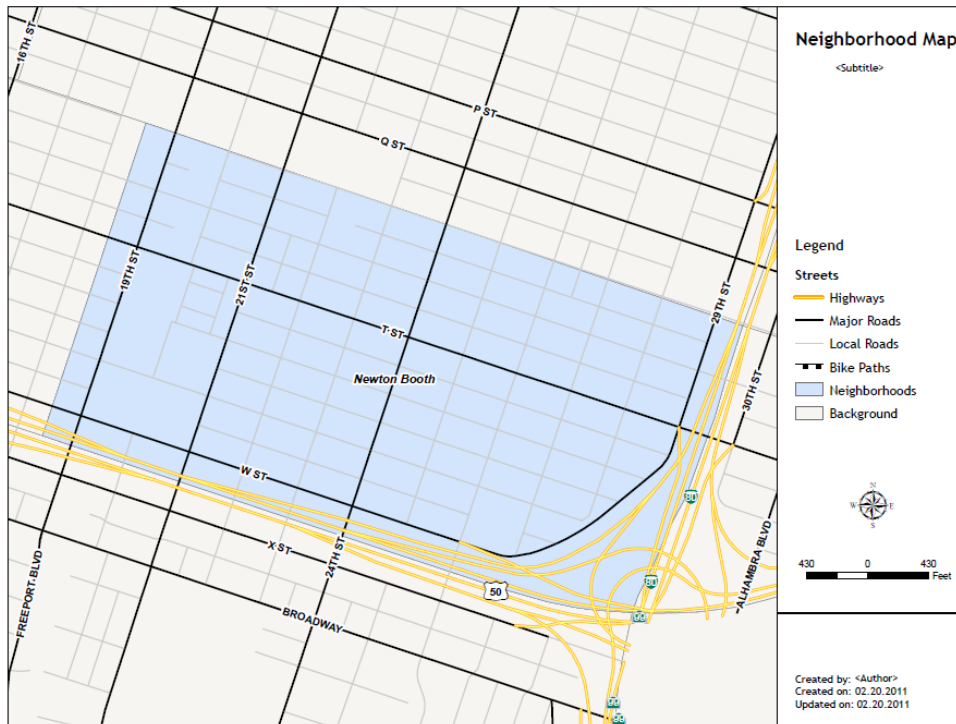
When automating the map production process, changing the map extent of the data frame in Layout View is often required. A number of methods provide this capability and can be implemented in a number of ways depending on the information and level of detail needed in a given map. Changing the map extent involves applying the new extent to the data frame. The map extent can be changed by using a definition query or changing the extent based on selected features from all layers or from a single layer. These methods are described below.

Changing the Map Extent using a Definition Query (or not)

One common method to change the map extent is to use the `definitionQuery` Layer property to limit a specific area or geographic location based on a set of criteria (such as a specific project location or area, like a neighborhood).

```
query = """"NAME" = 'Newton Booth'"""  
TOCLayer.definitionQuery = query
```

This property can be useful to show a focus area where other geographic data provide some context (such as roads). The map below shows a neighborhood shown shaded (e.g. from a neighborhood layer) as a result of a definition query. The neighborhood appears as the focus of the map.



From a code development point of view the following process is used. A looping structure is used to cycle through all of the layers in the map document using a `ListLayers` routine as illustrated above. See the **Accessing Layers** section above. The `TOCLayer` is a variable that points to a specific layer (in this case the neighborhoods layer). A query is created to identify a specific neighborhood ('*Newton Booth*'). This query is applied to the `definitionQuery` property of the layer. The next line uses the `getExtent()` method (a method in the `Layer` class) of the layer and assigns it to the `dataframe.extent` property, which effectively changes the map extent to the extent of the "queried" features in the definition query. The `getExtent()` will get the extent of the feature (or features) that meet the criteria of the definition query. The `dataframe.scale = dataframe.scale * 1.1` applies a ten percent buffer around the extent of the queried feature that prevents the edges of the feature from touching the map frame (i.e. it makes the map a little more pleasing to view). Applying this scale factor is common practice when creating map sets or map books. See the code below.

```

query = """NAME" = 'Newton Booth'"""

# Assign the existing query to the
# definitionQuery property of the layer

TOCLayer.definitionQuery = query

# get the extent of the features (in this case the
# definition queried features
# assign this extent to the data frame extent

dataframe.extent = TOCLayer.getExtent()
dataframe.scale = dataframe.scale * 1.1

```

If a definition query is not used then the `getExtent()` method will return the full extent of the data layer, which the code developer may not desire.

Changing the Map Extent using Selected Features

Oftentimes, zooming to the extent of one or more selected features is required when developing automated processes for creating maps or map books. Automated map sets and books usually have a focus area (such as a project area, neighborhood, map grid, etc.). In addition, in more complex map books, multiple map sheets may be required to show the appropriate level of detail of geographic features, labels, and symbols. To take advantage of the ability to automate the map production workflow, being able to change the map extent to one or more selected features is very useful. ArcGIS using Python provides a couple of methods that help make this possible.

1. `zoomToSelectedFeatures()`
2. `getSelectedExtent()`

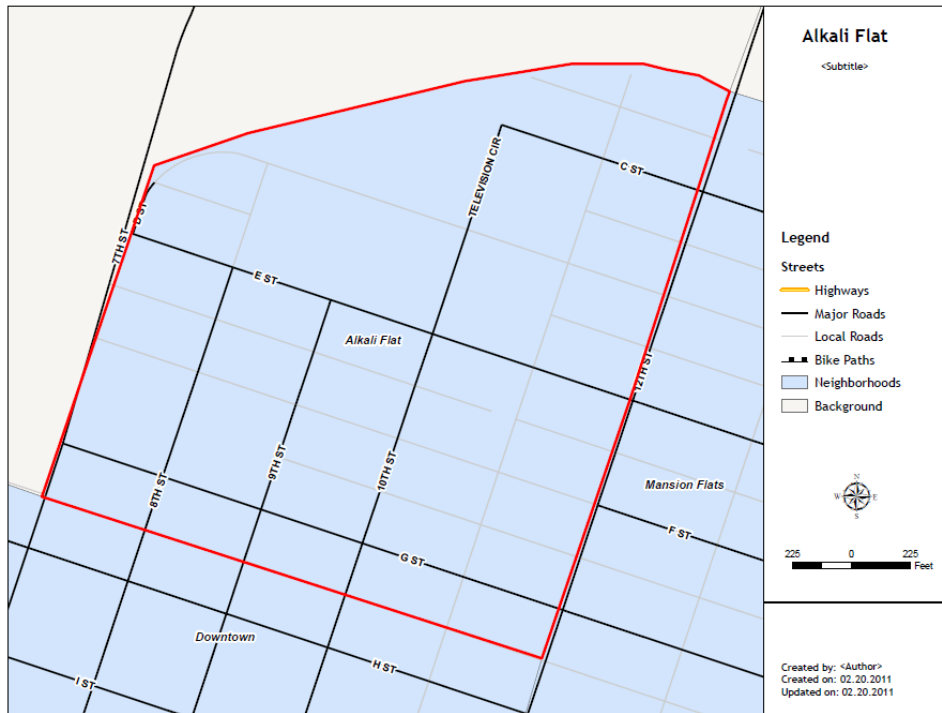
The first method references the data frame class and can change the data frame's extent based on any set of features that are selected in all layers. This functionality mimics the **Selection—Zoom To Selected Features** (for all layers in the data frame) menu option in ArcMap. The second method is in the layer class and can change the data frame's extent based on the selected feature within a specific layer. This functionality mimics the **Selection—Zoom To Select Features** option found under the context menu when the user right-clicks on a layer's name in the Table of Contents (i.e. the list that is often activated when the user needs to look at the layer's properties). Both methods can be used to change the map display and scale of the map to focus on a specific area. Either method is often preceded by a selection method (`SelectLayerByAttribute` or `SelectLayerByLocation`) to create a set of selected features that can then be used to change the display of the map.

The snippet below shows an example where the `zoomToSelectedFeatures()` method is used.

```
if TOCLayer.longName == 'Detail Map\Neighborhoods':  
    query = """"NAME" = 'Alkali Flat'"""  
    arcpy.SelectLayerByAttribute_management(TOCLayer,  
        "NEW_SELECTION", query)  
    result = arcpy.GetCount_management(TOCLayer)  
    print 'Number of selected features: ' + str(result)  
    # zoom to the extent of the selected feature  
    dataframe.zoomToSelectedFeatures()
```

A loop is used to access a list of layers from the map document. The `TOCLayer` variable points to one of these layers (in this case the **Neighborhoods** layer which is part of the **Detail Map** group layer). A query is defined that is used in the `SelectLayerByAttribute` routine. Once the feature is selected, the `zoomToSelectedFeatures()` method is implemented. Notice that it is used with a `dataframe` variable (defined towards the top of the script), since the `zoomToSelectedFeatures()` is a method of the Data Frame class.

When the map is eventually exported to a PDF document, the resulting map shows the highlighted feature.



If other features from different layers were selected, the `zoomToSelectedFeatures()` method would zoom to the extent of all of the selected features.

The `getSelectedExtent()` method works in a similar manner, but is limited to a specific layer's selected features because its method is related to the layer. The snippet below illustrates the `getSelectedExtent()` method. See the **Demo9b_Mapping_Module_Methods.py** script for more details.

```

query = """"NAME" = 'Downtown'"""

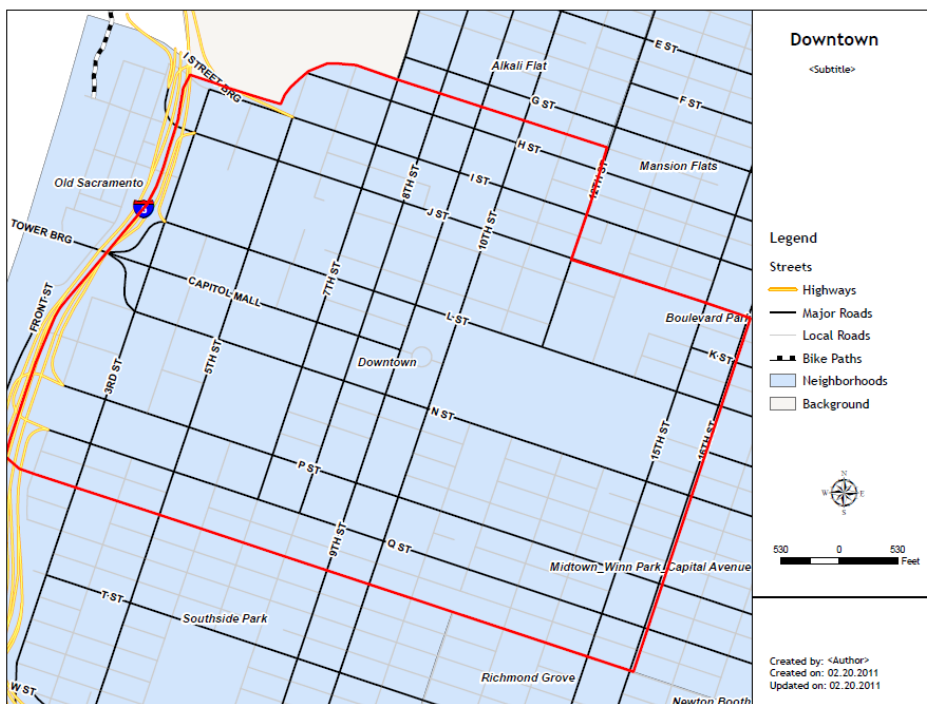
arcpy.SelectLayerByAttribute_management(TOCLayer,
    "NEW_SELECTION", query)

dataframe.extent = TOCLayer.getSelectedExtent()

```

The script, as shown above, uses a query string in a `SelectLayerByAttributes` function to select a specific neighborhood from the neighborhoods layer (the same layer referred to above). After the selected features are obtained, the `getSelectedExtent()` method is used to assign the data frame's extent property to the extent of the selected feature in the layer.

When the map is eventually exported to a PDF document, the extent of the map will represent that of the selected feature (in this case the selected neighborhood). The resulting map will show the feature highlighted.



Notice above that the highlighted feature's edges touch the sides of the data frame. The section of code does not implement the `dataframe.scale * 1.1` as previously shown. If it did, the map would be "zoomed out" slightly, where the feature's edges would not touch the

data frame. If the programmer wanted to “clear” the selected features, the `SelectLayerByAttribute` routine could be used with the `'CLEAR_SELECTION'` selection type before the map is exported or printed.

Adding and Saving Layer Files

The last two methods to discuss in this chapter are adding and saving layer (`.lyr`) files. A programmer may want to dynamically add and save layer files as part of the map production process. For example, a major roads layer may need to be added to certain kinds of map documents. If a programmer wants to “save” an existing layer file to a new name as part of the map production process, this can also be performed.

When datasets are added to the Table of Contents, feature classes automatically become feature layers. The layers in the Table of Contents often have the styling changed as well showing labels and applying definition queries. These layer “conditions” can be saved to a layer file with the (`.lyr`) extension and can be part of current or other map documents. If a programmer wants to add an existing layer to a map programmatically, the `AddLayer` function can be used.

When an ArcGIS user saves a layer file in ArcMap, the analyst typically performs this function by right-clicking on the layer name and choosing **Save As Layer File**. This same function can also be performed programmatically by using the `save()` or `saveACopy()` method. To overwrite an existing layer (`.lyr`) file, the `save()` method is used; to save the layer to a new (`.lyr`) file, the `saveACopy()` method is used. Both of these are related to the layer.

NOTE: From a Python programming point of view only some specific styling changes can be made regarding the “symbolology type” (see ArcGIS Help under **arcpy mapping layer**). Keep in mind that Esri has indicated that they may not implement full symbolology (styling) support for `arcpy` as they do with `ArcObjects`. See Esri forum post (<https://geonet.esri.com/thread/40656>, ca. 2011-2014).

The following Python script snippet illustrates how a layer file can be added to a map document, have one of its properties (`showLabels`) changed, and then be saved to a new layer file. See **Demo9b_Mapping_Module_Methods.py** for additional code and details.

```
# Get the first and only legend
legend = ListLayoutElements(mxd, "LEGEND_ELEMENT") [0]

cityFacilitiesLayer = Layer(datapath + 'data\\city
facilities.lyr')

# set the autoAdd before adding the layer
legend.autoAdd = False

# Add the new layer
AddLayer(dataframe, cityFacilitiesLayer)

# Turn labels on
cityFacilitiesLayer.showLabels = True

# Save the layer file to a new .lyr file name
# The layer saveACopy only works for any layer that is
# a .lyr file. Save and saveACopy cannot be used with # other
# layers in the Table of Contents that do not
# originate from a .lyr file

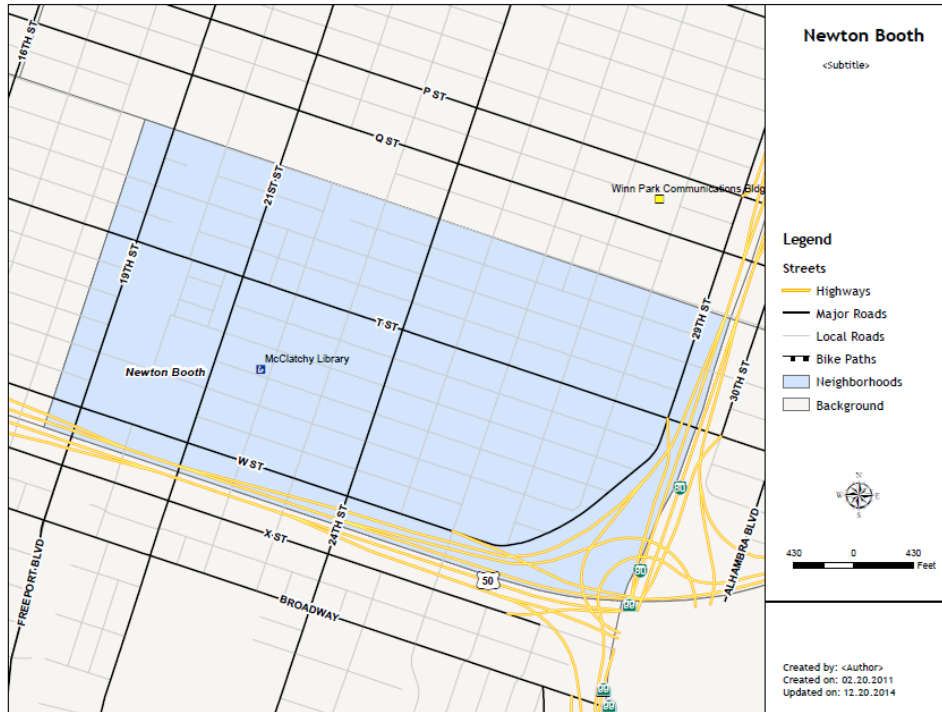
cityFacilitiesLayer.saveACopy(mappath +
'City_Facilities_Labels.lyr')

if arcpy.Exists(mappath + 'City_Facilities_Map7.pdf'):
    arcpy.Delete_management(mappath + 'City_Facilities_Map7.pdf')

ExportToPDF(mxd, mappath + 'City_Facilities_Map7.pdf')
```

This script that shows a variable (`legend`) being assigned to the first (and only) legend item found in the map document from a list of layout elements. The next line locates the `.lyr` file on disk and assigns this to a layer variable (`cityFacilitiesLayer`). The `.lyr` file contains settings such as symbology (styling), label types and fonts, definition queries, etc. and are part of the original layer file. Before the layer is added, the legend's `autoAdd` property is set to `False` so that the new layer is not added to the map's legend. The layer is then added to the map's data frame using the `AddLayer` function. Since this map only has one data frame, (which is defined towards the top of the code, not shown), the layer is added to this data frame. Next, the layer's labels are turned on using the `showLabels` property. Any label settings (fields, fonts, types, etc.) that are contained in the `.lyr` file definition will be displayed for the layer. A new layer file can be created by using the `saveACopy` method and will include any definition queries, label changes, transparency, or brightness changes that were modified programmatically. The current layer definition can be overwritten by using the `save()` method. The map resulting from the changes above is then exported to a PDF file, which looks

like the map below. Note the subtitle and author are not set because these text elements are not used in this section of code.



Creating a Map Book Programmatically using the mapping Module

One goal of creating maps programmatically can be to create a map book (or map set) that contains a number of different maps where the differences are typically the geographic data displayed in the data frame, title, subtitle, legend, scale, date, and other title block information such as a map sheet number or index. As mentioned previously, ArcGIS has a Data Driven Pages set of tools to assist the creation of a map book, such as creating a map sheet index feature class and creating attribute fields that can be used in naming map sheets. These can be useful when developing a map production routine. This section will cover some of the fundamentals of programmatically creating a number of unique map sheets outside the use of the Data Driven Pages tools and routines. Additional map elements can be added and modified as the programmer becomes more proficient with the methods illustrated in this chapter.

If the reader closely reviews the **Demo9b_Mapping_Module_Methods.py** script he/she will note that multiple maps are created by using conditional statements (e.g `if` statements). Within the conditional statements queries are built and used with selection routines to change

the content in the data frame, rename the title, and export the map to an output map name. If the reader used the **Sacramento_Neighborhoods.shp** file, there are 129 unique neighborhoods. Creating a map for each one using conditional statements could be quite cumbersome.

A more flexible method is to use a search cursor and a looping structure to iterate through each unique neighborhood in neighborhood shapefile and variables in query strings so that feature selections (such as `SelectLayerbyAttribute` or `SelectLayerByLocation`) or definition queries can be performed to display the desired geographic data in the map's data frame. Looping structures can also be used to iterate through text elements to modify titles, subtitles, map sheet names, etc. and for changing the display properties of layers in the table of contents. The reader will see some of these elements illustrated in the demonstration scripts for this chapter. The practical implementation of creating a series of map sheets is the goal of **Exercise 9**.

Summary

This chapter has covered many of the fundamental programming tasks that can be used to auto-generate map sheets or map books. It has introduced the primary requirements of accessing an ArcMap document, its data frame(s), layers, and map elements. In addition, the chapter has covered many of the specific map, layer, and element methods and properties that are often used when creating maps and map books. This chapter serves as a launch point to add more elements and functionality to the map automation process. For example, if the code developer needed to use a separate map grid system to create multiple map sheets that maintained a single scale, an additional layer can be used that contains these specific map grid shapes as well as attributes that contain the unique map sheet name. The code would essentially have a more involved looping strategy (possibly multiple loops) to generate all of the maps for the map series. The fundamentals remain the same, just a bit more complex to implement (and can end up saving a lot of time creating unique maps with relatively minor changes).

Chapter 9 Demos

The demonstrations in this chapter are organized into two scripts:

1. the `mapping` module overview (**Demo9a_Mapping_Module_Overview.py**), which focuses on the script organization to access different parts of a map, such as the map document, data frame, layers, and layout elements as well as manipulating some of the respective properties
2. the `mapping` module methods (**Demo9b_Mapping_Module_Methods.py**), which focus on changing the data frame's extent through the use of selected features and definition queries and making changes to the layers and layout elements. A series of seven maps are produced from the **Demo9b_Mapping_Module_Methods.py** script.

Demo 9a: Mapping Module Overview and Properties

This script demonstrates some of the fundamentals of the map document, layers, and layout elements. See the `\PythonPrimer\Chapter09\Demo9a_Mapping_Module_Overview.py` script file and the **Mapping_Module_Overview.mxd** map document. Make sure to change the data paths appropriately to properly run the script.

A unique element in this script is the use of the `from arcpy.mapping import *` line. This line “imports the entire mapping module” so that some of the code related to the mapping module can be simplified such that one need not type `arcpy.mapping` in front of the entire mapping module related functions, properties, and methods. See the Chapter 9 text for more details or consult the ArcGIS Help documents. The format of the script has been modified to fit the page. See **Demo9a_Mapping_Module_Overview.py** for the proper format.

The reader can use this demo to help develop the programming code for creating an automated map routine as well as be able to change some of the properties to see the effect on the resulting map (**test_map.pdf**) that is created at the end of the script.

After setting up the data paths pointing to the ArcMap document, the `MapDocument` function is used to locate the map document on disk. Remember, a pre-existing map document must exist before any other mapping functions can be implemented.

The first part of the script sets up the data paths and, in this case, prints out some general information about the map document. Next a series of map document properties are reported to the Python Shell so that the reader can see that a number of different properties exist for the map document. All of these properties can be found under the Map Document Properties on the File menu in ArcMap. The code has been formatted to fit the page. See the **Demo9a_Mapping_Module_Overview.py** script for the proper formatting.

```
import arcpy, sys, traceback, datetime
from arcpy.mapping import *

author = <author name>

# reformat date to MM.DD.YYYY
# NOTE: using a lowercase %y will result in MM.DD.YY format
# See Python.org or text regarding the datetime module

CUR_DATE = datetime.date.today().strftime('%m.%d.%Y')

try:

    # Get the map document
    # In this case a custom template set up for map
    # production. Change the paths as needed to run the
    # script form a local system

    datapath = 'C:\\PythonPrimer\\Chapter09\\'

    # output for PDFs
    mappath = datapath + 'MyData\\Maps\\'
    mxd = MapDocument(datapath + 'Mapping_Module_Overview.mxd')

    # Report some of the Map Document Properties

    print 'Map Document Title: ' + str(mxd.title)
    print 'Map Document Author: ' + str(mxd.author)
    print 'Map Document Summary: ' + str(mxd.summary)
    print 'Map Document Description: ' + str(mxd.description)
    print 'Map Document last Date Saved: ' + str(mxd.dateSaved)
    print 'Is the Map Document Relative Path checked? : ' +
          str(mxd.relativePaths)
```

The next step when working with the map document is to obtain a data frame. ArcGIS uses a `ListDataFrames` function to get a list of data frames. Most of the time only one exists, but a given map page can have more than one data frame. ArcGIS uses a Python list to keep track of the data frames. To access a specific data frame, the `[index]` value can be used. The index value refers to the position in the Python list the data frame is stored. Python lists have a starting index value of zero (or the first element in the list). As shown below, the first data frame is assigned to the `dataframe` variable.

A series of `print` statements are written to the Python Shell again so that the user can see a number of the properties available for the data frame. The X and Y anchor points as well as the width and height can be used to position the data frame on the map layout page. These values

are provided in page units and refer to inches on a page with the lower left corner of a page having X and Y values of (0,0). Also shown is the use of the data frame's extent where the next line provides a report of the specific X and Y minimum and maximum values. These might be needed to set other extents in the map layout or change the display of the geographic data.

```
dataframe = ListDataFrames(mxd, "Layers") [0]

print 'Map Frame Map Units: ' + str(dataframe.mapUnits)

print 'Map Frame Scale: ' + str(dataframe.scale)

print 'Map Spatial Reference: ' +
str(dataframe.spatialReference.name)
print 'Map Frame Anchor Point X Position (page units):
' + str(dataframe.elementPositionX)
print 'Map Frame Anchor Point Y Position (page units):
' + str(dataframe.elementPositionY)
print 'Map Frame Width (page units): ' +
str(dataframe.elementWidth)
print 'Map Frame Height (page units): ' +
str(dataframe.elementHeight)

# Get the extent of the data frame and assign it to a
# variable

mapExtent = dataframe.extent

# The string formatting can be found in a Python text
or at Python.org
# %f indicates a decimal value will be added to a string
# each %f below is substituted with the appropriate
# mapExtent.<value>

print 'Map Extent of Data Frame is: \n' + \
      'XMin: %f, YMin: %f, \nXMax: %f, YMax: %f' % \
      (mapExtent.XMin, mapExtent.YMin, \
       mapExtent.XMax, mapExtent.YMax)
```

The next section of script illustrates how to access a layer. Similar to the data frame, a list (`ListLayers`) is used to obtain a list of layers in the map document. Like the data frame, a single layer could have been selected by obtaining its position in the table of contents. Often a list is retrieved for all layers and then a `for` loop, like the one shown below, is used to iterate through the list. Then, if a specific set of processes is required on a specific layer (such as turning it off, changing transparency, labeling, etc.) `if` statements are used. In this example, a

number of layer properties are shown. One in particular is the `longName` property. The `longName` property can be used to access or check if a layer is a group layer and has layers within it.

```
TOCLayers = ListLayers(mxd)

print 'Processing layout elements...'

# loop through the layers
for TOCLayer in TOCLayers:
    print 'Layer Name: ' + str(TOCLayer.name)

    # longName provides the ability to use
    # "Group Layers"

    print 'Longname: ' + str(TOCLayer.longName)

    # Detail Map is the group layer
    # Neighborhoods is a layer within the group

    if TOCLayer.longName == 'Detail
        Map\Neighborhoods':

        # 50% transparency
        TOCLayer.transparency = 50

    if TOCLayer.longName == 'Detail Map\Streets':

        # turn labels on
        TOCLayer.showLabels = True

    if TOCLayer.longName == 'Detail Map\Parcels':

        # turn layer off
        TOCLayer.visibility = False
```

The script continues by creating a list of layout elements, in this case “text elements.” If no layer element type is used, such as a `'TEXT_ELEMENT'`, then all layout elements can be obtained. See Chapter 9 and the ArcGIS help for more specifics. See the individual text elements in the **Mapping_Module_Overview.mxd** map document. Remember that the map document contains uniquely named layout elements that are named in the existing map document so that a list like this can programmatically access and change the properties of the layout elements as required. Like the data frame and layers, a `for` loop can be used to iterate through each element and make changes.

```
tElements = ListLayoutElements(mxd, "TEXT_ELEMENT")

for tElement in tElements:

    # if the text element name is 'Map Title',
    # then assign a specific name for the title

    if tElement.name == 'Map Title':

        tElement.text = 'Test Map'

        # Anchor point is lower left
        tElement.elementPositionX = 8.7

    if tElement.name == 'Print Date':

        tElement.text = str(CUR_DATE)

    if arcpy.Exists(mappath + 'test_map.pdf'):
        arcpy.Delete_management(mappath + 'test_map.pdf')

print 'Writing PDF file...'

# Create the PDF document
ExportToPDF(mxd, mappath + 'test_map.pdf')

print 'Created : ' + 'test_map.pdf'

# printMap(mxd) # prints map to default printer

# print to networked printer
# printMap(mxd,
# '\\\\network_location\\printer_name')

# mxd.save()
# mxd.saveACopy(<path and file name of different
# MXD>, '<ArcGIS version>')
```

The script above shows some of the common properties that are changed with layout elements. For text elements, the text displayed on the map can be changed as well as the position (using `elementPositionX` and/or `elementPositionY`, the `elementWidth`, and the `elementHeight`), similar to the data frame position properties. To set the date of the map, the current date of the computer system is used. Toward the top of the script, the variable `CUR_DATE` is set to the current system data in a format that indicates MM.DD.YYYY. Consult a Python text or the **Python.org** site for additional date format options. The `datetime` module has also been imported to provide the functionality of dates and times.

Once all of the changes have been made, the map can be printed or exported to one of the supported formats. One of the most common export formats is PDF. The user must have a PDF reader or viewer to open the resulting PDF. The script below shows the `ExportToPDF` function to generate the PDF file.

The commented sections of the script also shows the use of `PrintMap`, `save`, and `saveACopy` routines. These can be uncommented and tested on the user's system. The `save` routine performs the same function as **File—Save**, while `saveACopy` performs the same function as **File—SaveACopy** from the ArcMap File menu. `saveACopy` can be used to save the changes made to the ArcMap document to a new file or even to a previous ArcGIS version. `save` will overwrite any changes made by the script to the existing map document.

Demo 9b: Implementing Mapping Module Methods

The previous demonstration focused on working through the logical organization of changing different map elements on a map page. This demonstration script focuses on the methods (or the actions) that can be used to change the map page, primarily the geographic data in the data frame. A number of different maps are created to illustrate different ways to change the geographic data and zoom extents in the map frame (as well as some of the layout elements). The pre-created maps (PDF documents) can be found under **\PythonPrimer\Chapter09\MyData\Maps**. Refer to **Demo9b_Mapping_Module_Methods.py** and the **Mapping_Module_Overview.mxd** for a completed script and the associated map document used in this demo.

The first part of the demonstration script is very similar to **Demo 9a**. A map document, the data frame, and layers are referenced. When the looping structure is used to iterate over the layers, a number of `if` statements are written to produce the different maps. This is not a real efficient way of creating the different maps, but for demonstration purposes, this method works fine. The code has been formatted to fit the page for the different examples. See the **Demo9b_Mapping_Module_Methods.py** script for the proper formatting.

Map 1

The code shown below begins by creating and using a query to select a specific neighborhood from the neighborhood layer. After reporting the count of selected features, the `zoomToSelectedFeatures()` method is used to modify the data frame to display the extent of the selected features. The `zoomToSelectedFeatures()` method zooms to the extent of *all* selected features in *all* layers in the data frame. In this case, since only a single feature (the *Alkali Flat* neighborhood) is selected from a single layer, the data frame zooms to the extent of a single selected feature in the neighborhood layer. Next, the map title and print date are updated through the use of the `ListLayoutElements` routine and a `for` loop to update specific text element properties and then the `ExportToPDF` routine is used to generate an output PDF. The resulting map is shown below after the code with the “selected” (highlighted) neighborhood (the outline around the neighborhood).

```

# Map 1
...
# Get a list of layers in the table of contents of the
# map document

TOCLayers = ListLayers(mxd)

# Get a list of layout elements (text elements)

tElements = ListLayoutElements(mxd, "TEXT_ELEMENT")

print 'Processing layout elements...'

# loop through the layers

for TOCLayer in TOCLayers:

# longName provides the ability to use "Group Layers"

# Detail Map is the group layer name
# Neighborhoods is a layer within the group

    if TOCLayer.longName == 'Detail Map\Neighborhoods':

# perform a Select by Attribute to select a single
# neighborhood

        query = """"NAME" = 'Alkali Flat'"""

        arcpy.SelectLayerByAttribute_management(TOCLayer,
            "NEW_SELECTION", query)

        result = arcpy.GetCount_management(TOCLayer)
        print 'Number of selected features: ' + str(result)
        # zoom to the extent of the selected feature

        dataframe.zoomToSelectedFeatures()

# cycle through the text elements to change the map
# title and date

for tElement in tElements:

    # if the text element name is 'Map Title',
    # then assign a specific name for the title
    # see the elements properties in ArcMap
    # (under size and position)

    if tElement.name == 'Map Title':

```

```

tElement.text = 'Alkali Flat'

if tElement.name == 'Print Date':

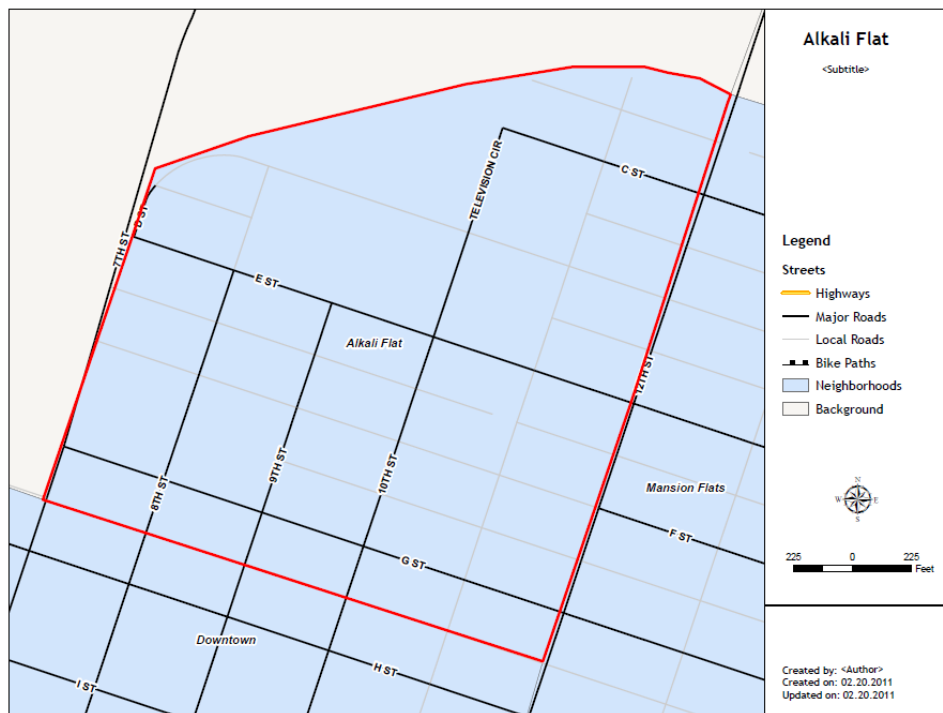
    # assign the current date
    tElement.text = str(CUR_DATE)

print 'Exporting Map 1...'

if arcpy.Exists(mappath +
'ZoomToSelectedFeatures_Map1.pdf'):
    arcpy.Delete_management(mappath +
        'ZoomToSelectedFeatures_Map1.pdf')

ExportToPDF(mxd, mappath + \
'ZoomToSelectedFeatures_Map1.pdf')

```



Map 1. Zoom to Extent of Selected Feature using `SelectLayerByAttribute` and `dataframe.zoomToSelectedFeatures()`.

Map 2

Map 2 shows the map display if no features are selected when implementing the `zoomToSelectedFeatures()` method. In this case all features are cleared and the map zooms to the full extent of the data since no features are selected.

```
# Map 2

# clear any selected features
arcpy.SelectLayerByAttribute_management(TOCLayer,
"CLEAR_SELECTION")

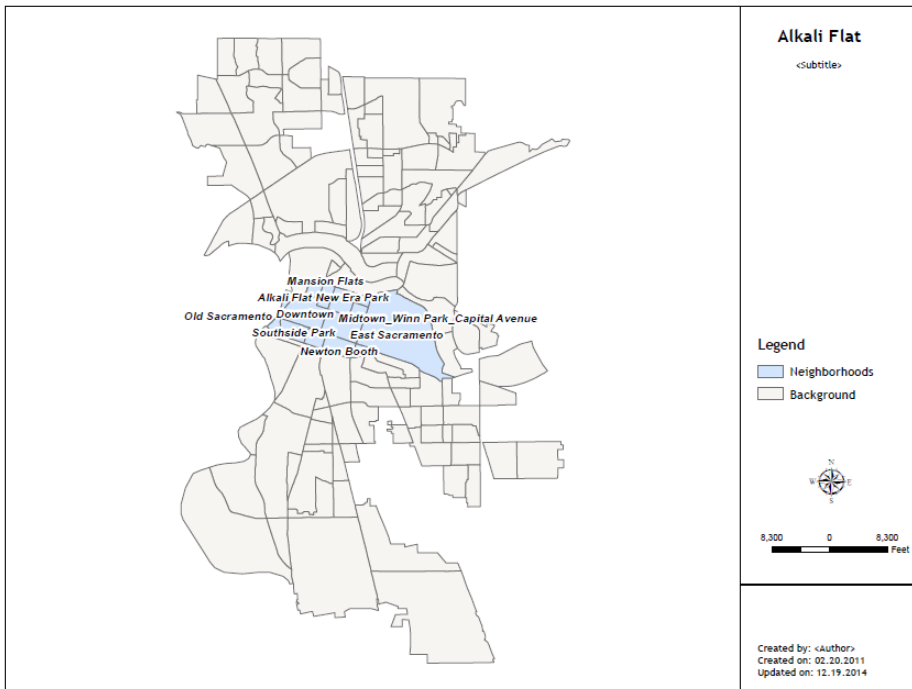
dataframe.zoomToSelectedFeatures()

print 'Exporting Map 2...'

if arcpy.Exists(mappath + 'NoSelectedFeaturesZoom_Map2.pdf'):

    arcpy.Delete_management(mappath +
        'NoSelectedFeaturesZoom_Map2.pdf')

ExportToPDF(mxd, mappath + 'NoSelectedFeaturesZoom_Map2.pdf')
```



Map 2. Zoom to Extent of Selected Feature after clearing selected features (i.e. no features are selected).

Map 3

For Map 3 the query changes to select a different neighborhood. A feature selection is performed using the `SelectLayerByAttribute` function. The *layer's* extent is retrieved from by using the `getSelectedExtent()` method that is related to the *layer* (instead of using the `dataframe.zoomToSelectedFeatures()` routine related to the *data frame*). This extent is then used to set the data frame's extent. The effect of setting this extent is similar to the `zoomToSelectedFeatures()`. Compare Map 3 to Map 1 to see how the map extents are similar. The map title is changed using a text layout element and the map is exported to a PDF file. Notice the outline around the "selected" neighborhood

```
# Map 3

query = """"NAME" = 'Downtown'"""
        arcpy.SelectLayerByAttribute_management(TOCLayer,
"NEW_SELECTION", query)

dataframe.extent = TOCLayer.getSelectedExtent()

if tElement.name == 'Map Title':

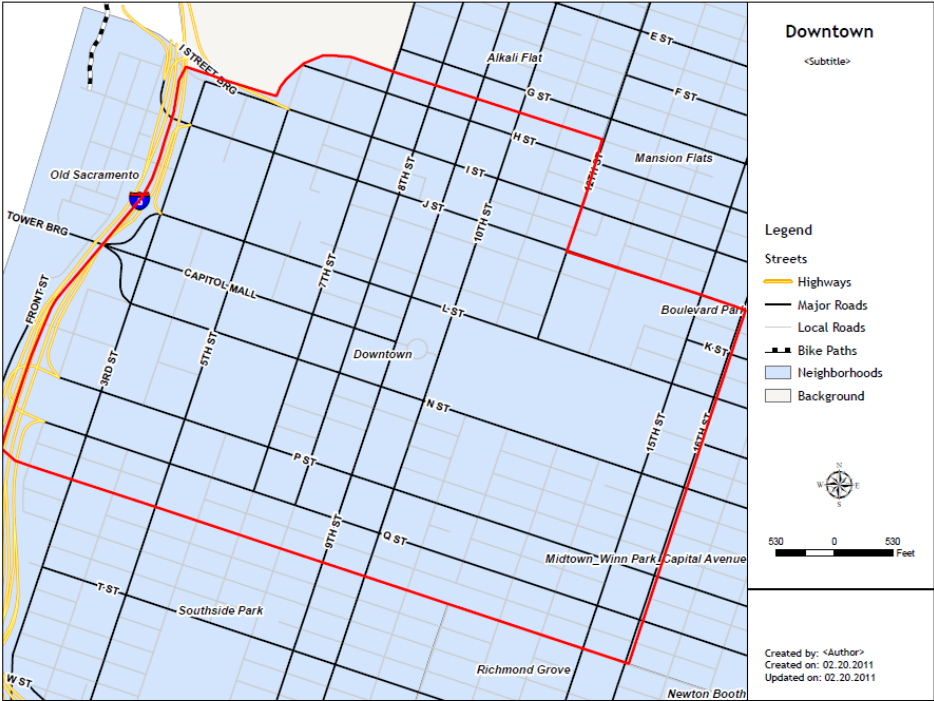
    tElement.text = 'Downtown'

print 'Exporting Map 3...'

if arcpy.Exists(mappath +
    'GetSelectedExtentfromLayer_Map3.pdf'):

    arcpy.Delete_management(mappath +
        'GetSelectedExtentfromLayer_Map3.pdf')

ExportToPDF(mxd, mappath + 'GetSelectedExtentfromLayer_Map3.pdf')
```



Map 3. Zoom to Extent of Selected Feature using the layer's `getSelectedExtent()` routine.

Map 5

The code to create Map 5 shows the same query used for Map 4, but this time the query (that was set before creating Map 3) is used as the `definitionQuery` property for the layer. The definition query displays only the features that meet the criteria, in this case, a specific neighborhood. In addition to using the definition query property, the data frame's scale is also slightly changed to change the scale by 10% so that the feature's edges do not touch the map frame, making the map look a little more centered.

```
# Map 5

# Assign the existing query to the
# definitionQuery property of the layer

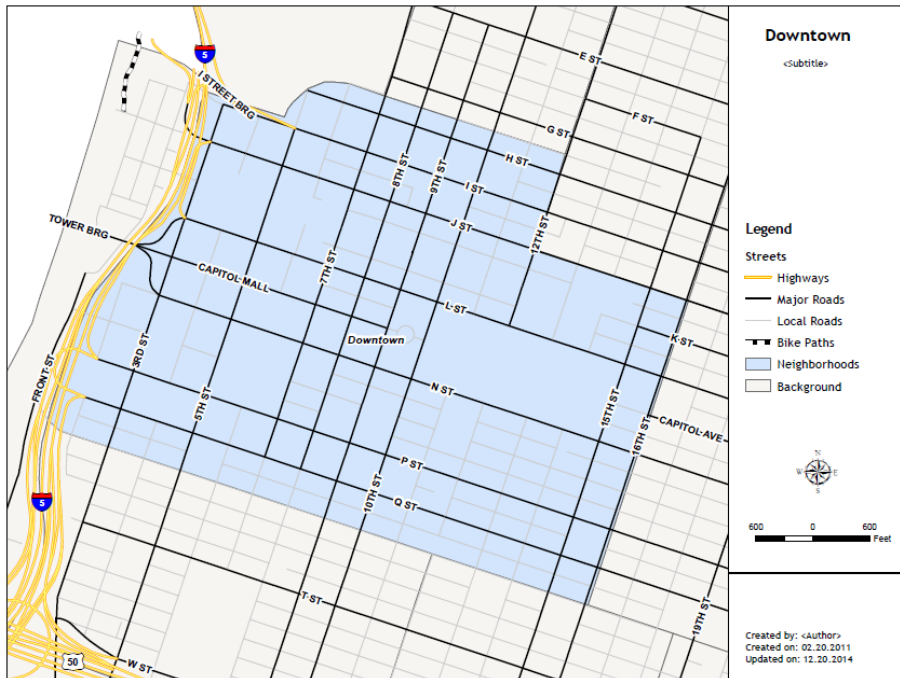
TOCLayer.definitionQuery = query

# Add a 10% "buffer" to the extent
dataframe.scale = dataframe.scale * 1.1

print 'Exporting Map 5...'

if arcpy.Exists(mappath +
    'DefinitionQuery_ScaleChange_Map5.pdf'):
    arcpy.Delete_management(mappath +
        'DefinitionQuery_ScaleChange_Map5.pdf')

ExportToPDF(mxd, mappath +
    'DefinitionQuery_ScaleChange_Map5.pdf')
```



Map 5. Neighborhood shown (shaded area) after using the `definitionQuery` property. A 10% scale has been applied to the data frame.

Map 6

Map 6 is similar to Map 3, but uses the `getExtent()` method for the neighborhood layer that has a definition query applied to it. Again, the data frame's scale is changed by 10% so that the feature's edges do not touch the data frame. Some of text elements are changed and the map is exported to a PDF file

```
# Map 6

query = """"NAME" = 'Newton Booth'"""

# Assign the existing query to the
TOCLayer.definitionQuery = query

dataframe.extent = TOCLayer.getExtent()
dataframe.scale = dataframe.scale * 1.1

for tElement in tElements:

    if tElement.name == 'Map Title':

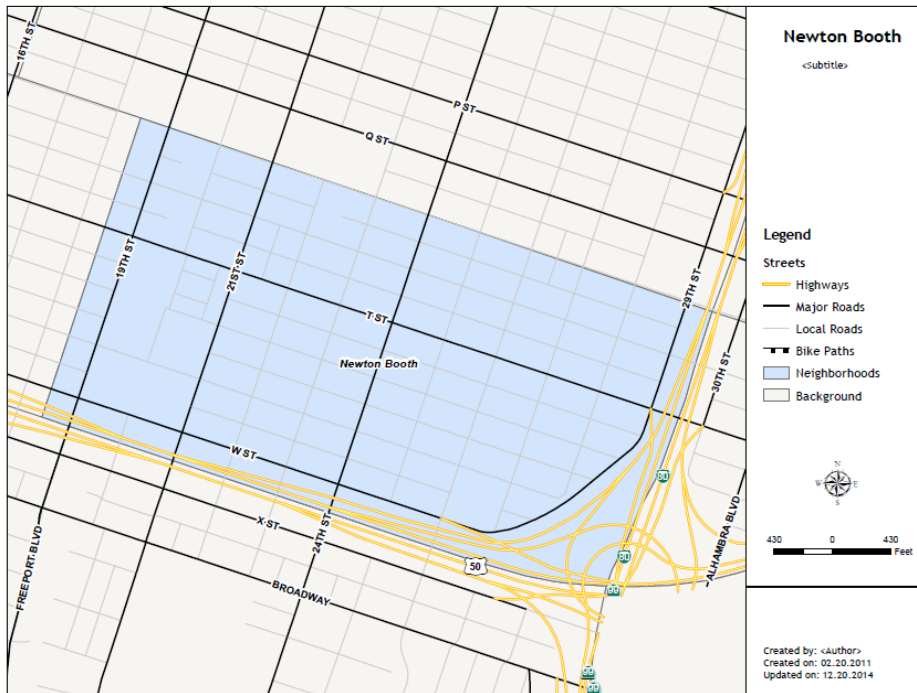
        tElement.text = 'Newton Booth'

    print 'Exporting Map 6...'

    if arcpy.Exists(mappath +
        'DefinitionQuery_GetExtent_ScaleChange_Map6.pdf'):

        arcpy.Delete_management(mappath +
            'DefinitionQuery_GetExtent_ScaleChange_Map6.pdf')

    ExportToPDF(mxd, mappath +
        'DefinitionQuery_GetExtent_ScaleChange_Map6.pdf')
```



Map 6. Zoom to a different area using the `definitionQuery` and `getExtent()` properties.

Map 7

Map 7 illustrates how to add a `.lyr` file to an existing data frame in addition to “not” changing the map’s legend by using the `autoAdd = False` option for the existing legend (which uses the `ListLegendElements` function to gain access to the map’s legend). Also, in this map, the newly added layer’s labels are turned on once a new “layer’s list” (`NewLayerList`) is created. The modified layer file is saved to a new `.lyr` file using the `saveACopy()` method. The map that is exported shows the newly added layer with the labels visible. Since a “new” layer is added to the map document, a new layer list needs to be created so that its labels can be shown (i.e. `lyr.showLabels = True`). Notice that a `for` loop and `if` statement are set up so the layer labels can be turned on and the “revised” layer can be saved as “a copy” to a new layer file.

```
# Map 7

legend = ListLayoutElements(mxd, "LEGEND_ELEMENT") [0] # Get the
first and only legend

# Get the city facilities layer
cityFacilitiesLayer = Layer(datapath + 'data\\city
facilities.lyr')

# set the autoAdd before adding the layer
legend.autoAdd = False

# Add the new layer
AddLayer(dataframe, cityFacilitiesLayer)

# new list contains the new city facilities layer
NewLayerList = ListLayers(mxd)

for lyr in NewLayerList:

    # check to see if the new layer name is present
    if lyr.name == 'City Facilities':

        # if so, turn the labels on
        lyr.showLabels = True
        cityFacilitiesLayer.saveACopy(mappath +
        'City_Facilities_Labels.lyr')

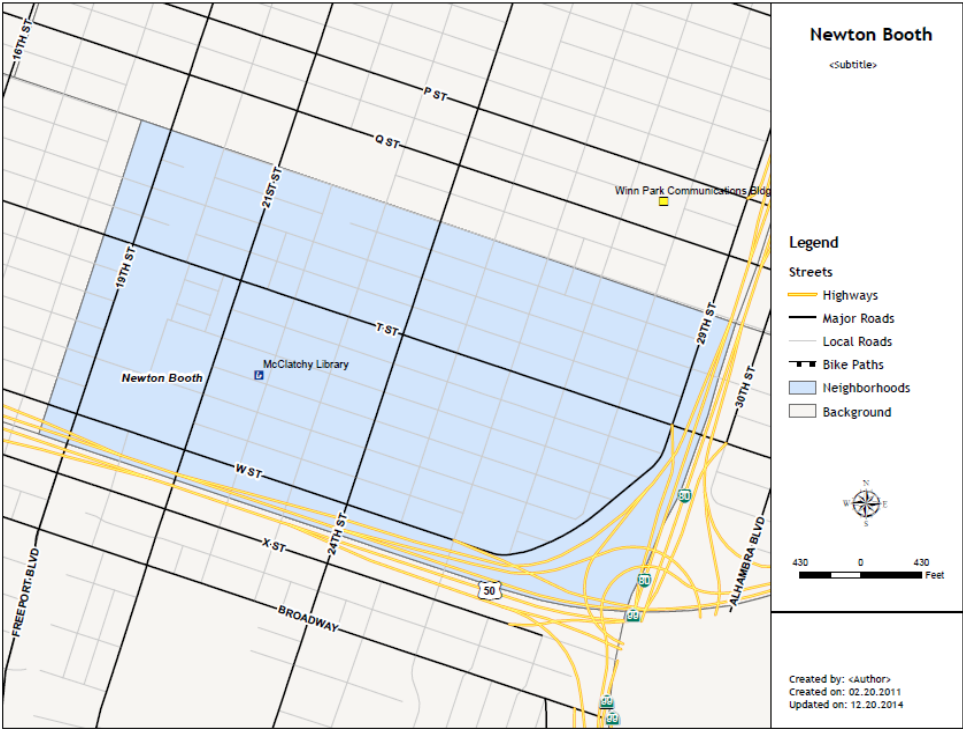
# Save the layer file to a new .lyr file name
# saveACopy only works for any layer that is
# a .lyr file

print 'Exporting Map 7...'

if arcpy.Exists(mappath + 'City_Facilities_Map7.pdf'):

    arcpy.Delete_management(mappath + 'City_Facilities_Map7.pdf')

ExportToPDF(mxd, mappath + 'City_Facilities_Map7.pdf')
```



Map 7. AddLayer and showLabels used to label the City Facilities layer.

Exercise 9 - Create a Simple Neighborhood Map Set

This exercise will expand on the concepts learned in this chapter where the code developer will automatically make a unique map for each neighborhood in the sample neighborhood set provided. The output maps will be exported to a PDF format. The reader should choose a folder to place the PDF files and use this in the script.

An ArcMap document (**Map_Template.mxd**) has been provided that contains the primary layers for the map as well as some of the layout elements.

A script called **Exercise9_StumpCode.py** can be used to work on the program. An outline is provided within the code to help the reader put additional code in the script. The steps are in the correct order. Use the demonstration scripts to assist with completing this exercise. All of the required elements can be found here. Some changes to the variables and syntax may be needed depending on how the code developer names variables, etc.

Basically, the same kinds of steps will be used that are found within the demonstration scripts, however, a search cursor and a loop will be used to iterate over all of the neighborhoods in the neighborhood layer to update the required layout elements. An export function will be used to output each unique map in PDF format.

Notes

Also before getting started the reader should review the map document and the layer and layer elements to determine layer names, layer element names, and positions on the map. Also, it will be a good idea to review the neighborhood layer (**neighborhoods.shp**) and take a look at the names of the neighborhoods. **Sacramento_neighborhoods.shp** is used in map as the "Background" layer. It is NOT used in the script.

Indicators within the stump code script show where code needs to be indented so that loops can be properly implemented. Change the data and map paths as necessary to locate the map and data as required.

The output should represent a uniquely named map that shows the specific neighborhood boundary centered in the data frame.

It will also be helpful to look at some of the questions below and answer them before beginning on the script.

Extra

A couple of “extra” sections can be found within the stump code. These are not required to implement the code. One section implements an `AddLayer` routine to add an existing layer file to the map and add it to the table of contents. A separate section can be implemented to reposition the neighborhood *'Midtown_Winn Park_Capital Avenue'* which is located within the *“Detail Map\Neighborhoods”* group layer. *'Midtown_Winn Park_Capital Avenue'* is a specific neighborhood within the *Neighborhoods* layer. See the attributes for the neighborhoods layer and the **Map_Template.mxd** Table of Contents as needed to develop the script.

Chapter 9 Questions

Answer the following based on information from the chapter. Use the ArcGIS Help documentation to supplement the information contained in Chapter 9.

1. What are the series of programming steps that are required to access a specific layer from a map?
2. Name 4 types of Layout Elements used in an ArcMap.
3. What are the specific element names for the following that can be found in the **Map_Template.mxd** document. The reader will likely need to open the map document and find the properties of the specific elements.
 - a. Title
 - b. Subtitle (where the specific neighborhood name will go)
 - c. Author
 - d. Date
 - e. Legend
4. What kind of information is required to programmatically modify, export, or print map documents?
5. Describe what the following do and list two examples of each.
 - a. Map Property
 - b. Map Method
 - c. Map Function
6. What Python routine is used to access a data frame?
7. What Python routine is used to access a layer?
8. What Python routine is used to access a layout element?
9. What Python routine is used to export a map to a PDF?

10. What Python routine is used to print a map to a local printer?
11. How can a map document that has been programmatically modified be saved?
12. What is the difference between `save()` and `saveACopy()` for the following:
 - a. A Map document
 - b. A layer file
13. What is the difference between the following?
 - a. `getExtent`
 - b. `getSelectedExtent`
 - c. `zoomToSelectedFeatures`
14. What is the benefit of using a search cursor and this syntax for the query in the Exercise 9 script?

```
query = "''''NAME" = '""' + NHName + "''''''''"
```

where `NHName` represents a variable for the name of a neighborhood
15. In the Exercise 9 script (once you are able to export PDF maps), why does the **Map_Template.mxd** document NOT need to be saved?

Accessing Data, Demos, and Code

The reader can obtain the supplemental information for this book at the author's website using the following credentials (which are case sensitive):

<http://pythonprimer.urbandalespatial.com/resources>

Username: PythonPrimer

Password: PP4AGIS!

Additional information will be provided on this website with any updates, changes, etc.

References

Author's website – www.urbandalespatial.com

Jennings, Nathan. "Managing Street Sign Assets: An enterprise geospatial business systems integration solution." *ArcUser*TM, Winter 2009. Date Accessed: 11.09.2011

<http://www.Esri.com/news/arcuser/0109/streetsigns.html>

ArcGIS

ArcGIS Resource Center - <http://resources.arcgis.com/>

ArcGIS Web-based Help - <http://resources.arcgis.com/en/communities/>

ArcGIS Blog - <http://blogs.Esri.com>

ArcGIS Forums - <https://geonet.esri.com/community/developers/gis-developers/python>

Geoprocessing script examples and models -

<http://resources.arcgis.com/en/communities/python/>

Esri Training courses - <http://www.esri.com/training/main>

ArcUser - <http://www.Esri.com/news/arcuser>

ArcGIS Resource Center. Exception Code Snippet. Esri, 2011.

<http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/002z0000000q000000>.

Python

Python website – www.python.org

Lutz, Mark. Learning Python, 4th Ed. Beijing: O'Reilly Media, Inc., 2009.

Organizations

Esri – www.Esri.com

American River College GIS Program - <http://wserver.arc.losrios.edu/~earthscience/>

Sierra College GIS Program - <http://www.sierracollege.edu/academics/divisions/science-math/geography.php>

UC Davis Extension - <https://extension.ucdavis.edu/subject-areas/geographic-information-systems>

Del Mar College - http://www.delmar.edu/CIS_-_Geographical_Information_Systems.aspx

City of Sacramento GIS – www.cityofsacramento.org/gis

County of Sacramento GIS – www.sacgis.org

Cal Atlas – <http://atlas.ca.gov>

INDEX

A

ArcToolbox, 8, 19, 24

C

Check Module, 49, 74

Cursors, 12, 65, 66, 67, 98

 Insert Cursor, 67, 71, 75, 99, 131, 132, 133

 Search Cursor, 67

 Update Cursor, 67, 77, 99

D

data path, 51, 79, 113, 114, 175

Describing Data, 139

E

error handling, 19, 173, 174, 176, 181

escape characters, 32

F

feature class, 13, 24, 26, 33, 35, 37, 38, 39, 45, 46, 47, 54,
 58, 61, 62, 63, 65, 70, 71, 75, 77, 79, 85, 86, 87, 88, 90,
 91, 92, 94, 96, 107, 117, 121, 123, 124, 126, 129, 130,
 134, 138, 161, 181, 185, 220

feature classes, 23, 48, 73, 85, 91, 93, 98, 99, 105, 114,
 121, 139, 144, 146, 159, 160, 194, 218

G

geoprocessing, 7, 19, 21, 23, 33, 37, 41, 47, 51, 53, 63, 86,
 144, 201

I

if statement, 48, 73, 107, 156, 161, 183, 184, 185

L

Listing Data, 144

M

Make Feature Layer, 25, 33, 45, 51, 53, 54, 55, 56, 63, 94,
 125

Make Table View, 25, 33, 63, 94, 125

mapping module, 12, 19, 191, 196, 197, 200, 201, 202,
 203, 208, 223, 225

ModelBuilder, 8, 24

P

Python

 IDLE, 40, 42, 49, 74, 77, 111

 Python Shell, 40, 42, 46, 49, 57, 74, 77, 95, 96, 103,
 126, 128, 129, 140, 168, 176, 178, 184, 187, 203,
 207, 225, 226

Python Constructs

 indentation, 99

Python list, 70, 80, 102, 109, 115, 132, 133, 134, 139, 145,
 160, 204, 205, 226

Python Modules

 arcpy, 13, 45, 51, 101, 105, 113, 130, 131, 140, 144,
 153, 165, 185, 202, 203, 225

 os, 99, 151, 159

 sys, 45, 51, 101, 105, 113, 153

 traceback, 45, 51, 101, 113, 153

Q

query syntax, 23, 26, 27, 55, 59, 92, 165, 182

Quotes

 double quotes, 27

 single quotes, 27, 30

 triple double quotes, 11

R

raster, 12, 39, 139, 146, 147, 149, 151, 153, 154, 158, 168

S

Spatial Analyst, 12, 139, 141, 146, 147, 149, 151, 153, 158,
 159, 163, 165, 181

strings, 28, 29, 31, 32, 63, 99, 107, 168, 181, 221

T

table join, 65, 86, 87, 93, 98, 121, 137

V

variables, 19, 37, 45, 48, 54, 56, 61, 84, 102, 107, 110, 115,
 121, 126, 151, 153, 154, 221, 245

W

workspace, 45, 46, 51, 72, 79, 82, 101, 105, 106, 107, 108,
 109, 113, 114, 121, 122, 130, 140, 144, 145, 146, 148,
 153, 154, 159, 161, 165, 169, 175, 203

