Quick answers to common problems

Programming ArcGIS 10.1 with Python Cookbook

Over 75 recipes to help you automate geoprocessing tasks, create solutions, and solve problems for ArcGIS with Python

Eric Pimpler

[PACKT] PUBLISHING

Programming ArcGIS 10.1 with Python Cookbook

Over 75 recipes to help you automate geoprocessing tasks, create solutions, and solve problems for ArcGIS with Python

Eric Pimpler

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Programming ArcGIS 10.1 with Python Cookbook

First published: February 2013

Production Reference: 1120213

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-444-5

`www.packtpub.com`

Cover Image by Sujay Gawand (`sujaygawand@gmail.com`)

Credits

Author
Eric Pimpler

Reviewers
Alissa Bickar
Ann Stark
Tripp Corbin, CFM, GISP

Acquisition Editor
Andrew Duckworth

Lead Technical Editor
Dayan Hyames

Technical Editor
Lubna Shaikh

Project Coordinator
Amey Sawant

Proofreaders
Chris Brown
Sandra Hopper

Indexer
Tejal Soni

Graphics
Aditi Gajjar

Production Coordinators
Manu Joseph
Nitesh Thakur

Cover Work
Nitesh Thakur

About the Author

Eric Pimpler is the founder and owner of GeoSpatial Training Services (`geospatialtraining.com`) and has over 20 years of experience in implementing and teaching GIS solutions using ESRI technology. Currently, Eric focuses on ArcGIS scripting with Python, and the development of custom ArcGIS Server web and mobile applications using JavaScript.

Eric has a Bachelor's degree in Geography from Texas A&M University and a Master's of Applied Geography degree with a concentration in GIS from Texas State University.

About the Reviewers

Alissa Bickar is a GIS Analyst and instructor who has a large interest in geospatial technologies and projects. She has developed various courses as an instructor for GeoSpatial Training Services and has been appointed as the ArcGIS Desktop Training Program Manager with GTS. She is responsible for developing and updating course materials for the program, as well as assisting clients with their course and annual subscriptions.

She has extensive experience in the GIS field as a consultant to federal and local governments, environmental engineering firms, and many clients in the Oil and Gas industry. She also has experience as a college professor and has helped develop GIS and Geography courses for higher education.

Alissa has both a Bachelor's and Master's degree in Geography from California University of Pennsylvania.

Ann Stark, a GISP since 2005, has been active in the GIS profession for 15 years. She is passionate about GIS and is an active and engaging member of the GIS community in the Pacific Northwest of the United States, coordinating local user groups and serving as the President of the region's GIS professional group. She is an enthusiastic teacher who explains how to effectively use Python with ArcGIS and maintains a blog devoted to the topic at `GISStudio.wordpress.com`. She co-owns a GIS consulting business, called Salish Coast Sciences, which provides strategic planning, process automation, and GIS development services.

To unwind from technology, Ann enjoys spending time with her husband and son at their urban farm in the heart of a city where they seek to live sustainably and as self-sufficiently as an urban farm allows.

Tripp Corbin, CFM, GISP is the CEO and a Co-founder of eGIS Associates, Inc. He has over 20 years of surveying, mapping, and GIS-related experience. Tripp is recognized as an industry expert with a variety of geospatial software packages including Esri, Autodesk, and Trimble products. He holds multiple certifications including **Microsoft Certified Professional** (**MCP**), **Certified Floodplain Manager** (**CFM**), **Certified GIS Professional** (**GISP**), **Comptia Certified Technical Trainer** (**CTT+**), and **Esri Certified Trainer**.

As a GIS Instructor, Tripp has taught students from around the world the power of GIS. He has authored many classes on topics ranging from the beginner level, such as Introduction to GIS, GIS Fundamentals to more advanced topics such as ArcGIS Server Installation, Configurations and Tweaks. Tripp recently helped the University of North Alabama Continuing Studies Center develop an online GIS Analyst Certificate Program.

Tripp believes in giving back to the profession that has given him so much. As a result, he is heavily active in multiple GIS-oriented professional organizations. He is a past President of Georgia, URISA, and was recently the Keynote Speaker for the Georgia Geospatial Conference. Tripp also serves on the URISA International Board of Directors, in addition to being a member of the GISP Application Review committee and an At-Large GITA Southeast Board Member.

Other contributions Tripp has made to the GIS Profession include helping to draft the new Geospatial Technology Competency Model that was adopted by the US Department of Labor, presenting at various conferences and workshops around the US, and providing help to other GIS professionals around the world on various blogs, lists, and forums.

Table of Contents

Preface

ArcGIS is an industry-standard geographic information system from ESRI.

This book will show you how to use the Python programming language to create geoprocessing scripts, tools, and shortcuts for the ArcGIS Desktop environment.

This book will make you a more effective and efficient GIS professional, by showing you how to use the Python programming language with ArcGIS Desktop to automate geoprocessing tasks, manage map documents and layers, find and fix broken data links, edit data in feature classes and tables, and much more.

Programming ArcGIS 10.1 with Python Cookbook starts by covering fundamental Python programming concepts in an ArcGIS Desktop context. Using a how-to instruction style, you'll then learn how to use Python to automate common important ArcGIS geoprocessing tasks.

In this book, you will also cover specific ArcGIS scripting topics that will help save you time and effort when working with ArcGIS. Topics include managing map document files, automating map production and printing, finding and fixing broken data sources, creating custom geoprocessing tools, and working with feature classes and tables, among others.

In *Programming ArcGIS 10.1 with Python Cookbook*, you'll learn how to write geoprocessing scripts using a pragmatic approach designed around accomplishing specific tasks in a cookbook style format.

What this book covers

Chapter 1, *Fundamentals of the Python Language for ArcGIS*, will cover many of the basic language constructs found in Python. Initially, you'll learn how to create new Python scripts or edit existing scripts. From there, you'll get into language features, such as adding comments to your code, variables, and the built-in typing systems that makes coding with Python easy and compact. Furthermore, we'll look at the various built-in data-types that Python offers, such as strings, numbers, lists, and dictionaries. In addition to this, we'll cover statements, including decision support and looping structures for making decisions in your code and/or looping through a code block multiple times.

Chapter 2, *Writing Basic Geoprocessing Scripts with ArcPy*, will teach the basic concepts of the ArcPy Python site package for ArcGIS, including an overview of the basic modules, functions, and classes. The reader will be able write a geoprocessing script using ArcPy with Python.

Chapter 3, *Managing Map Documents and Layers*, will use the Arcpy Mapping module to manage map document and layer files. You will learn how to add and remove geographic layers from map document files, insert layers into data frames, and move layers around within the map document. The reader will also learn how to update layer properties and symbology.

Chapter 4, *Finding and Fixing Broken Data Links*, will teach how to generate a list of broken data sources in a map document file and apply various Arcpy Mapping functions to fix these data sources. The reader will learn how to automate the process of fixing data sources across many map documents.

Chapter 5, *Automating Map Production and Printing*, will teach how to automate the process of creating production-quality maps. These maps can then be printed, exported to image file formats, or exported to PDF files for inclusion in map books.

Chapter 6, *Executing Geoprocessing Tools from Scripts*, will teach how to write scripts that access and run geoprocessing tools provided by ArcGIS.

Chapter 7, *Creating Custom Geoprocessing Tools*, will teach how to create custom geoprocessing tools that can be added to ArcGIS and shared with other users. Custom geoprocessing tools are attached to a Python script that process or analyze geographic data in some way.

Chapter 8, *Querying and Selecting Data*, will teach how to execute the **Select by Attribute** and **Select by Location** geoprocessing tools from a script to select features and records. The reader will learn how to construct queries that supply an optional `where` clause for the **Select by Attribute** tool. The use of feature layers and table views as temporary datasets will also be covered.

Chapter 9, *Using the ArcPy Data Access Module to Select, Insert, and Update Geographic Data and Tables*, will teach how to create geoprocessing scripts that select, insert, or update data from geographic data layers and tables. With the new ArcGIS 10.1 Data Access module, geoprocessing scripts can create in-memory tables of data, called cursors, from feature classes and tables. The reader will learn how to create various types of cursors including search, insert, and update

Chapter 10, *Listing and Describing GIS Data*, will teach how to obtain descriptive information about geographic datasets through the use of the Arcpy Describe function. As the first step in a multi-step process, geoprocessing scripts frequently require that a list of geographic data be generated followed by various geoprocessing operations that can be run against these datasets.

Chapter 11, *Customizing the ArcGIS Interface with Add-Ins*, will teach how to customize the ArcGIS interface through the creation of Python add-ins. Add-ins provide a way of adding user interface items to ArcGIS Desktop through a modular code base designed to perform specific actions. Interface components can include buttons, tools, toolbars, menus, combo boxes, tool palettes, and application extensions. Add-ins are created using Python scripts and an XML file that define how the user interface should appear.

Chapter 12, *Error Handling and Troubleshooting*, will teach how to gracefully handle errors and exceptions as they occur while a geoprocessing script is executing. Arcpy and Python errors can be trapped with the Python `try/except` structure and handled accordingly.

Appendix A, *Automating Python Scripts*, will teach how to schedule geoprocessing scripts to run at a prescribed time. Many geoprocessing scripts take a long time to fully execute and need to be scheduled to run during non-working hours on a regular basis. The reader will learn how to create batch file containing geoprocessing scripts and execute these at a prescribed time.

Appendix B, *Five Things Every GIS Programmer Should Know How to Do with Python*, will teach how to write scripts that perform various general purpose tasks with Python. Tasks, such as reading and writing delimited text files, sending e-mails, interacting with FTP servers, creating ZIP files, and reading and writing JSON and XML files are common. Every GIS programmer should know how to write Python scripts that incorporate this functionality.

What you need for this book

To complete the exercises in this book, you will need to have installed ArcGIS Desktop 10.1 at either the Basic, Standard, or Advanced license level. Installing ArcGIS Desktop 10.1 will also install Python 2.7 along with the IDLE Python code editor.

Who this book is for

Programming ArcGIS 10.1 with Python Cookbook is written for GIS professionals who wish to revolutionize their ArcGIS workflow with Python. Whether you are new to ArcGIS or a seasoned professional, you almost certainly spend time each day performing various geoprocessing tasks. This book will teach you how to use the Python programming language to automate these geoprocessing tasks and make you a more efficient and effective GIS professional.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: " we have loaded the `ListFeatureClasses.py` script with IDLE."

A block of code is set as follows:

```
import arcpy
fc = "c:/ArcpyBook/data/TravisCounty/TravisCounty.shp"

# Fetch each feature from the cursor and examine the extent properties
and spatial reference
for row in arcpy.da.SearchCursor(fc, ["SHAPE@"]):
  # get the extent of the county boundary
  ext = row[0].extent
  # print out the bounding coordinates and spatial reference
  print "XMin: " + ext.XMin
  print "XMax: " + ext.XMax
  print "YMin: " + ext.YMin
  print "YMax: " + ext.YMax
  print "Spatial Reference: " + ext.spatialReference.name
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import arcpy

fc = "c:/data/city.gdb/streets"

# For each row print the Object ID field, and use the SHAPE@AREA
# token to access geometry properties

with arcpy.da.SearchCursor(fc, ("OID@", "SHAPE@AREA")) as cursor:
  for row in cursor:
    print("Feature {0} has an area of {1}".format(row[0], row[1]))
```

Any command-line input or output is written as follows:

```
[<map layer u'City of Austin Bldg Permits'>, <map layer u'Hospitals'>,
<map layer u'Schools'>, <map layer u'Streams'>, <map layer u'Streets'>,
<map layer u'Streams_Buff'>, <map layer u'Floodplains'>, <map layer
u'2000 Census Tracts'>, <map layer u'City Limits'>, <map layer u'Travis
County'>]
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "go to **Start** | **Programs** | **ArcGIS** | **Python 2.7** | **IDLE**".

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Fundamentals of the Python Language for ArcGIS

Python supports many of the programming constructs found in other languages. In this chapter, we'll cover many of the basic language constructs found in Python. Initially, we'll cover how to create new Python scripts and edit existing scripts. From there, we'll delve into language features, such as adding comments to your code, creating and assigning data to variables, and built-in variable typing with Python, which makes coding with Python easy and compact.

Next, we'll look at the various built-in data-types that Python offers, such as strings, numbers, lists, and dictionaries. Classes and objects are a fundamental concept in object-oriented programming and in the Python language. We'll introduce you to these complex data structures, which you'll use extensively when you write geoprocessing scripts with ArcGIS.

In addition, we'll cover statements including decision support and looping structures for making decisions in your code and/or looping through a code block multiple times along with `with` statements, which are used extensively with the new `cursor` objects in the Arcpy Data Access module. Finally, you'll learn how to access modules that provide additional functionality to the Python language. By the end of this chapter, you will have learned the following:

- How to create and edit new Python scripts
- Python language features
- Comments and data variables
- Built-in datatypes (Strings, Numbers, Lists, and Dictionaries)
- Complex data structures
- Looping structures
- Additional Python functionality

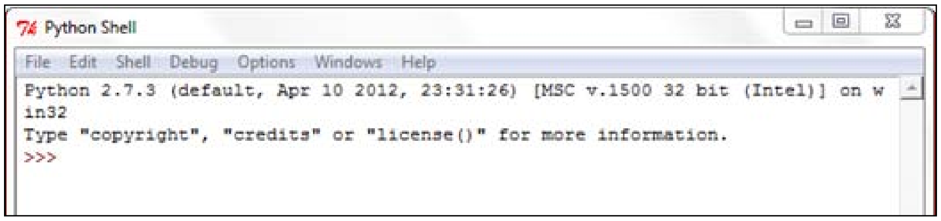
Using IDLE for Python script development

As I mentioned in the preface, when you install ArcGIS Desktop, Python is also installed along with a tool called IDLE that allows you to write your own code. **IDLE** stands for **Integrated DeveLopment Environment**. Because it is available with every ArcGIS Desktop installation, we'll use the IDLE development environment for many of the scripts that we write in this book along with the Python window embedded in ArcGIS Desktop. As you progress as a programmer, you may find other development tools that you prefer over IDLE. You can write your code in any of these tools.

The Python shell window

To start the IDLE development environment for Python, you can go to **Start** | **Programs** | **ArcGIS** | **Python 2.7** | **IDLE**. Please note that the version of Python installed with ArcGIS will differ depending upon the ArcGIS version that you have installed. For example, ArcGIS 10.0 uses Python 2.6 while ArcGIS 10.1 uses Python 2.7.

A Python shell window similar to the screenshot will be displayed:



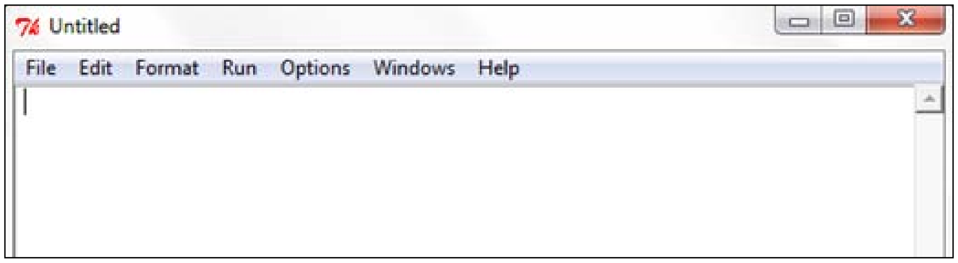
```
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on w
in32
Type "copyright", "credits" or "license()" for more information.
>>>
```

The Python shell window is used for output and error messages generated by scripts. A common mistake for beginners is to assume that the geoprocessing scripts will be written in this shell window. That is not the case. You will need to create a separate code window to hold your scripts.

Although the shell window isn't used to write entire scripts, it can be used to interactively write code and get immediate feedback. ArcGIS has a built-in Python shell window that you can use in much the same way. We'll examine the ArcGIS Python window in the next chapter.

The Python script window

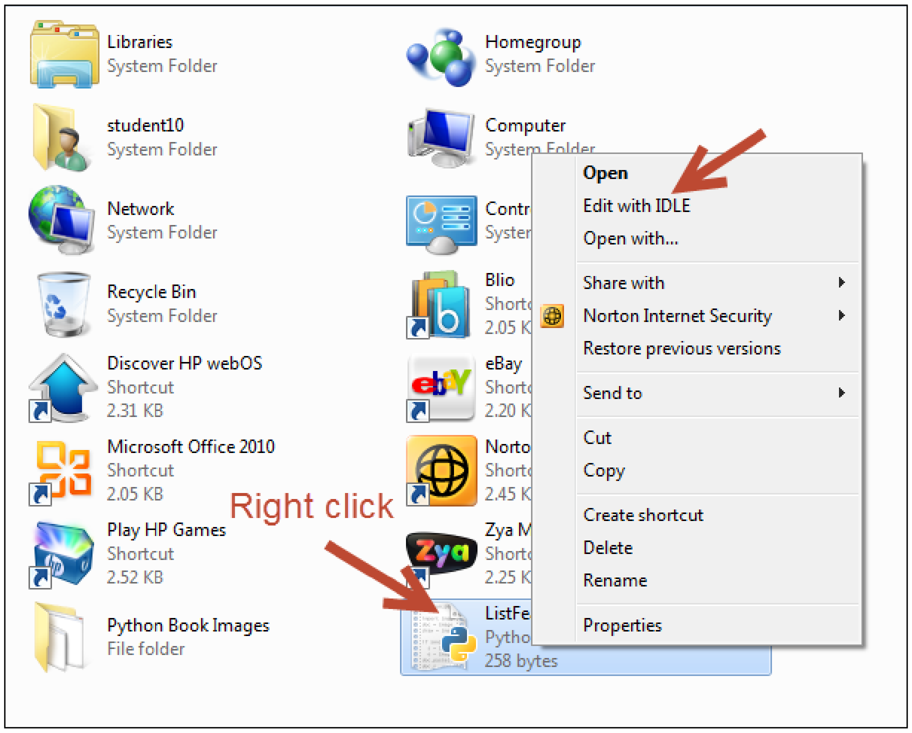
Your scripts will be written in IDLE inside a separate window known as the **Python script window**. To create a new code window, select **File** | **New Window** from the IDLE shell window. A window similar to that in the following screenshot will be displayed:



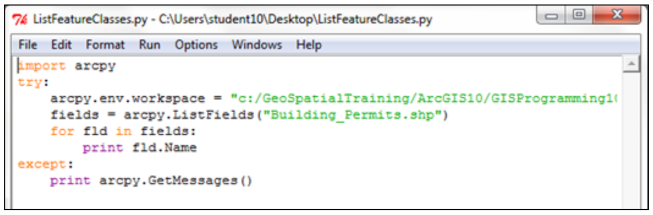
Your Python scripts will be written inside this new code window. Each script will need to be saved to a local or network drive. By default, scripts are saved with a `.py` file extension.

Editing existing Python scripts

Existing Python script files can be opened from Windows Explorer by right-clicking on the file and selecting **Edit with IDLE**, which brings up a new shell window along with the script loaded in the Python script editor. You can see an example of this in the following screenshot:



In this instance, we have loaded the `ListFeatureClasses.py` script with IDLE. The code is loaded inside the script window:


ListFeatureClasses.py - C:\Users\student10\Desktop\ListFeatureClasses.py
File Edit Format Run Options Windows Help

```
import arcpy
try:
    arcpy.env.workspace = "c:/GeoSpatialTraining/ArcGIS10/GISProgramming1(
    fields = arcpy.ListFields("Building_Permits.shp")
    for fld in fields:
        print fld.Name
except:
    print arcpy.GetMessages()
```

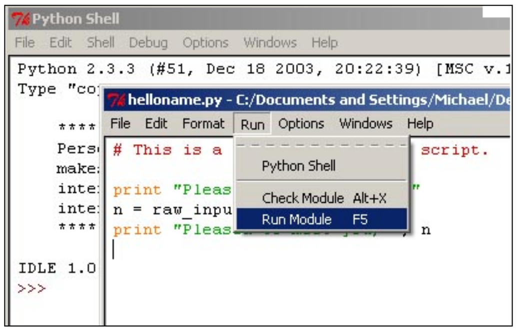
Now that the code window is open, you can begin writing or editing code. You can also perform some basic script debugging with the IDLE interface. Debugging is the process of identifying and fixing errors in your code.

Executing scripts from IDLE

Once you've written a geoprocessing script in the IDLE code window or opened an existing script, you can execute the code from the interface. IDLE does provide functionality that allows you to check the syntax of your code before running the script. In the code window, select **Run** | **Check Module** to perform a syntax check of your code.

Any syntax errors will be displayed in the shell window. If there aren't any syntax errors, you should just see the prompt in the shell window. While the IDLE interface can be used to check for syntax errors, it doesn't provide a way of checking for logical errors in your code nor does it provide more advanced debugging tools found in other development environments, such as PythonWin or Wingware.

Once you're satisfied that no syntax errors exist in your code, you can run the script. Select **Run** | **Run Module** to execute the script:


Python Shell
File Edit Shell Debug Options Windows Help
hellonăme.py - C:/Documents and Settings/Michael/De
File Edit Format Run Options Windows Help
Python Shell
Check Module Alt+X
Run Module F5

```
Python 2.3.3 (#51, Dec 18 2003, 20:22:39) [MSC v.1
Type "co
    ****
    Pers
    make:
    inte:
    inte:
    ****

IDLE 1.0
>>>
```

```
# This is a                 script.

print "Pleas                "
n = raw_inpu
print "Pleas               , n
```

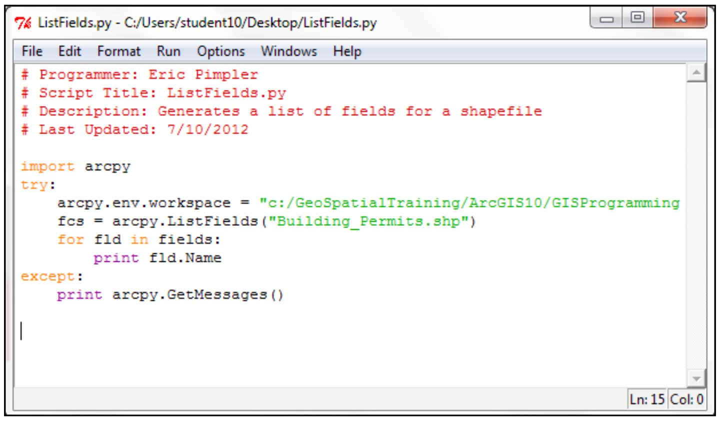
Any error messages will be written to the shell window along with output from `print` statements and system-generated messages. The `print` statement simply outputs a string to the shell window. It is often used for updating the status of a running script or for debugging the code.

Python language fundamentals

To effectively write geoprocessing scripts for ArcGIS, you are going to need to understand at least the basic constructs of the Python language. Python is easier to learn than most other programming languages, but it does take some time to learn and effectively use it. This section will teach you how to create variables, assign various datatypes to variables, understand the different types of data that can be assigned to variables, use different types of statements, use objects, read and write files, and import third-party Python modules.

Commenting code

Python scripts should follow a common structure. The beginning of each script should serve as documentation detailing the script name, author, and a general description of the processing provided by the script. This documentation is accomplished in Python through the use of comments. Comments are lines of code that you add to your script that serve as a documentation of what functionality the script provides. These lines of code begin with a single pound sign (`#`) or a double pound sign (`##`), and are followed by whatever text you need to document the code. The Python interpreter does not execute these lines of code. They are simply used for documenting your code. In the next screenshot, the commented lines of code are displayed in red. You should also strive to include comments throughout your script to describe important sections of your script. This will be useful to you (or another programmer) when the time comes to update your scripts.


ListFields.py - C:/Users/student10/Desktop/ListFields.py
File Edit Format Run Options Windows Help
Ln: 15 Col: 0

```
# Programmer: Eric Pimpler
# Script Title: ListFields.py
# Description: Generates a list of fields for a shapefile
# Last Updated: 7/10/2012

import arcpy
try:
    arcpy.env.workspace = "c:/GeoSpatialTraining/ArcGIS10/GISProgramming
    fcs = arcpy.ListFields("Building_Permits.shp")
    for fld in fields:
        print fld.Name
except:
    print arcpy.GetMessages()
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Importing modules

Although Python includes many built-in functions, you will frequently need to access specific bundles of functionality, which are stored in external modules. For instance, the **Math module** stores specific functions related to processing numeric values and the **R module** provides statistical analysis functions. Modules are imported through the use of the `import` statement. When writing geoprocessing scripts with ArcGIS, you will always need to import the ArcPy module, which is the Python package for accessing GIS tools and functions provided by ArcGIS. `import` statements will be the first lines of code (not including comments) in your scripts:

```
import arcpy, os
```

Variables

At a high level, you can think of a variable as an area in your computer's memory reserved for storing values while the script is running. Variables that you define in Python are given a name and a value. The values assigned to variables can then be accessed by different areas of your script as needed, simply by referring to the variable name. For example, you might create a variable that contains a feature class name, which is then used by the **Buffer** tool to create a new output dataset. To create a variable, simply give it a name followed by the assignment operator, which is just an equal sign (=), and then a value:

```
fcParcels = "Parcels"
fcStreets = "Streets"
```

The following table illustrates the variable name and values assigned to the variable using the preceding code example:

Variable name	Variable value
`fcParcels`	`Parcels`
`fcStreets`	`Streets`

There are certain naming rules that you must follow when creating variables, including the following:

- Can contain letters, numbers, and underscores
- First character must be a letter
- No special characters in variable name other an underscore
- Can't use Python keywords

There are a few dozen Python keywords that must be avoided including `class`, `if`, `for`, `while`, and others.

Some examples of legal variable names in Python:

- `featureClassParcel`
- `fieldPopulation`
- `field2`
- `ssn`
- `my_name`

Some examples of illegal variable names in Python:

- `class` (Python keyword)
- `return` (Python keyword)
- `$featureClass` (illegal character, must start with a letter)
- `2fields` (must start with a letter)
- `parcels&Streets` (illegal character)

Python is a case-sensitive language, so pay particular attention to the capitalization and naming of variables in your scripts. Case-sensitivity issues are probably the most common source of errors for new Python programmers, so always consider this as a possibility when you encounter errors in your code. Let's look at an example. The following is a list of three variables; note that although each variable name is the same, the casing is different, resulting in three distinct variables.

- `mapsize = "22x34"`
- `MapSize = "8x11"`
- `Mapsize = "36x48"`

If you print these variables, you will get the following output:

```
print mapsize
>>> 22x34

print MapSize
>>> 8x11

print Mapsize
>>>36x48
```

Python variable names need to be consistent throughout the script. Best practice is to use camel casing, wherein the first word of a variable name is all lowercase and then each successive word begins with an uppercase letter. This concept is illustrated in the following example with the variable name `fieldOwnerName`. The first word (`field`) is all lower case followed by an uppercase letter for the second word (`Owner`) and third word (`Name`):

```
fieldOwnerName
```

In Python, variables are dynamically typed. **Dynamic typing** means that you can define a variable and assign data to it without specifically defining that a variable name will contain a specific type of data. Commonly used datatypes that can be assigned to variables include the following:

Datatype	**Example value**	**Code example**
String	`"Streets"`	`fcName = "Streets"`
Number	`3.14`	`percChange = 3.14`
Boolean	`True`	`ftrChanged = true`
List	`Streets, Parcels, Streams`	`lstFC = ["Streets", "Parcels", "Streams"]`
Dictionary	`'0':Streets,'1':Parcels`	`dictFC = {'0':Streets,'1':Parcels]`
Object	`Extent`	`spatialExt = map.extent`

We will discuss each of these data-types in greater detail in the coming sections.

For instance, in C# you would need to define a variable's name and type before using it. This is not necessary in Python. To use a variable, simply give it a name and value, and you can begin using it right away. Python does the work behind the scenes to figure out what type of data is being held in the variable.

For example, in C# .NET you would need to name and define the datatype for a variable before working with the variable. In the following code example, we've created a new variable called `aTouchdown`, which is defined as an integer variable, meaning that it can contain only integer data. We then assign the value `6` to the variable:

```
int aTouchdown;
aTouchdown = 6;
```

In Python, this same variable can be created and assigned data through dynamic typing. The Python interpreter is tasked with dynamically figuring out what type of data is assigned to the variable:

```
aTouchdown = 6
```

Your Python geoprocessing scripts for ArcGIS will often need to reference the location of a dataset on your computer or perhaps a shared server. References to these datasets will often consist of paths stored in a variable. In Python, pathnames are a special case that deserve some extra mention. The backslash character in Python is a reserved escape character and a line continuation character, thus there is a need to define paths using two back slashes, a single forward slash, or a regular single backslash prefixed with the letter `r`. These pathnames are always stored as strings in Python. You'll see an example of this in the following section.

Illegal path reference:

```
fcParcels = "c:\Data\Parcels.shp"
```

Legal path references:

```
fcParcels = "c:/Data/Parcels.shp"
fcParcels = "c:\\Data\\Parcels.shp"
fcParcels = r"c:\Data\Parcels.shp"
```

There may be times when you know that your script will need a variable, but don't necessarily know ahead of time what data will be assigned to the variable. In these cases, you could simply define a variable without assigning data to it. Data that is assigned to the variable can also be changed while the script is running.

Variables can hold many different kinds of data including primitive datatypes such as strings and numbers along with more complex data, such as lists, dictionaries and even objects. We're going to examine the different types of data that can be assigned to a variable along with various functions that are provided by Python for manipulating the data.

Built-in datatypes

Python has a number of built-in data-types. The first built-in type that we will discuss is the `string` data-type. We've already seen several examples of `string` variables, but these types of variables can be manipulated in a lot of ways, so let's take a closer look at this data-type.

Strings

Strings are ordered collections of characters used to store and represent text-based information. This is a rather dry way of saying that string variables hold text. String variables are surrounded by single or double quotes when being assigned to a variable. Examples could include a name, feature class name, a `Where` clause, or anything else that can be encoded as text.

String manipulation

Strings can be manipulated in a number of ways in Python. String concatenation is one of the more commonly used functions and is simple to accomplish. The + operator is used with string variables on either side of the operator to produce a new string variable that ties the two string variables together:

```
shpStreets = "c:\\GISData\\Streets" + ".shp"
print shpStreets
```

Running this code example produces the following result:

```
>>>c:\GISData\Streets.shp
```

String equality can be tested using Python's == operator, which is simply two equal signs placed together. Don't confuse the equality operator with the assignment operator, which is a single equal to sign. The equality operator tests two variables for equality, while the assignment operator assigns a value to a variable:

```
firstName = "Eric"
lastName = "Pimpler"
firstName == lastname
```

Running this code example produces the following result:

```
>>>False
```

Strings can be tested for containment using the `in` operator, which returns `True` if the first operand is contained in the second.

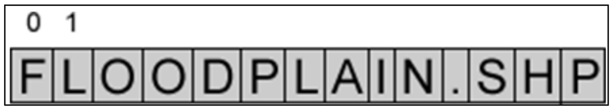
```
fcName = "Floodplain.shp"
print ".shp" in fcName
>>>True
```

I briefly mentioned that strings are an ordered collection of characters. What does this mean? It simply means that we can access individual characters or a series of characters from the string. In Python, this is referred to as **indexing** in the case of accessing an individual character, and **slicing** in the case of accessing a series of characters.

Characters in a string are obtained by providing the numeric offset contained within square brackets after a string. For example, you could obtain the first string character in the `fc` variable by using the syntax `fc[0]`. Negative offsets can be used to search backwards from the end of a string. In this case, the last character in a string is stored at index `-1`. Indexing always creates a new variable to hold the character:

```
fc = "Floodplain.shp"
print fc[0]
>>>'F'
print fc[10]
>>>'.'
print fc[13]
>>>'p'
```

The following image illustrates how strings are an ordered collection of characters with the first character occupying position **0**, the second character occupying position **1**, and each successive character occupying the next index number:



While string indexing allows you to obtain a single character from a string variable, string slicing enables you to extract a contiguous sequence of strings. The format and syntax is similar to indexing, but with the addition of a second offset, which is used to tell Python how many characters to return.

The following code example provides an example of string slicing. The `theString` variable has been assigned a value of `Floodplain.shp`. To obtain a sliced variable with the contents of `Flood`, you would use the `theString[0:5]` syntax:

```
theString = "Floodplain.shp"
print theString[0:5]
>>>Flood
```

Python slicing returns the characters beginning with the first offset up to, but not including, the second offset. This can be particularly confusing for new Python programmers and is a common source of error. In our example, the returned variable will contain the characters `Flood`. The first character, which occupies position `0`, is `F`. The last character returned is index `4`, which corresponds to the character `d`. Notice that index number `5` is not included since Python slicing only returns characters up to but not including the second offset.

Either of the offsets can be left off. This in effect creates a wild card. In the case of `theString[1:]`, you are telling Python to return all characters starting from the second character to the end of the string. In the second case, `theString[:-1]`, you are telling Python to start at character zero and return all characters except the last.

Python is an excellent language for manipulating strings and there are many additional functions that you can use to process this type of data. Most of these are beyond the scope of this text, but in general all of the following string manipulation functions are available:

- String length
- Casing functions for conversion to upper and lower case
- Removal of leading and trailing whitespace
- Finding a character within a string
- Replacement of text
- Splitting into a list of words based on a delimiter
- Formatting

Numbers

Python also has built-in support for numeric data including `int`, `long`, `float`, and `complex` values. Numbers are assigned to variables in much the same way as strings, with the exception that you do not enclose the value in quotes and obviously it must be a numeric value.

Python supports all the commonly used numeric operators including addition, subtraction, multiplication, division, and modulus or remainder. In addition, functions for returning the absolute value, conversion of strings to numeric datatypes, and rounding are also available.

Although Python provides a few built-in mathematical functions, the `math` module can be used to access a wide variety of more advanced `math` functions. To use these functions, you must specifically import the `math` module as follows:

```
import math
```

Functions provided by the `math` module include those for returning the ceiling and floor of a number, the absolute value, trigonometric functions, logarithmic functions, angular conversion, and hyperbolic functions.

Lists

A third built-in datatype provided by Python is the `list`. A list is an ordered collection of objects that can hold any type of data supported by Python as well as being able to hold multiple datatypes at the same time. This could be numbers, strings, other lists, dictionaries, or objects. So, for instance, a list variable could hold numeric and string data at the same time. Lists are zero-based, with the first element in the list occupying position `0`. Each successive object in the list is incremented by one. In addition, lists have the special capability of dynamically growing and shrinking.

Lists are created by assigning a series of values enclosed by brackets. To pull a value from a list, simply use an integer value in brackets along with the variable name. The following code example provides an illustration of this. You can also use slicing with lists to return multiple values. **Slicing** a list always returns a new list variable.

```
fcList = ["Hydrants", "Water Mains", "Valves", "Wells"]
fc = fcList[0]
print fc
>>>Hydrants
fc = fcList[3]
print fc
>>>Wells
```

Lists are dynamic in nature, enabling them to grow, shrink, and change contents. This is all done without the need to create a new copy of the list. Changing values in a list can be accomplished either through indexing or slicing. Indexing allows you to change a single value, while slicing allows you to change multiple list items.

Lists have a number of methods that allow you to manipulate the values that are part of the list. You can sort the contents of the list in either an ascending or descending order through the use of the `sort()` method. Items can be added to a list with the `append()` method, which adds an object to the end of the list, and with the `insert()` method which inserts an object at a position within the list. Items can be removed from a list with the `remove()` method which removes the first occurrence of a value from the list, or `the pop()` method which removes and returns the object last added to the list. The contents of the list can also be reversed with the `reverse()` method.

Tuples

Tuples are similar to lists but with some important differences. Just like lists, tuples contain a sequence of values. The only difference is that tuples can't be changed, and they are referred to with parentheses instead of square brackets. Creating a tuple is as simple as placing a number of comma-separated values inside parentheses, as shown in the following code example:

```
fcTuples = ("Hydrants", "Water Mains", "Valves", "Wells")
```

Like lists, tuple indices start with an index value of `0`. Access to values stored in a tuple occurs in the same way as lists. This is illustrated in the following code example:

```
fcTuples = ("Hydrants", "Water Mains", "Valves", "Wells")
print fcTuples[1]
>>>Water Mains
```

Tuples are typically used in place of a list when it is important for the contents of the structure to be static. You can't insure this with a list, but you can with a tuple.

Dictionaries

Dictionaries are a second type of collection object in Python. They are similar to lists, except that dictionaries are an unordered collection of objects. Instead of fetching objects from the collection through the use of an offset, items in a dictionary are stored and fetched by a key. Each key in a dictionary has an associated value. Similar to lists, dictionaries can grow and shrink in place through the use of methods on the `dictionary` class. In the following code example, you will learn to create and populate a dictionary and see how values can be accessed through the use of a key. Dictionaries are created with the use of curly braces. Inside these braces each key is surrounded by quotes followed by a colon and then a value that is associated with the key. These key/value pairs are separated by commas:

```
##create the dictionary
dictLayers = {'Roads': 0, 'Airports': 1, 'Rail': 2}

##access the dictionary by key
dictLayers['Airports']
>>>1
dictLayers['Rail']
>>>2
```

Basic dictionary operations include getting the number of items in a dictionary, getting a value using the key, determining if a key exists, converting the keys to a list, and getting a list of values. Dictionary objects can be changed, expanded, and shrunk in place. What this means is that Python does not have to create a new `dictionary` object to hold the altered version of the dictionary. Assigning values to a dictionary key is accomplished by stating the key value in brackets and setting it equal to some value.

[Unlike lists, dictionaries can't be sliced due to the fact that their contents are unordered. Should you have the need to iterate over all values in a dictionary, simply use the `keys()` method, which returns a collection of all the keys in the dictionary and which can then be used individually to set or get the value.]

Classes and objects

Classes and objects are a fundamental concept in object-oriented programming. While Python is more of a procedural language, it also supports object-oriented programming. In object-oriented programming, classes are used to create object instances. You can think of classes as blueprints for the creation of one or more objects. Each object instance has the same properties and methods, but the data contained in an object can and usually will differ. Objects are complex datatypes in Python composed of properties and methods, and can be assigned to variables just like any other datatype. Properties contain data associated with an object, while methods are actions that an object can perform.

These concepts are best illustrated with an example. In ArcPy, the `Extent` class is a rectangle specified by providing the coordinate of the lower-left corner and the coordinate of the upper-right corner in map units. The `Extent` class contains a number of properties and methods. Properties include `XMin`, `XMax`, `YMin`, and `YMax`, `spatialReference`, and others. The minimum and maximum of x and y properties provide the coordinates for the extent rectangle. The `spatialReference` property holds a reference to a `SpatialReference` object for the `Extent`. Object instances of the `Extent` class can be used both to set and get the values of these properties through dot notation. An example of this is seen in the following code example:

```
import arcpy
fc = "c:/ArcpyBook/data/TravisCounty/TravisCounty.shp"

# Fetch each feature from the cursor and examine the extent properties
and spatial reference
for row in arcpy.da.SearchCursor(fc, ["SHAPE@"]):
  # get the extent of the county boundary
  ext = row[0].extent
```

```
    # print out the bounding coordinates and spatial reference
    print "XMin: " + ext.XMin
    print "XMax: " + ext.XMax
    print "YMin: " + ext.YMin
    print "YMax: " + ext.YMax
    print "Spatial Reference: " + ext.spatialReference.name
```

Running this script yields the following output:

```
XMin: 2977896.74002
XMax: 3230651.20622
YMin: 9981999.27708
YMax:10200100.7854
Spatial Reference: NAD_1983_StatePlane_Texas_Central_FIPS_4203_Feet
```

The `Extent` class also has a number of methods, which are actions that an object can perform. In the case of this particular object, most of the methods are related to performing some sort of geometric test between the `Extent` object and another geometry. Examples include `contains()`, `crosses()`, `disjoint()`, `equals()`, `overlaps()`, `touches()`, and `within()`.

One additional object-oriented concept that you need to understand is dot notation. **Dot notation** provides a way of accessing the properties and methods of an object. It is used to indicate that a property or method belongs to a particular class.

The syntax for using the dot notation includes an object instance followed by a dot and then the property or method. The syntax is the same regardless of whether you're accessing a property or a method. A parenthesis, and zero or more parameters, at the end of the word following the dot indicates that a method is being accessed. Here are a couple of examples to better illustrate this concept:

```
Property: extent.XMin
Method: extent.touches()
```

Statements

Each line of code that you write with Python is known as a **statement**. There are many different kinds of statements, including those that create and assign data to variables, decision support statements that branch your code based on a test, looping statements that execute a code block multiple times, and others. There are various rules that your code will need to follow as you create the statements that are part of your script. You've already encountered one type of statement: variable creation and assignment.

Decision support statements

The `if/elif/else` statement is the primary decision making statement in Python and tests for a true/false condition. Decision statements enable you to control the flow of your programs. Here are some example decisions that you can make in your code: if the variable holds a point feature class, get the X, Y coordinates; if the feature class name equals `Roads` then get the `Name` field.

Decision statements such as `if/elif/else` test for a true/false condition. In Python, a "true" value means any nonzero number or nonempty object. A "false" value indicates "not true" and is represented in Python with a zero number or empty object. Comparison tests return values of one or zero (true or false). Boolean and/or operators return a true or false operand value.

```
if fcName == 'Roads':
  gp.Buffer_analysis(fc, "c:\\temp\\roads.shp", 100)
elif fcName == 'Rail':
  gp.Buffer_analysis(fc, "c:\\temp\\rail.shp", 50)
else:
  print "Can't buffer this layer"
)
```

Python code must follow certain syntax rules. Statements execute one after another, until your code branches. Branching typically occurs through the use of `if/elif/else`. In addition, the use of looping structures, such as `for` and `while`, can alter the statement flow. Python automatically detects statement and block boundaries, so there is no need for braces or delimiters around your blocks of code. Instead, indentation is used to group statements in a block. Many languages terminate statements with the use of a semicolon, but Python simply uses the end of line character to mark the end of a statement. Compound statements include a "`:`" character. Compound statements follow the pattern: header terminated by a colon. Blocks of code are then written as individual statements and are indented underneath the header.

Statement indentation deserves a special mention as it is critical to the way Python interprets code. As I mentioned, a contiguous section of code is detected by Python through the use of indentation. By default, all Python statements should be left-justified until looping, decision support, `try/except`, and `with` statements are used. This includes `for` and `while` loops, `if/else` statements, `try/except` statements, and `with` statements. All statements indented with the same distance belong to the same block of code until that block is ended by a line less indented.

Looping statements

Looping statements allow your program to repeat lines of code over and over as necessary. `while` loops repeatedly execute a block of statements as long as the test at the top of the loop evaluates to true. When the condition test evaluates to false, Python begins interpreting code immediately after the `while` loop. In the next code example, a value of `10` has been assigned to the variable `x`. The test for the `while` loop then checks to see if `x` is less than `100`. If `x` is less than `100` the current value of `x` is printed to the screen and the value of `x` is incremented by `10`. Processing then continues with the `while` loop test. The second time, the value of `x` will be 20; so the test evaluates to true once again. This process continues until `x` is larger than `100`. At this time, the test will evaluate to false and processing will stop. It is very important that your `while` statements have some way of breaking out of the loop. Otherwise, you will wind up in an infinite loop. An infinite loop is a sequence of instructions in a computer program that loops endlessly, either due to the loop having no terminating condition, having one that can never be met, or one that causes the loop to start over:

```
x = 10
while x < 100:
 print x
 x = x + 10
```

`for` loops execute a block of statements a predetermined number of times. They come in two varieties—a counted loop for running a block of code a set number of times, and a list loop that enables you to loop through all objects in a list. The list loop in the following example executes once for each value in the dictionary and then stops looping:

```
dictLayers = {"Roads":"Line","Rail":"Line","Parks":"Polygon"}
lstLayers = dictLayers.keys()
for x in lstLayers:
  print dictLayers[x]
```

There are times when it will be necessary for you to break out of the execution of a loop. The `break` and `continue` statements can be used to do this. `break` jumps out of the closest enclosing loop while `continue` jumps back to the top of the closest enclosing loop. These statements can appear anywhere inside the block of code.

Try statements

A `try` statement is a complete, compound statement that is used to handle exceptions. Exceptions are a high-level control device used primarily for error interception or triggering. Exceptions in Python can either be intercepted or triggered. When an error condition occurs in your code, Python automatically triggers an exception, which may or may not be handled by your code. It is up to you as a programmer to catch an automatically triggered exception. Exceptions can also be triggered manually by your code. In this case, you would also provide an exception handling routine to catch these manually triggered exceptions.

There are two basic types of try statements: `try/except/else` and `try/finally`. The basic `try` statement starts with a `try` header line followed by a block of indented statements, then one or more optional except clauses that name exceptions to be caught, and an optional `else` clause at the end:

```
import arcpy
import sys

inFeatureClass = arcpy.GetParameterAsText(0)
outFeatureClass = arcpy.GetParameterAsText(1)

try:
  # If the output feature class exists, raise an error

  if arcpy.Exists(inFeatureClass):
    raise overwriteError(outFeatureClass)
  else:
    # Additional processing steps


except overwriteError as e:
  # Use message ID 12, and provide the output feature class
  #  to complete the message.

  arcpy.AddIDMessage("Error", 12, str(e))
```

The `try/except/else` statement works as follows. Once inside a `try` statement, Python marks the fact that you are in a `try` block and knows that any exception condition that occurs at this point will be sent to the various `except` statements for handling. If a matching exception is found, the code block inside the `except` block is executed. The code then picks up below the full `try` statement. The `else` statements are not executed in this case. Each statement inside the `try` block is executed. Assuming that no exception conditions occur, the code pointer will then jump to the `else` statement and execute the code block contained by the `else` statement before moving to the next line of code below the try block.

The other type of `try` statement is the `try/finally` statement which allows for finalization actions. When a `finally` clause is used in a `try` statement, its block of statements always run at the very end, whether an error condition occurs or not.

The `try/finally` statement works as follows. If an exception occurs, Python runs the `try` block, then the `finally` block, and then execution continues past the entire `try` statement. If an exception does not occur during execution, Python runs the `try` block, then the `finally` block. This is useful when you want to make sure an action happens after a code block runs, regardless of whether an error condition occurs. Cleanup operations, such as closing a file or a connection to a database are commonly placed inside a `finally` block to ensure that they are executed regardless of whether an exception occurs in your code:

```
import arcpy
from arcpy import env

try:
  if arcpy.CheckExtension("3D") == "Available":
    arcpy.CheckOutExtension("3D")
  else:
    # Raise a custom exception
    raise LicenseError

  env.workspace = "D:/GrosMorne"
  arcpy.HillShade_3d("WesternBrook", "westbrook_hill", 300)
  arcpy.Aspect_3d("WesternBrook", "westbrook_aspect")

except LicenseError:
  print "3D Analyst license is unavailable"
except:
  print arcpy.GetMessages(2)
finally:
  # Check in the 3D Analyst extension
  arcpy.CheckInExtension("3D")
```

with statements

The `with` statement is handy when you have two related operations that need to be executed as a pair with a block of code in between. A common scenario for using `with` statements is opening, reading, and closing a file. Opening and closing a file are the related operations, and reading a file and doing something with the contents is the block of code in between. When writing geoprocessing scripts with ArcGIS, the new `cursor` objects introduced with version 10.1 of ArcGIS are ideal for using `with` statements. We'll discuss `cursor` objects in great detail in a later chapter, but I'll briefly describe these objects now. Cursors are an in-memory copy of records from the attribute table of a feature class or table. There are various types of cursors. Insert cursors allow you to insert new records, search cursors are a read-only copy of records, and update cursors allow you to edit or delete records. Cursor objects are opened, processed in some way, and closed automatically using a `with` statement.

The closure of a file or cursor object is handled automatically by the `with` statement resulting in cleaner, more efficient coding. It's basically like using a `try/finally` block but with fewer lines of code. In the following code example, the `with` block is used to create a new search cursor, read information from the cursor, and implicitly close the cursor:

```
import arcpy

fc = "c:/data/city.gdb/streets"
```

```
# For each row print the Object ID field, and use the SHAPE@AREA
# token to access geometry properties

with arcpy.da.SearchCursor(fc, ("OID@", "SHAPE@AREA")) as cursor:
  for row in cursor:
    print("Feature {0} has an area of {1}".format(row[0], row[1]))
```

File I/O

You will often find it necessary to retrieve or write information to files on your computer. Python has a built-in object type that provides a way to access files for many tasks. We're only going to cover a small subset of the file manipulation functionality provided, but we'll touch on the most commonly used functions including opening and closing files, and reading and writing data to a file.

Python's `open()` function creates a file object, which serves as a link to a file residing on your computer. You must call the `open()` function on a file before reading and/or writing data to a file. The first parameter for the `open()` function is a path to the file you'd like to open. The second parameter corresponds to a mode, which is typically read (`r`), write (`w`), or append (`a`). A value of `r` indicates that you'd like to open the file for read only operations, while a value of `w` indicates you'd like to open the file for write operations. In the event that you open a file that already exists for write operations, this will overwrite any data currently in the file, so you must be careful with write mode. Append mode (`a`) will open a file for write operations, but instead of overwriting any existing data, it will append the new data to the end of the file. The following code example shows the use of the `open()` function for opening a text file in a read-only mode:

```
f = open('Wildfires.txt','r')
```

After you've completed read/write operations on a file, you should always close the file with the `close()` method.

After a file has been opened, data can be read from it in a number of ways and using various methods. The most typical scenario would be to read data one line at a time from a file through the `readline()` method. `readline()` can be used to read the file one line at a time into a string variable. You would need to create a looping mechanism in your Python code to read the entire file line by line. If you would prefer to read the entire file into a variable, you can use the `read()` method, which will read the file up to the **end of file** (**EOF**) marker. You can also use the `readlines()` method to read the entire contents of a file, separating each line into individual strings, until the EOF is found.

In the following code example, we have opened a text file called `Wildfires.txt` in read-only mode and used the `readlines()` method on the file to read its entire contents into a variable called `lstFires`, which is a Python list containing each line of the file as a separate string value in the list. In this case, the `Wildfire.txt` file is a comma-delimited text file containing the latitude and longitude of the fire along with the confidence values for each file. We then loop through each line of text in `lstFires` and use the `split()` function to extract the values based on a comma as the delimiter, including the latitude, longitude, and confidence values. The latitude and longitude values are used to create a new `Point` object, which is then inserted into the feature class using an insert cursor:

```
import arcpy, os
try:

  arcpy.env.workspace = "C:/data/WildlandFires.mdb"
  # open the file to read
  f = open('Wildfires.txt','r')

  lstFires = f.readlines()
  cur = arcpy.InsertCursor("FireIncidents")

  for fire in lstFires:
    if 'Latitude' in fire:
      continue
    vals = fire.split(",")
    latitude = float(vals[0])
    longitude = float(vals[1])
    confid = int(vals[2])
    pnt = arcpy.Point(longitude,latitude)
    feat = cur.newRow()
    feat.shape = pnt
    feat.setValue("CONFIDENCEVALUE", confid)
    cur.insertRow(feat)
    except:
  print arcpy.GetMessages()
finally:
   del cur
   f.close()
```

Just as is the case with reading files, there are a number of methods that you can use to write data to a file. The `write()` function is probably the easiest to use and takes a single string argument and writes it to a file. The `writelines()` function can be used to write out the contents of a list structure to a file. In the following code example, we have created a list structure called `fcList`, which contains a list of feature classes. We can write this list to a file using the `writelines()` method:

```
outfile = open('c:\\temp\\data.txt','w')
fcList = ["Streams", "Roads", "Counties"]
outfile.writelines(fcList)
```

Summary

In this chapter, we covered some of the fundamental Python programming concepts that you'll need to understand before you can write effective geoprocessing scripts. We began the chapter with an overview of the IDLE development environment for writing and debugging Python scripts. You learned how to create a new script, edit existing scripts, check for syntax errors, and execute the script. We also covered the basic language constructs including importing modules, creating and assigning variables, if/else statements, looping statements, and the various data-types including strings, numbers, Booleans, lists, dictionaries, and objects. You also learned how to read and write text files.

In the next chapter, you will learn the basic techniques for writing geoprocessing scripts for ArcGIS with Python. You'll learn how to use the embedded Python window in ArcGIS Desktop, import the ArcPy package to your scripts, execute `ArcToolbox` tools from your scripts, use the help system when writing scripts, use variables to store data, and access the various ArcPy modules.

2
Writing Basic Geoprocessing Scripts with ArcPy

In this chapter, we will cover the following recipes:

- Using the ArcGIS Python window
- Accessing ArcPy with Python
- Executing tools from a script
- Using ArcGIS Desktop help
- Using variables to store data
- Accessing ArcPy modules with Python

Introduction

Geoprocessing tasks tend to be time consuming and repetitive, and often need to be run on a periodic basis. Frequently, many data layers and functions are involved. The ArcPy Python site package for ArcGIS provides a set of tools and execution environments that can be used to transform your data into meaningful results. Using scripts you can automate your geoprocessing tasks and schedule them to run when it is most convenient for your organization.

ArcGIS provides a geoprocessing framework for the purpose of automating your repetitive GIS tasks through a set of tools and execution environments for these tools. All tools operate on an input dataset, which you supply and transform in some way (depending upon the nature of the tool used) to produce a new output dataset. This new output dataset can then, if necessary, be used as the input dataset to additional geoprocessing tools in a larger workflow for your tasks. There are many tools provided by the ArcGIS geoprocessing framework, each designed to provide specific functionality.

While there are many different environments that you can use to write your geoprocessing scripts with Python, this book will focus on the use of the built-in ArcGIS Python window and the Python IDLE editor.

Using the ArcGIS Python window

In this recipe, you'll learn how to use the ArcGIS Python window. In *Chapter 1*, *Fundamentals of the Python Language for ArcGIS*, you learned how to use the IDLE development environment for Python, so this chapter will give you an alternative for writing your geoprocessing scripts. Either development environment can be used but it is common for people to start writing scripts with the ArcGIS Desktop Python window and then move on to IDLE when scripts become more complex. I should also mention that there are many other development environments that you may want to consider, including PythonWin, Wingware, Komodo, and others. The development environment you choose is really a matter of preference.

Getting ready

The new Python window is an embedded, interactive Python window in ArcGIS Desktop 10, that is ideal for testing small blocks of code, learning Python basics, building quick and easy workflows, and executing geoprocessing tools. However, as your scripts become more complex, you'll quickly find the need for a more robust development environment. By default, IDLE is installed with ArcGIS Desktop, so this is a logical next choice for many. For new programmers, though, the ArcGIS Python window is a great place to start!

The ArcGIS Python window has a number of capabilities in addition to being the location for writing your code. You can save the content of the window to a Python script file on a disk or load an existing Python script into the window. The window can be either pinned or floating. While floating, the window can be expanded or contracted as you wish. The window can also be pinned to various parts of the ArcGIS display. You can also format the font and text colors displayed in the window by right-clicking on the window and selecting **Format**.

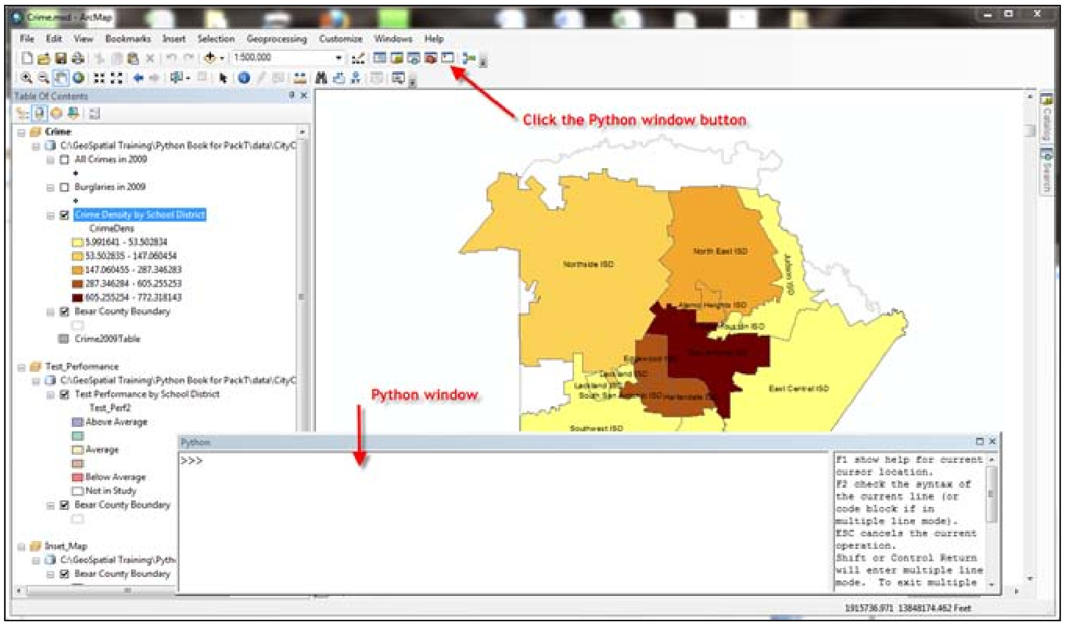
How to do it...

The Python window can be opened by clicking on the Python window button on the main ArcGIS Desktop toolbar.

1. Open `c:\ArcpyBook\Ch2\Crime_Ch2.mxd` with ArcMap.

   > You don't have to specifically open `Crime_Ch2.mxd`. Any map document file can be used with the Python window.

2. Click on the Python window button from the main **ArcMap** toolbar. The Python window will be displayed as shown in the following screenshot. This is a floating window, so you can resize as needed and also dock it at various places on the **ArcMap** interface:



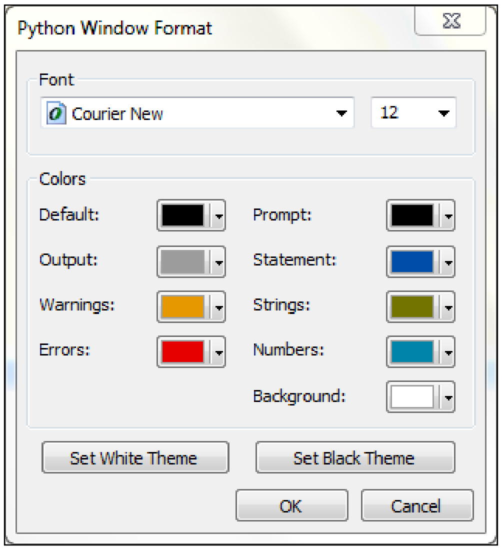
```
>>>
```

```
F1 show help for current
cursor location.
F2 check the syntax of
the current line (or
code block if in
multiple line mode).
ESC cancels the current
operation.
Shift or Control Return
will enter multiple line
mode.  To exit multiple
```

   The Python window is essentially a shell window that allows you to type in statements one line at a time, just after the line input characters `>>>`. On the right-hand side of the divider, you will find a help window.

3. Load an existing script by right-clicking inside the Python window and selecting **Load** from the menu. Navigate to the `c:\ArcpyBook\Ch2` directory and select `ListFields.py` to load a sample script.

You can also format the font and text colors displayed in the window by right-clicking on the window and selecting **Format**. You will be provided with White and Black themes; you can select fonts and colors individually:

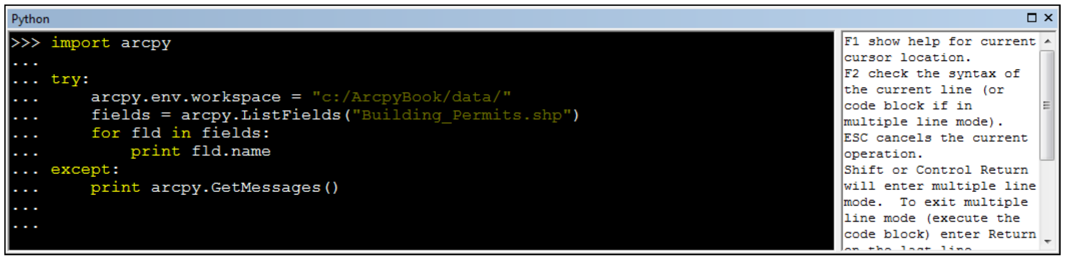

Python Window Format

Font: Courier New 12

Colors

Default: ⬛ Prompt: ⬛
Output: ⬜ Statement: 🟦
Warnings: 🟧 Strings: 🟩
Errors: 🟥 Numbers: 🟦
Background: ⬜

Click on the **Set Black Theme** button to see an example. If you spend a lot of time writing code, you may find that darker themes are easier on your eyes:



```
>>> import arcpy
...
... try:
...     arcpy.env.workspace = "c:/ArcpyBook/data/"
...     fields = arcpy.ListFields("Building_Permits.shp")
...     for fld in fields:
...         print fld.name
... except:
...     print arcpy.GetMessages()
...
...
```

```
F1 show help for current
cursor location.
F2 check the syntax of
the current line (or
code block if in
multiple line mode).
ESC cancels the current
operation.
Shift or Control Return
will enter multiple line
mode.  To exit multiple
line mode (execute the
code block) enter Return
```

Accessing ArcPy with Python

Before you can take advantage of all the geoprocessing functionality provided by ArcPy, you must first import the package into your script. This will always be the first line of code in every geoprocessing script that you write.

Getting ready

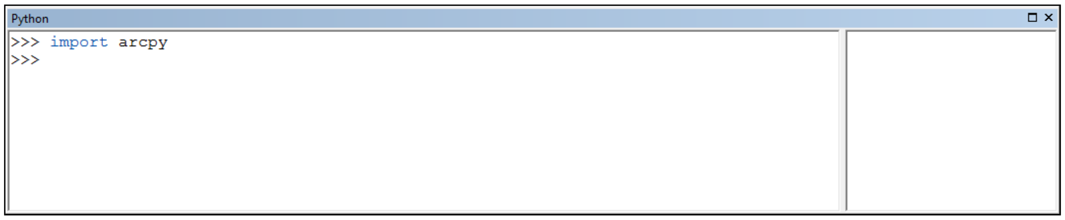
ArcPy is a Python site package that is part of the ArcGIS 10 release, and fully encompasses the functionality provided with the `arcgis scripting` module at ArcGIS 9.2, which further enhances its capabilities. With ArcPy you have access to the geoprocessing tools, extensions, functions, and classes for working with ESRI GIS data. ArcPy provides code-completion and integrated documentation for the modules, classes, and functions. ArcPy can also be integrated with other Python modules to widen the scope of its capabilities. All ArcGIS geoprocessing scripts that you write with Python must first provide a reference to ArcPy.

How to do it...

Follow these steps to import the `arcpy` site package into the ArcGIS Python window:

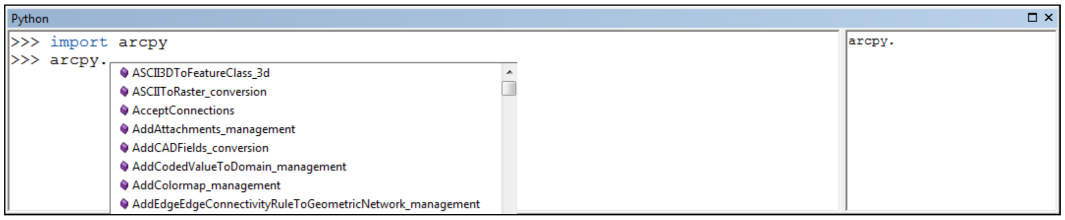
1. Open the `c:\ArcpyBook\Ch2\Crime_Ch2.mxd` file with ArcMap.
2. Click on the Python window button to display a shell window where you can write the Python code.
3. In the Python window, import the `arcpy` package and press the *Enter* key on your keyboard. After each statement that you enter in the Python window, you will press the *Enter* key. You will include this line of code in every single script that you write, so get used to it! This `import` statement is what gives you access to all the functionality provided by ArcPy.

Technically, you don't have to include the `import arcpy` statement when working inside the ArcMap Python window. It's inherent to this window. However, it is required when creating standalone scripts in IDLE, PythonWin, or any other integrated development environment. It's also a good habit to get into as many of your scripts will ultimately be run as standalone scripts.



```
>>> import arcpy
>>>
```

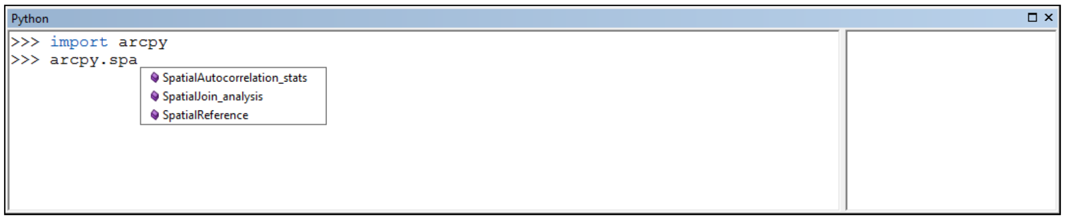
4. ArcPy also provides code-completion functionality that makes your life as a programmer much easier. On the second line, begin by typing `arcpy` and then a dot. ArcPy is an object-oriented package, which means that you access the properties and methods of an object using the dot notation. Notice that a drop-down list of available items is provided. These are the tools, functions, classes, and extensions that are available on this particular object. All objects have their own associated items, so the list of items presented will differ depending on the object that you have currently selected:



```
>>> import arcpy
>>> arcpy.
```

ASCII3DToFeatureClass_3d
ASCIIToRaster_conversion
AcceptConnections
AddAttachments_management
AddCADFields_conversion
AddCodedValueToDomain_management
AddColormap_management
AddEdgeEdgeConnectivityRuleToGeometricNetwork_management

5. This is an auto-filtering list, so as you begin typing the name of the tool, function, class, or extension, the list will be filtered by what you have typed:



```
>>> import arcpy
>>> arcpy.spa
```

SpatialAutocorrelation_stats
SpatialJoin_analysis
SpatialReference

6. You can choose to have the Python window auto-complete the text for you by selecting an item from the list using your mouse or by using the arrow keys to highlight your choice, and then using the *Tab* key to enter the command. This auto-completion feature makes you a faster, more efficient programmer. Not only is it easy to use, but it also dramatically cuts down the number of typos in your code.

How it works...

Once the ArcPy module has been imported, you get access to the geoprocessing tools, extensions, functions, and classes for working with ESRI GIS data. One of the most important aspects of ArcPy is that it provides access to all of the geoprocessing tools available, based on the license level of ArcGIS Desktop currently in use. The tools available to your script will vary depending upon whether you are using the ArcGIS Basic, Standard, or Advanced license level, with the Basic level providing the fewest tools and the Advanced level providing a complete set of tools.

Executing tools from a script

As an ArcGIS user, you have almost certainly used the many available tools in ArcToolbox to accomplish your geoprocessing tasks. Some examples include Clip, Buffer, Feature Class to Feature Class, Add Field, and many more. Your scripts can execute any of the tools found in ArcToolbox. Remember that the tools available to you as a programmer are dependent upon the license level of ArcGIS Destkop that you are using. These tasks can be automated through the creation of a Python script that executes these tools programmatically.

How to do it...

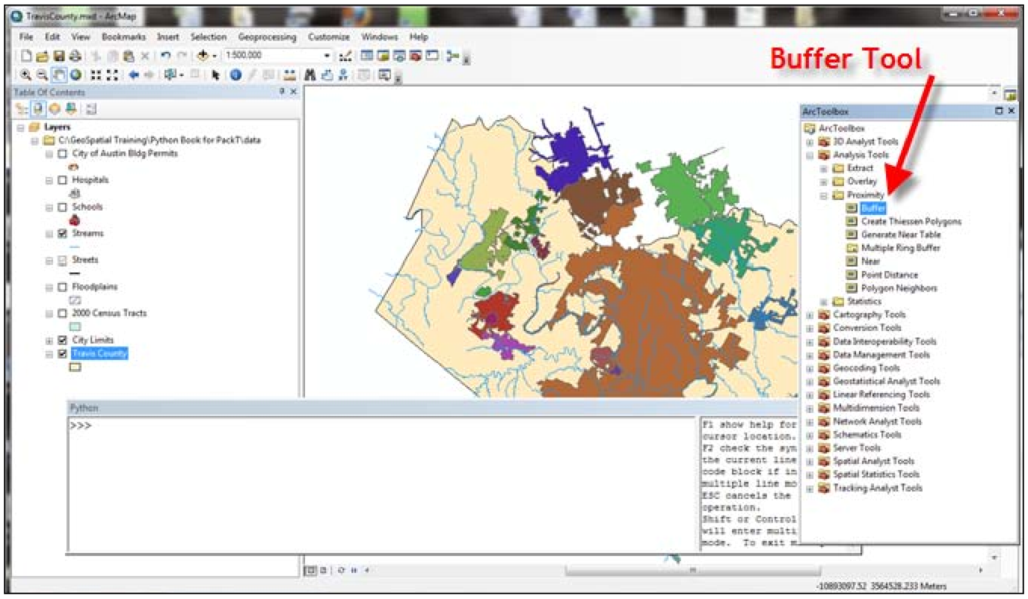
1. Follow these steps to learn how to execute a geoprocessing tool from your script. Open `c:\ArcpyBook\Ch2\TravisCounty.mxd` with ArcMap.
2. Open the Python window.
3. Import the `arcpy` package:

   ```
   import arcpy
   ```

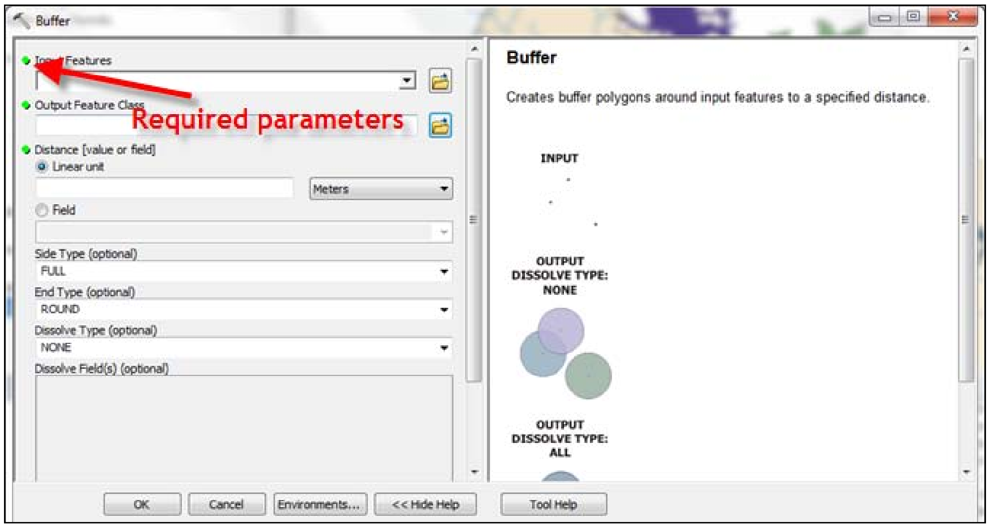
4. Set the workspace. We haven't discussed the `env` class yet. Environment settings for ArcGIS are exposed as properties of this `env` class, which is a part of `arcpy`. One of the properties of the `env` class is workspace, which defines the current working directory for data input and output. In this case, it will be the directory where we'll write the output dataset from our tool:

   ```
   arcpy.env.workspace = "c:/ArcpyBook/data/"
   ```

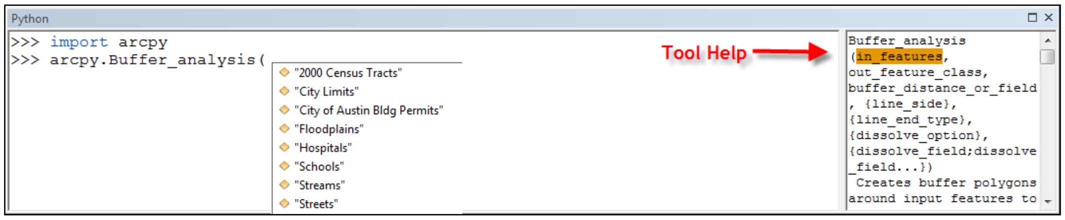
5. We're going to use the **Buffer** tool from the **Analysis Tools** toolbox to buffer the **Streams** layer seen in the active data frame in ArcMap. Open ArcToolbox and find this tool, as shown in the following screenshot:



6. Double-click on the **Buffer** tool to display the interface shown in the following screenshot. Most tools have one or more input parameters that must be supplied for the tool to execute. Whether you're running the tool from the user interface or from a Python script, you must always supply these required parameters:



7. Close the **Buffer** tool.
8. Execute the **Buffer** tool in the Python window. Use the code-completion feature of the Python window to help you as well as the tool help displayed in the window on the right.



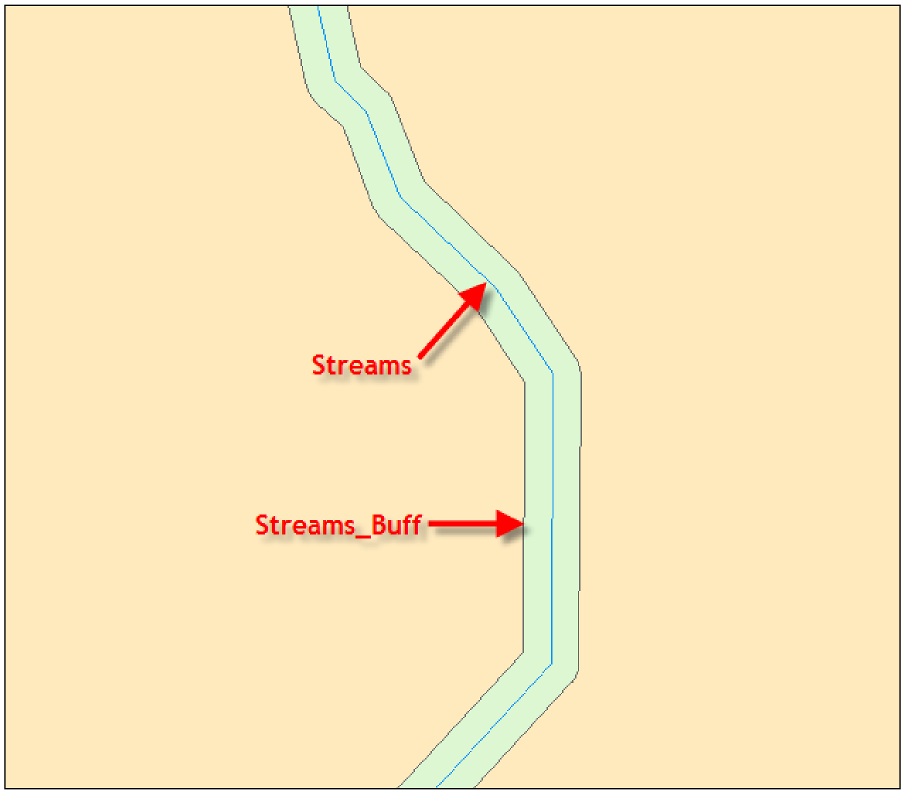
```
>>> import arcpy
>>> arcpy.Buffer_analysis(
```

```
Buffer_analysis
(in_features,
out_feature_class,
buffer_distance_or_field
, {line_side},
{line_end_type},
{dissolve_option},
{dissolve_field;dissolve
_field...})
 Creates buffer polygons
around input features to
```

This will buffer the **Streams** layer by 50 meters to create a new `Streams_Buff` polygon layer:

```
arcpy.Buffer_analysis("Streams", "Streams_Buff", "50 Meters")
```

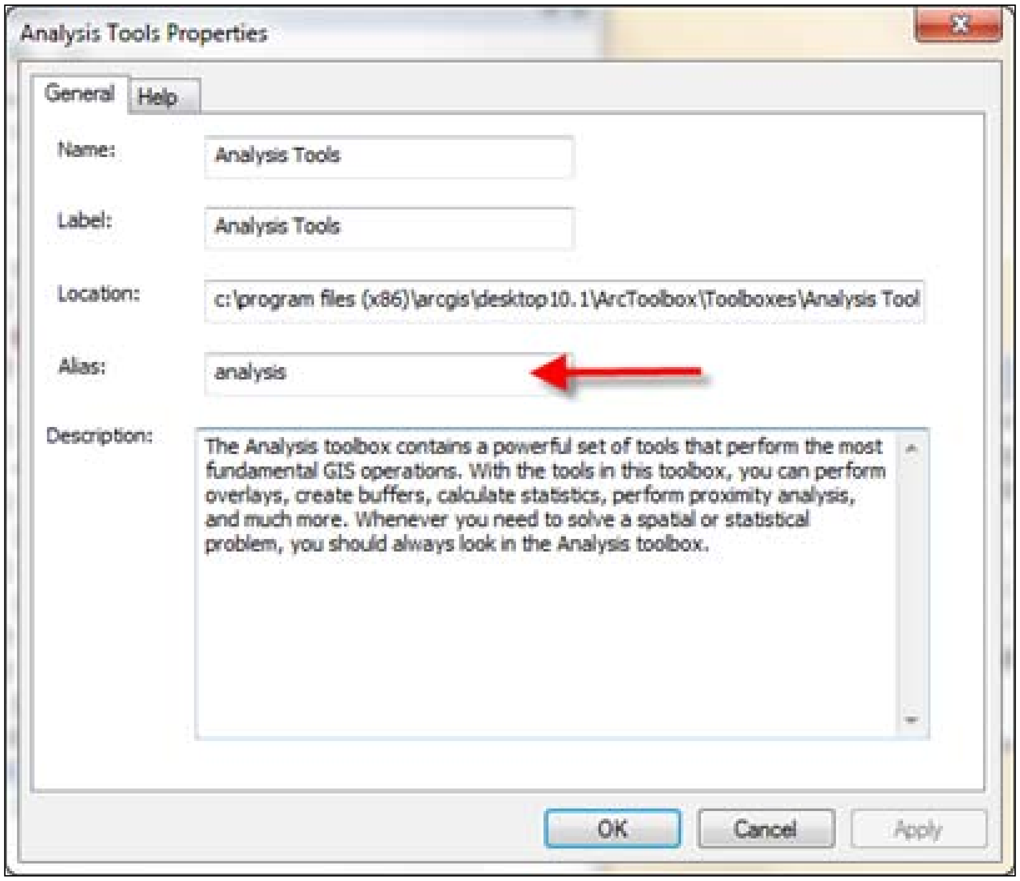
9. Use your ArcMap **zoom** and **pan** tools to get a better look at the output dataset, as shown in the following screenshot:



How it works...

All geoprocessing tools available to your script are defined as dynamic functions from the main `arcpy` object. Each tool that you execute from a script must follow a specific syntax that first defines the tool name, followed by an underscore, and then the alias for the toolbox name. In our example, the **Buffer** tool is located in the **Analysis Tools** toolbox, which has an alias of `analysis`. This is done because it is possible for more than one tool to have the same name. A unique reference for each tool is generated using the syntax `<toolname>_<toolbox_alias>`.

Getting the toolbox alias is easy in ArcGIS Desktop. Find the toolbox associated with the tool and right-click on the toolbox name. Select **Properties**. In the **Properties** dialog box that is displayed, find the **Alias** textbox. You will see the following alias when referring to a particular tool in your geoprocessing scripts:



Name: Analysis Tools

Label: Analysis Tools

Location: c:\program files (x86)\arcgis\desktop10.1\ArcToolbox\Toolboxes\Analysis Tool

Alias: analysis

Description: The Analysis toolbox contains a powerful set of tools that perform the most fundamental GIS operations. With the tools in this toolbox, you can perform overlays, create buffers, calculate statistics, perform proximity analysis, and much more. Whenever you need to solve a spatial or statistical problem, you should always look in the Analysis toolbox.

In addition to the dynamic functions that represent geoprocessing tools, there are many additional functions available on the `arcpy` class that are not geoprocessing tools. Functions for creating cursors, listing datasets, describing datasets, working with environment settings, messaging, and many others are provided. We'll cover many of these functions as we move through the book.

There's more...

Geoprocessing workflows often require multiple steps that require the use of one or more geoprocessing tools. You can more efficiently and effectively develop scripts by first creating an outline for your script. This outline will help you consider the task at hand and identify the geoprocessing tools that will be used. Outlines don't have to be complex undertakings. You can simply draw a diagram of the workflow and then implement your code based on this workflow. The point is to do some planning ahead of time before you actually start coding.

Using ArcGIS Desktop help

The ArcGIS Desktop help system is an excellent resource for obtaining information about any available tool. Each tool is described in detail on a unique page. The help system is available through ArcGIS Desktop or online.

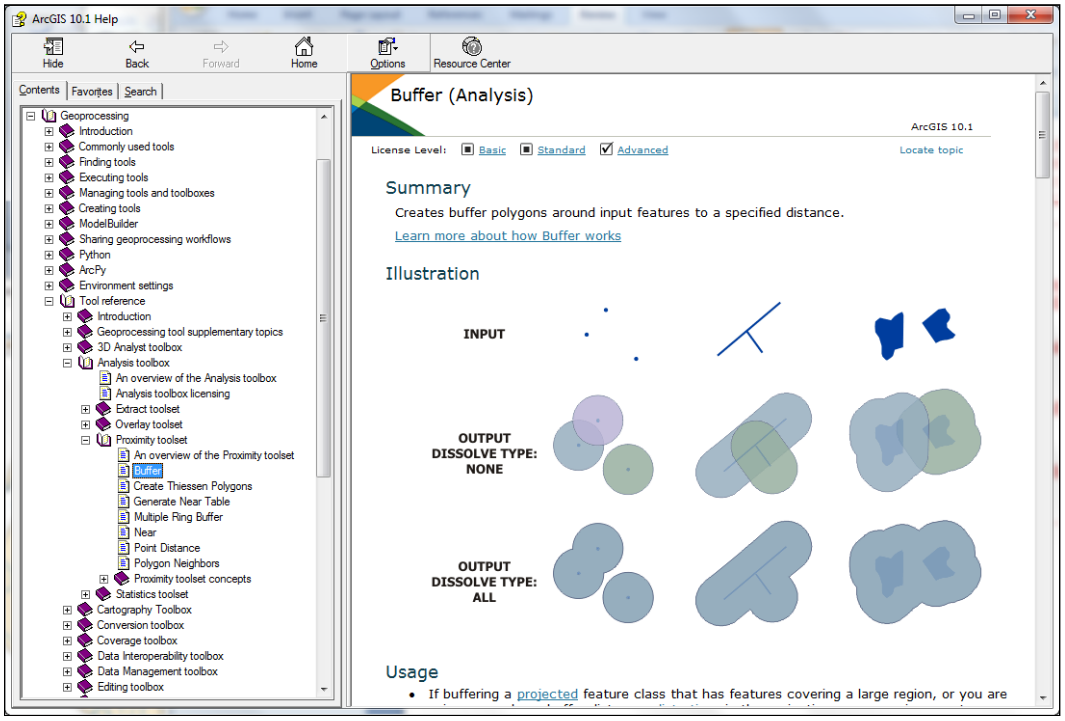
Getting ready

In addition to containing basic descriptive information about each tool, the help system also includes information of interest to Python programmers, including syntax and code examples that provide detailed information about how the tool can be used in your scripts. In this recipe, you will learn how to access the ArcGIS Desktop help system to obtain syntax information and code examples.

How to do it...

Follow these steps to learn how to use the ArcGIS Desktop help system to access syntax information about a tool as well as a code example showing how the tool is used in a script.

1. If necessary, open ArcMap and select **Help** | **ArcGIS Desktop Help** from the main menu.
2. Select the **Contents** tab.
3. Select **Geoprocessing** | **Tool reference**. The tools are grouped according to toolbox just as they are in ArcToolbox.
4. Select **Analysis toolbox** and then **Proximity toolset**.
5. Click on the **Buffer** tool. You should see the **Buffer** tool help displayed, as shown in the following screenshot:



Buffer (Analysis)

ArcGIS 10.1

License Level: Basic Standard Advanced ✓ Locate topic

Summary

Creates buffer polygons around input features to a specified distance.

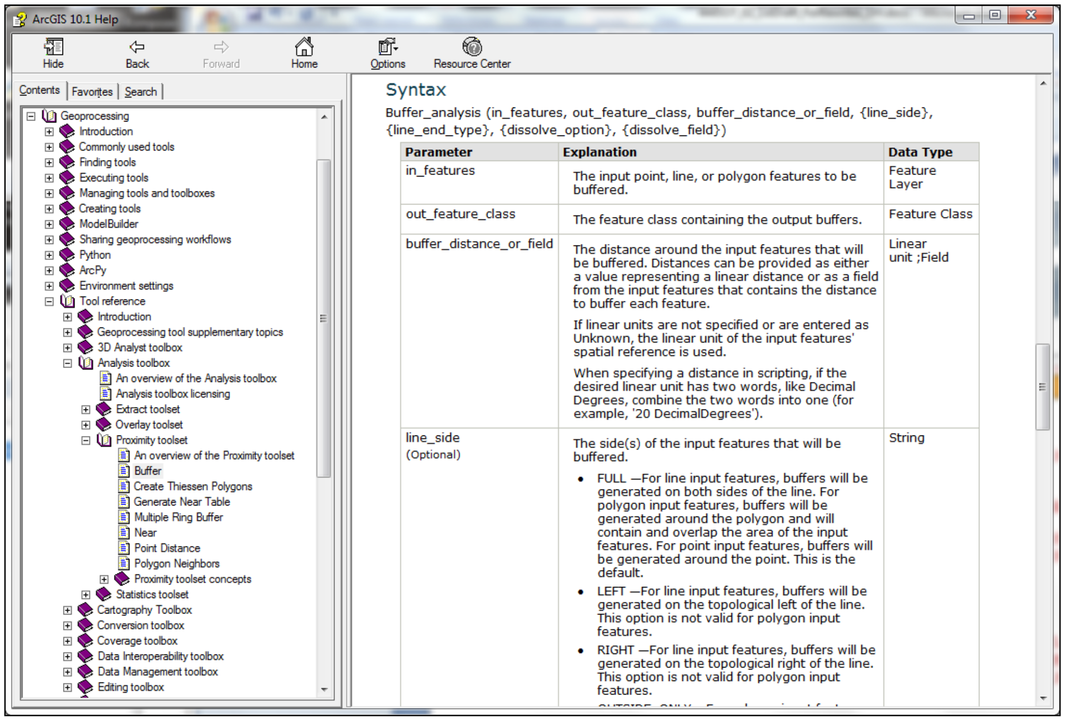
Learn more about how Buffer works

Illustration

Usage

- If buffering a projected feature class that has features covering a large region, or you are

6. Scroll down to the **Syntax** section, as shown in the following screenshot:



Syntax

Buffer_analysis (in_features, out_feature_class, buffer_distance_or_field, {line_side}, {line_end_type}, {dissolve_option}, {dissolve_field})

Parameter	Explanation	Data Type
in_features	The input point, line, or polygon features to be buffered.	Feature Layer
out_feature_class	The feature class containing the output buffers.	Feature Class
buffer_distance_or_field	The distance around the input features that will be buffered. Distances can be provided as either a value representing a linear distance or as a field from the input features that contains the distance to buffer each feature. If linear units are not specified or are entered as Unknown, the linear unit of the input features' spatial reference is used. When specifying a distance in scripting, if the desired linear unit has two words, like Decimal Degrees, combine the two words into one (for example, '20 DecimalDegrees').	Linear unit ;Field
line_side (Optional)	The side(s) of the input features that will be buffered. • FULL —For line input features, buffers will be generated on both sides of the line. For polygon input features, buffers will be generated around the polygon and will contain and overlap the area of the input features. For point input features, buffers will be generated around the point. This is the default. • LEFT —For line input features, buffers will be generated on the topological left of the line. This option is not valid for polygon input features. • RIGHT —For line input features, buffers will be generated on the topological right of the line. This option is not valid for polygon input features.	String

7. This section defines the syntax for how the tool should be called from your script. In this case, the syntax is as follows:

```
Buffer_analysis (in_features, out_feature_class, buffer_distance_
or_field, {line_side}, {line_end_type}, {dissolve_option},
{dissolve_field})
```

8. Scroll down to the **Code Sample** section. Here, you will find one or more code samples showing you how the tool can be used in your scripts. I always recommend taking a look at these samples before writing your scripts.

How it works...

The help system for each tool contains several sections including a summary, illustration, usage, syntax, code sample, environments, related topics, and licensing information. As a programmer, you will primarily be interested in the syntax and code sample sections.

When looking at the syntax section, notice that you call each tool using a combination of the name followed by an underscore and finally the alias for the toolbox where the tool resides. We discussed this briefly in a recipe earlier in this chapter.

Tools typically accept one or more parameters, which are passed into the tool inside parentheses. Parameters can be either required or optional. In this case, the **Buffer** tool includes three required parameters: the input feature, an output feature class, and a distance. Required parameters are listed first and are not enclosed by any special characters. Optional parameters, on the other hand, are enclosed by curly braces and will follow any required parameters. The **Buffer** tool contains several optional parameters including the line side, line end type, dissolve option, and dissolve field. Notice that each of these parameters is enclosed by curly braces. You do not have to include these parameters when calling a tool for it to execute.

You should also examine the syntax information in greater detail to determine the data-type that should be passed in for each parameter. For example, the `buffer_distance_or_field` parameter can accept a data-type that is either a linear unit or a field. Therefore, you can supply a numeric value for this parameter or a `Field` object that represents an attribute field containing distance information.

Always review the syntax for each tool before using it in your code to make sure that you have the right parameters in the right order and of the right data type.

I recommend taking a look at the code samples as well since they will frequently give you a starting point for your script. Often you will find that you can copy and paste at least a portion of the sample into your own script and then alter the script to suit your needs. This can make you a more efficient programmer and for learning purposes it is helpful to take a look at other scripts and examine the script line-by-line to determine how the script works.

Using variables to store data

In *Chapter 1*, *Fundamentals of the Python Language for ArcGIS*, we covered the topic of variables, so you should have a basic understanding of these structures. Variables are given a name and assigned a data value in your scripts. These named variables occupy space in your computer's memory and the data contained within these structures can change while a script is running. After the script has finished the memory space occupied by these variables is then released and can be used for other operations.

Getting ready

When writing geoprocessing scripts with Python there will be many times when you will need to create variables to hold data of one type or another. This data can then be used in your script as input parameters for tools and functions, as intermediate data for internal processing, to hold paths to datasets, and for other reasons. In addition, many of the ArcPy functions and tools also return data that can be stored in a variable for further use in your script. In this recipe, you will learn the basic techniques for creating variables and assigning data to them.

How to do it...

Follow these steps to create a script that contains variables that are hardcoded with values and that are returned from a function:

1. Open IDLE and create a new script window.
2. Save the script to `c:\ArcpyBook\Ch2\WorkingWithVariables.py`.
3. Import the `arcpy` package:
   ```
   import arcpy
   ```
4. Create a variable called `path` and assign a value to it:
   ```
   path = "c:/ArcpyBook/data"
   ```
5. Use the newly-created variable to set the workspace:
   ```
   arcpy.env.workspace = path
   ```
6. Call the `ListFields()` function and assign the returned value to a new variable called `fields`:
   ```
   fields = arcpy.ListFields("Building_Permits.shp")
   ```
7. Start a `for` loop to process each of the field objects contained within the `fields` variable:
   ```
   for fld in fields:
   ```
8. Print the name of each field:
   ```
   print fld.name
   ```
9. The entire script should appear as follows:
   ```
   import arcpy
   path = "c:/ArcpyBook/data"

   arcpy.env.workspace = path
   fields = arcpy.ListFields("Building_Permits.shp")
   for fld in fields:
          print fld.name
   ```
10. Save the script.

How it works...

We created three variables in this script. The first variable, `path`, was created and assigned a hard-coded value with a data path. This is an example of a literal variable, meaning that they literally mean exactly what they say. They are distinguished from variables, whose values are not directly determined by their name. The second variable, `fields`, is created from the returned value of the `ListFields()` function and is a Python `list` object containing one or more `Field` objects. Each `Field` represents a field from the attribute table of a feature class or a standalone table. The final variable is a dynamic variable called `fld`. As the `for` loop cycles through the list returned by the `ListFields()` function, each `Field` is assigned to the `fld` variable. The name of each field is then printed to the screen.

Accessing ArcPy modules with Python

Up to this point, we have covered some basic concepts related to ArcPy. In addition to the basic ArcPy site package, there are a number of modules that you can use to access specific functionality. These modules must be specifically imported into your scripts before you can use the functionality provided. In this recipe, you will learn how to import these modules.

Getting ready

In addition to providing access to tools, functions, and classes, ArcPy also provides several modules. **Modules** are purpose-specific Python libraries containing functions and classes. The modules include a Mapping module (`arcpy.mapping`), a Data Access module (`arcpy.da`), a Spatial Analyst module (`arcpy.sa`), a Geostatistical module (`arcpy.ga`), a Network Analyst module (`arcpy.na`), and a Time module (`arcpy.time`). To use the functions and classes included with each of these modules you must specifically import their associated libraries.

How to do it...

Follow these steps to learn how to use the functions and classes provided by the `arcpy.mapping` module:

1. Open `c:\ArcpyBook\Ch2\Crime_Ch2.mxd` with ArcMap.
2. Open the Python window.
3. Import the `arcpy.mapping` module:
   ```
   import arcpy.mapping as mapping
   ```
4. Get a reference to the current map document (`Crime_Ch2.mxd`):
   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Call the `arcpy.mapping.ListLayers` function:

   ```
   print mapping.ListLayers(mxd)
   ```

   This will return a list of all layers in the map document and print these to the shell window:

   ```
   [<map layer u'City of Austin Bldg Permits'>, <map layer
   u'Hospitals'>, <map layer u'Schools'>, <map layer u'Streams'>,
   <map layer u'Streets'>, <map layer u'Streams_Buff'>, <map layer
   u'Floodplains'>, <map layer u'2000 Census Tracts'>, <map layer
   u'City Limits'>, <map layer u'Travis County'>]
   ```

Access to all the functions and objects available in the Mapping module is done in the same way.

How it works...

Each of the modules provided by ArcPy gives access to functionality that fills a specific purpose. For example, the ArcPy Mapping module provides access to functions that allow you to manage map documents and layer files. The functions and objects in this module all relate in some way to managing these files.

3 Managing Map Documents and Layers

In this chapter, we will cover the following recipes:

- Referencing the current map document
- Referencing map documents on disk
- Accessing a data frame
- Getting a list of layers in a map document
- Restricting the list of layers
- Changing the map extent
- Getting a list of tables
- Adding layers to a map document
- Inserting layers into a map document
- Updating layer symbology
- Updating layer properties

Introduction

The ArcPy mapping module is new to ArcGIS 10 and brings some really exciting features for mapping automation, including the ability to manage map documents and layer files as well as the data within these files. Support is provided for automating map export and printing, for the creation of PDF map books and publication of map documents to ArcGIS Server map services. This is an incredibly useful module for accomplishing many of the day-to-day tasks performed by GIS analysts.

In this chapter, you will learn how to use the ArcPy mapping module to manage map documents and layer files. You will learn how to add and remove geographic layers and tables from map document files, insert layers into data frames, and move layers around within the map document. Finally, you will learn how to update layer properties and symbology.

Referencing the current map document

When running a geoprocessing script from the ArcGIS Python window or a custom script tool, you will often need to get a reference to the map document currently loaded in ArcMap. This is typically the first step in your script before you perform geoprocessing operations against layers and tables in a map document. In this recipe, you will learn how to reference the current map document from your Python geoprocessing script.

Getting ready

Before you can actually perform any operations on a map document file, you need to get a reference to it in your Python script. This is done by calling the `MapDocument()` method on the `arcpy.mapping` module. You can reference either the currently active document or a document at a specific location on disk. To reference the currently active document, you simply supply the keyword `CURRENT` as a parameter to the `MapDocument()` function. This gets the currently active document in ArcMap. The following code example shows how a reference to the currently active document is obtained:

```
mxd = mapping.MapDocument("CURRENT")
```

You can only use the `CURRENT` keyword when running a script from the ArcGIS Python window or a custom script tool. If you attempt to use this keyword when running a script from IDLE or any other development environment, it won't have access to the map document file currently loaded in ArcGIS. I should also point out that the `CURRENT` keyword is not case sensitive. You could just as easily use `current`.

To reference a map document on a local or remote drive, simply supply the path to the map document as well as the map document name as a parameter to `MapDocument()`. For example, you would reference the `crime.mxd` file in the `c:\data` folder with the following reference: `arcpy.mapping.MapDocument("C:/data/crime.mxd")`.

How to do it...

Follow these steps to learn how to access the currently active map document in ArcMap:

1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap.
2. Click on the Python window button from the main ArcMap toolbar.
3. Import the `arcpy.mapping` module by typing the following into the Python window:

   ```
   import arcpy.mapping as mapping
   ```

4. Reference the currently active document (`Crime_Ch3.mxd`) and assign the reference to a variable by typing the following into the Python Window below the first line of code that you added in the last step:

   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Get the title of the map document and print it out to the shell window. When the script executes, the title of the map document will be printed using the Python `print` statement:

   ```
   print mxd.title
   ```

6. Set a new title for map document:

   ```
   mxd.title = "Copy of Crime Project"
   ```

7. Save a copy of the map document file with the saveACopy() method.

   ```
   mxd.saveACopy("c:/ArcpyBook/Ch3/crime_copy.mxd")
   ```

8. Run the script .
9. In ArcMap open the `crime_copy.mxd` file that you just created and select **File** | **Map Document Properties** to view the new title you gave to the map document.

How it works...

The `MapDocument` class has a constructor that creates an instance of this class. In object-oriented programming, an **instance** is also known as an **object**. The constructor for `MapDocument` can accept either the `CURRENT` keyword or a path to a map document file on a local or remote drive. The constructor creates an object and assigns it to the variable `mxd`. You can then access the properties and methods available on this object using dot notation. In this particular case, we've printed out the title of the map document file using the `MapDocument.title` property and we also used the `MapDocument.saveACopy()` method to save a copy of the map document file.

Referencing map documents on a disk

In addition to being able to reference the currently active map document file in ArcMap, you can also access map document files stored on a local or remote drive using the `MapDocument()` method. In this recipe, you'll learn how to access these map documents.

Getting ready

As I mentioned earlier, you can also reference a map document file that resides somewhere on your computer or a shared server. This is done simply by providing a path to the file. This is a more versatile way of obtaining a reference to a map document because it can be run outside the ArcGIS Python window or a custom script tool.

How to do it...

Follow these steps to learn how to access a map document stored on a local or remote drive:

1. Open the IDLE development environment from **Start** | **Programs** | **ArcGIS** | **Python 2.7** | **IDLE**.
2. Create a new IDLE script window by selecting **New** | **New Window** from the IDLE shell window.
3. Import `arcpy.mapping`:
   ```
   import arcpy.mapping as mapping
   ```
4. Reference the copy of the `crime` map document that you created in the last recipe:
   ```
   mxd = mapping.MapDocument("c:/ArcpyBook/Ch3/crime_copy.mxd")
   ```
5. Print the title of the map document:
   ```
   print mxd.title
   ```
6. Run the script to see the following output:
   ```
   Copy of Crime Project
   ```

How it works...

The only difference between this recipe and the last is that we've provided a reference to a map document file on a local or remote drive rather than using the `CURRENT` keyword. This is the recommended way of referencing a map document file unless you know for sure that your geoprocessing script will be run inside ArcGIS, either in the Python window or as a custom script tool.

Accessing a data frame

The ArcMap table of contents is composed of one or more data frames. Each data frame can contain layers and tables. Data frames can be used to filter lists that are returned from the various list functions such as `ListLayers()`. For example, a `DataFrame` object can be used as an input parameter to restrict the layers returned by the ListLayers() function to only those layers within a particular data frame. You can also use a DataFrame object to get or set the current map extent, which can be useful when creating map books. In this recipe you will learn how to access data frames from your Python scripts.

Getting ready

The `ListDataFrames()` function returns a list of `DataFrame` objects. Each data frame can hold layers and tables and can be used to restrict the lists returned by the `ListLayers()` and `ListTablesViews()` functions.

How to do it...

Follow the steps below to learn how to get a list of layers from a map document:

1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap.
2. Click on the Python window button from the main ArcMap toolbar.
3. Import the `arcpy.mapping` module:
   ```
   import arcpy.mapping as mapping
   ```
4. Reference the currently active document (`Crime_Ch3.mxd`) and assign the reference to a variable:
   ```
   mxd = mapping.MapDocument("CURRENT")
   ```
5. Call the `ListDataFrames()` function and pass a reference to the map document along with a wildcard used to find only the data frames that begin with the letter `C`:
   ```
   frames = mapping.ListDataFrames(mxd,"C*")
   ```
6. Start a `for` loop and print out the name of each layer in the map document:
   ```
   for df in frames:
       print df.name
   ```
7. Run the script to see the following output:
   ```
   Crime

   Crime_Inset
   ```

How it works...

The `ListDataFrames()` function returns a list of all the data frames in the ArcMap table of contents. Like any other Python list, you can iterate through the contents of the list using a `for` loop. Inside the `for` loop, each data frame is dynamically assigned to the `df` variable and the name of the data frame is printed to the screen.

Getting a list of layers in a map document

Frequently, one of the first steps in a geoprocessing script is to obtain a list of layers in the map document. Once obtained, your script might then cycle through each of the layers and perform some type of processing. The mapping module contains a `ListLayers()` function that provides the capability of obtaining this list of layers. In this recipe, you will learn how to get a list of layers contained within a map document.

Getting ready

The `arcpy.mapping` module contains various list functions to return lists of layers, data frames, broken data sources, table views, and layout elements. These list functions are normally the first step in a multi-step process, where the script needs to get one or more items from a list for further processing. Each of these list functions returns a Python list, which as you know from earlier in the book, is a highly functional data structure for storing information.

Normally, the list functions are used as part of a multi-step process, where creating a list is only the first step. Subsequent processing in the script will iterate over one or more of the items in this list. For example, you might obtain a list of layers in a map document and then iterate through each layer looking for a specific layer name, which will then be subject to further geoprocessing.

In this recipe, you will learn how to obtain a list of layers from a map document file.

How to do it...

Follow these steps to learn how to get a list of layers from a map document:

1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap.
2. Click on the Python window button from the main ArcMap toolbar.
3. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

4. Reference the currently active document (`Crime_Ch3.mxd`) and assign the reference to a variable:

   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Call the ListLayers() function and pass a reference to the map document:

   ```
   layers = mapping.ListLayers(mxd)
   ```

6. Start a `for` loop and print out the name of each layer in the map document:

   ```
   for lyr in layers:
       print lyr.name
   ```

7. Run the script to see the following output:

   ```
   Burglaries in 2009
   Crime Density by School District
   Bexar County Boundary
   Test Performance by School District
   Bexar County Boundary
   Bexar County Boundary
   Texas Counties
   School_Districts
   Crime Surface
   Bexar County Boundary
   ```

How it works...

The `ListLayers()` function retrieves a list of layers in a map document, a specific data frame, or a layer file. In this case, we passed a reference to the current map document to the `ListLayers()` function, which will retrieve a list of all layers in the map document. The results will be stored in a variable called `layers`, which is a Python list that can be iterated with a `for` loop. This Python list contains one or more `Layer` objects.

There's more...

The `ListLayers()` function is only one of many list functions provided by the `arcpy mapping` module. Each of these functions returns a Python list containing data of some type. Some of the other list functions include `ListTableViews()`, which returns a list of `Table` objects, `ListDataFrames()` which returns a list of `DataFrame` objects, and `ListBookmarks()` which returns a list of bookmarks in a map document. There are additional list functions, many of which we'll cover later in the book.

Restricting the list of layers

In the previous recipe, you learned how to get a list of layers using the `ListLayers()` function. There will be times when you will not want a list of all the layers in a map document, but rather only a subset of the layers. The `ListLayers()` function allows you to restrict the list of layers that is generated. In this recipe, you will learn how to restrict the layers returned using a wildcard and a specific data frame from the ArcMap table of contents.

Getting ready

By default, if you only pass a reference to the map document or layer file, the `ListLayers()` function will return a list of all the layers in these files. However, you can restrict the list of layers returned by this function through the use of an optional wildcard parameter or by passing in a reference to a specific data frame.

> If you're working with a layer file (`.lyr`), you can't restrict layers with a data frame. Layer files don't support data frames.

In this recipe, you will learn how to restrict the list of layers returned by `ListLayers()` through the use of a wildcard and data frame.

How to do it...

Follow these steps to learn how to restrict a list of layers from a map document:

1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap.
2. Click on the Python window button from the main ArcMap toolbar.
3. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

4. Reference the currently active document (`Crime_Ch3.mxd`) and assign the reference to a variable:

   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Get a list of data frames in the map document and search for a specific data frame name of `Crime`. Please note that text strings can be surrounded by either single or double quotes:

   ```
   for df in mapping.ListDataFrames(mxd):
     if (df.name == 'Crime'):
   ```

6. Call the `ListLayers()` function and pass a reference to the map document, a wildcard to restrict the search, and the data frame found in the last step to further restrict the search. The `ListLayers()` function should be indented inside the `if` statement you just created:

   ```
   layers = mapping.ListLayers(mxd,'Burg',df)
   ```

7. Start a `for` loop and print out the name of each layer in the map document.

   ```
   for layer in layers:
       print layer.name
   ```

8. The complete script should appear as follows:

   ```
   import arcpy.mapping as mapping
   mxd = mapping.MapDocument("CURRENT")
   for df in mapping.ListDataFrames(mxd):
     if (df.name == 'Crime'):
           layers = mapping.ListLayers(mxd,"'Burg*',df)
           for layer in layers:
         print layer.name
   ```

9. Run the script to see the following output:

   ```
   Burglaries in 2009
   ```

How it works...

As you learned in a previous recipe, the `ListDataFrames()` function is another list function provided by `arcpy mapping`. This function returns a list of all the data frames in a map document. We then loop through each of the data frames returned by this function, looking for a data frame that has the name `Crime`. If we do find a data frame that has this name, we call the `ListLayers()` function, passing in the optional wildcard value of `Burg*` as the second parameter, and a reference to the `Crime` data frame. The wildcard value passed in as the second parameter accepts any number of characters and an optional wildcard character (`*`).

In this particular recipe, we're searching for all the layers that begin with the characters `Burg` and that have a data frame name of `Crime`. Any layers found matching these restrictions are then printed. Keep in mind that all we're doing in this case is printing the layer names, but in most cases, you would be performing additional geoprocessing through the use of tools or other functions.

Changing the map extent

There will be many occasions when you will need to change the map extent. This is frequently the case when you are automating the map production process and need to create many maps of different areas or features. There are a number of ways that the map extent can be changed with `arcpy`. But, for this recipe, we'll concentrate on using a definition expression to change the extent.

Getting ready

The `DataFrame` class has an `extent` property that you can use to set the geographic extent. This is often used in conjunction with the `Layer.definitionQuery` property that is used to define a definition query for a layer. In this recipe, you will learn how to use these objects and properties to change the map extent.

How to do it...

Follow these steps to learn how to get a list of layers from a map document:

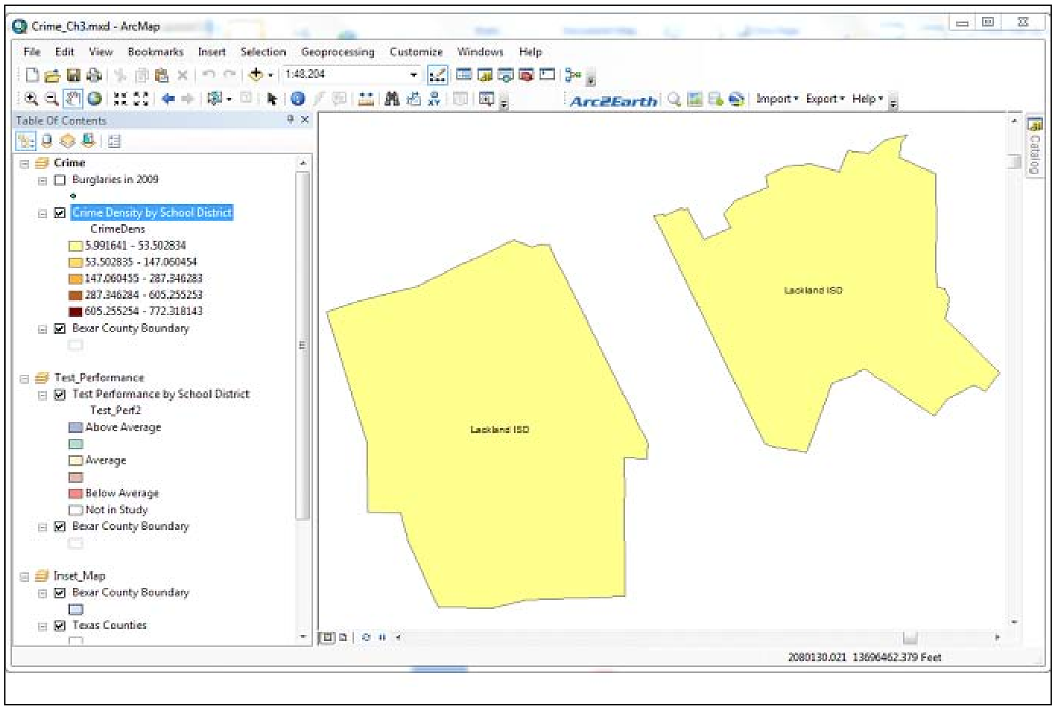
1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap.
2. Click on the Python window button from the main ArcMap toolbar.
3. Import the `arcpy.mapping` module:
   ```
   import arcpy.mapping as mapping
   ```
4. Reference the currently active document (`Crime_Ch3.mxd`), and assign the reference to a variable:
   ```
   mxd = mapping.MapDocument("CURRENT")
   ```
5. Create a `for` loop that will loop through all the data frames in the map document:
   ```
   for df in mapping.ListDataFrames(mxd):
   ```
6. Find the data frame called `Crime` and a specific layer that we'll apply the definition query against:
   ```
   if (df.name == 'Crime'):
     layers = mapping.ListLayers(mxd,'Crime Density by
     School District',df)
   ```
7. Create a `for` loop that will loop through the layers. There will only be one, but we'll create the loop anyway. In the `for` loop, create a definition query and set the new extent of the data frame:

```
for layer in layers:
  query = '"NAME" = \'Lackland ISD\''
  layer.definitionQuery = query
  df.extent = layer.getExtent()
```

8. The entire script should appear as follows:

```
import arcpy.mapping as mapping
mxd = mapping.MapDocument("CURRENT")
for df in mapping.ListDataFrames(mxd):
  if (df.name == 'Crime'):
    layers = mapping.ListLayers(mxd,'Crime Density by School
    District',df)
  for layer in layers:
      query = '"NAME" = \'Lackland ISD\''
      layer.definitionQuery = query
      df.extent = layer.getExtent()
```

9. Save and run the script. The extent of the data view should update to visualize only the features matching the definition expression, as shown in the following screenshot:



How it works...

This recipe used a definition query on a layer to update the map extent. Near the end of the script, you created a new variable called `query` that holds the definition expression. The definition expression is set up to find school districts with a name of `Lackland ISD`. This query string is then applied to the `definitionQuery` property. Finally, the `df.extent` property is set to the returned value of `layer.getExtent()`.

Getting a list of tables

The `arcpy.mapping` module also has a `ListTableViews()` function that you can use to obtain a list of standalone tables that are contained within a map document. In this recipe, you will learn how to use the `ListTableViews()` function to create this list of tables.

Getting ready

In addition to providing the ability to generate a list of layers in a map document or data frame, the `arcpy mapping` module also provides a `ListTableViews()` function that generates a list of tables.

`ListTableViews()` only works with map document files and the data frames contained within. Layer files do not have the capability of holding tables.

How to do it...

Follow these steps to learn how to get a list of standalone tables in a map document:

1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap.
2. Click on the Python window button from the main ArcMap toolbar.
3. Import the `arcpy.mapping` module:
   ```
   import arcpy.mapping as mapping
   ```
4. Reference the currently active document (`Crime_Ch3.mxd`), and assign the reference to a variable:
   ```
   mxd = mapping.MapDocument("CURRENT")
   ```
5. Generate a list of tables in the map document:
   ```
   for tableView in mapping.ListTableViews(mxd):
       print tableView.name
   ```
6. Run the script to see the following output:.
   ```
   Crime2009Table
   ```

How it works...

The `ListTableViews()` function is very similar to the other list functions provided by `arcpy.mapping`. As was the case with `ListLayers()`, the `ListTableViews()` function accepts a reference to a map document (but not a layer file), along with an optional wildcard and data frame parameters. The output is a list of tables that can be iterated with a `for` loop.

Adding layers to a map document

There will be many situations where you will need to add a layer to a map document. The mapping module provides this functionality through the `AddLayer()` function. In this recipe, you will learn how to add a layer to a map document using this function.

Getting ready

`arcpy.mapping` provides the ability to add layers or group layers into an existing map document file. You can take advantage of the ArcMap "auto-arrange" functionality, which automatically places a layer in the data frame for visibility. This is essentially the same functionality provided by the **Add Data** button in ArcMap, which positions a layer in the data frame based on geometry type and layer weight rules.

Layers can't be added to a layer file (`.lyr`).

When adding a layer to a map document, the layer must reference an existing layer found in a layer file on disk, the same map document and data frame, the same map document with a different data frame, or a completely separate map document. A layer can be either a layer in a map document or a layer in a `.lyr` file. To add a layer to a map document, you must first create an instance of the `Layer` class and then call the `AddLayer()` function, passing in the new layer along with the data frame where it should be placed and rules for how it is to be positioned.

How to do it...

Follow these steps to learn how to add a layer to a map document:

1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap.
2. Click on the Python window button from the main ArcMap toolbar.
3. Import the `arcpy.mapping` module:

```
import arcpy.mapping as mapping
```

4. Reference the currently active document (`Crime_Ch3.mxd`), and assign the reference to a variable:

   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Get a reference to the `Crime` data frame, which is the first data frame in the list returned by `ListDataFrames()`. The `[0]` specified at the end of the code gets the first data frame returned from the `ListDataFrames()` method, which returns a list of data frames. Lists are zero-based, so to retrieve the first data frame we provide an index of `0`.

   ```
   df = mapping.ListDataFrames(mxd)[0]
   ```

6. Create a `Layer` object that references a `.lyr` file.

   ```
   layer = mapping.Layer(r"C:\ArcpyBook\data\School_Districts.lyr")
   ```

7. Add the layer to the data frame:

   ```
   mapping.AddLayer(df,layer,"AUTO_ARRANGE")
   ```

8. Run the script. The `School_District.lyr` file will be added to the data frame, as shown in the following screenshot:

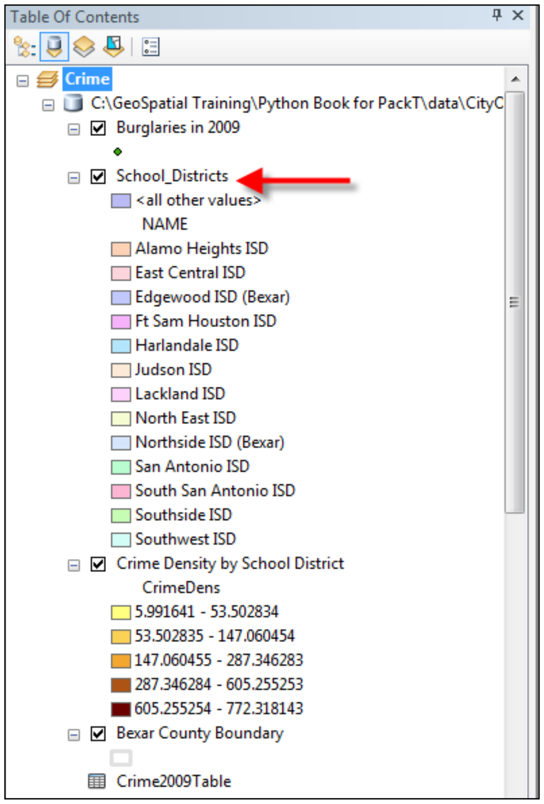


Table Of Contents

Crime
C:\GeoSpatial Training\Python Book for PackT\data\CityC
☑ Burglaries in 2009
●
☑ School_Districts
▨ <all other values>
NAME
▨ Alamo Heights ISD
▨ East Central ISD
▨ Edgewood ISD (Bexar)
▨ Ft Sam Houston ISD
▨ Harlandale ISD
▨ Judson ISD
▨ Lackland ISD
▨ North East ISD
▨ Northside ISD (Bexar)
▨ San Antonio ISD
▨ South San Antonio ISD
▨ Southside ISD
▨ Southwest ISD
☑ Crime Density by School District
CrimeDens
▨ 5.991641 - 53.502834
▨ 53.502835 - 147.060454
▨ 147.060455 - 287.346283
▨ 287.346284 - 605.255253
▨ 605.255254 - 772.318143
☑ Bexar County Boundary
□
Crime2009Table

How it works...

In the first two lines, we simply reference the `arcpy.mapping` module and get a reference to the currently active map document. Next, we create a new variable called `df`, which holds a reference to the `Crime` data frame. This is obtained through the `ListDataFrames()` function that returns a list of data frame objects. We then use list access to return the first item in the list, which is the `Crime` data frame. A new `Layer` instance, called `layer` is then created from a `layer` file stored on disk. This `layer` file is called `School_Districts.lyr`. Finally, we call the `AddLayer()` function, passing in the data frame where the layer will reside along with a reference to the layer, and a parameter indicating that we would like to use the **auto-arrange** feature. In addition to allowing ArcMap to automatically place the layer into the data frame using auto-arrange, you can also specifically place the layer at either the top or bottom of the data frame or a group layer using the `BOTTOM` or `TOP` position.

There's more...

In addition to providing the capability of adding a layer to a map document, `arcpy.mapping` also provides an `AddLayerToGroup()` function, which can be used to add a layer to a group layer. The layer can be added to the top or bottom of the group layer or you can use auto-arrange for placement. You may also add layers to an empty group layer. However, just as with regular layer objects, group layers cannot be added to a layer file.

Layers can also be removed from a data frame or group layer. `RemoveLayer()` is the function used to remove a layer or group layer. In the event that two layers have the same name, only the first is removed unless your script is set up to iterate.

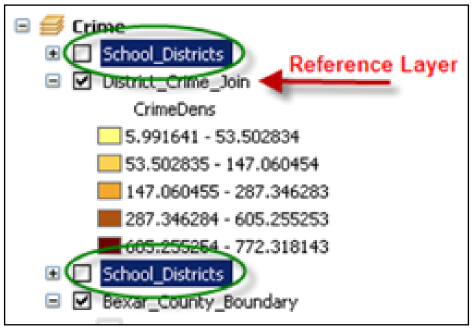
Inserting layers into a map document

The `AddLayer()` function can be used to add a layer to a map document either through auto-arrange or as the first or last layer in a data frame. However, it doesn't provide the control you need for inserting a layer in a specific position within a data frame. For this added control, you can use the `InsertLayer()` function. In this recipe, you will learn how to control the placement of layers added to a data frame.

Getting ready

The `AddLayer()` function simply adds a layer into a data frame or a group layer, and places the layer automatically using auto-arrange or specific placement at the top or bottom of the data frame or group layer. The `InsertLayer()` method allows for more precise positioning of a new layer into a data frame or a group layer. It uses a reference layer to specify a location and the layer is added either before or after the reference layer, as specified in your code. Since `InsertLayer()` requires the use of a reference layer, you can't use this method on an empty data frame.

This is illustrated in the following screenshot, where `District_Crime_Join` is the reference layer and `School_Districts` is the layer to be added. The `School_Districts` layer can be placed either before or after the reference layer using `InsertLayer()`.



How to do it...

Follow these steps to learn how to use `InsertLayer()` to insert a layer into a data frame:

1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap.
2. Click on the Python window button from the main ArcMap toolbar.
3. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

4. Reference the currently active document (`Crime_Ch3.mxd`), and assign the reference to a variable:

   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Get a reference to the `Crime` data frame:

   ```
   df = mapping.ListDataFrames(mxd, "Crime")[0]
   ```

6. Define the reference layer:

   ```
   refLayer = mapping.ListLayers(mxd, "Burglaries*", df)[0]
   ```

7. Define the layer to be inserted relative to the reference layer:

   ```
   insertLayer = mapping.Layer(r"C:\ArcpyBook\data\CityOfSanAntonio.
   gdb\Crimes2009")
   ```

8. Insert the layer into the data frame:

   ```
   mapping.InsertLayer(df,refLayer,insertLayer,"BEFORE")
   ```

9. Run the script. The `Crimes2009` feature class will be added as a layer to the data frame as seen in the following screenshot:

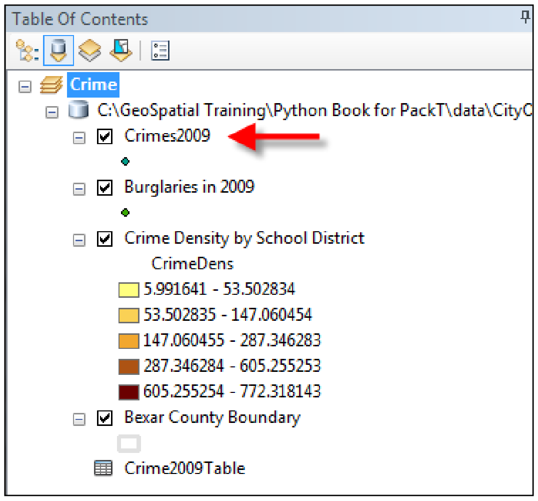


Table Of Contents

Crime
C:\GeoSpatial Training\Python Book for PackT\data\CityC
Crimes2009
Burglaries in 2009
Crime Density by School District
CrimeDens
🟨 5.991641 - 53.502834
🟨 53.502835 - 147.060454
🟧 147.060455 - 287.346283
🟫 287.346284 - 605.255253
🟥 605.255254 - 772.318143
Bexar County Boundary
Crime2009Table

How it works...

After obtaining references to the `arcpy.mapping` module, current map document file, and the `Crime` data frame, our script then defines a reference layer. In this case, we use the `ListLayers()` function with a wildcard of `Burglaries*`, and the `Crime` data frame to restrict the list of layers returned to only one item. This item will be the **Burglaries in 2009** layer. We use Python list access with a value of `0` to retrieve this layer from the list and assign it to a `Layer` object. Next, we define the insert layer, which is a new `Layer` object that references the **Crimes2009** feature class from the `CityOfSanAntonio` geodatabase database. Finally, we call the `InsertLayer()` function passing in the data frame, reference layer, layer to be inserted, and keyword indicating that the layer to be inserted should be placed before the reference layer. This is illustrated in the following screenshot:

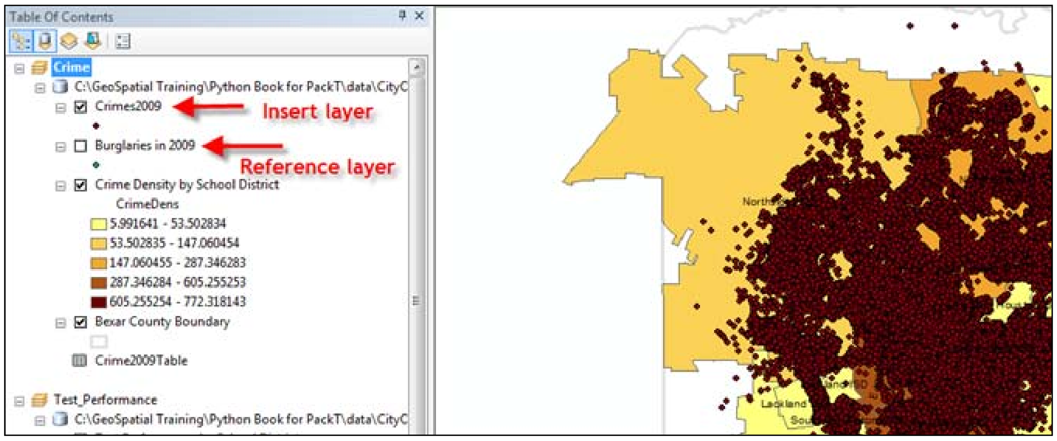


Table Of Contents

Crime
C:\GeoSpatial Training\Python Book for PackT\data\CityC
Crimes2009
Burglaries in 2009
Crime Density by School District
CrimeDens
🟨 5.991641 - 53.502834
🟨 53.502835 - 147.060454
🟧 147.060455 - 287.346283
🟫 287.346284 - 605.255253
🟥 605.255254 - 772.318143
Bexar County Boundary
Crime2009Table
Test_Performance
C:\GeoSpatial Training\Python Book for PackT\data\CityC

There's more...

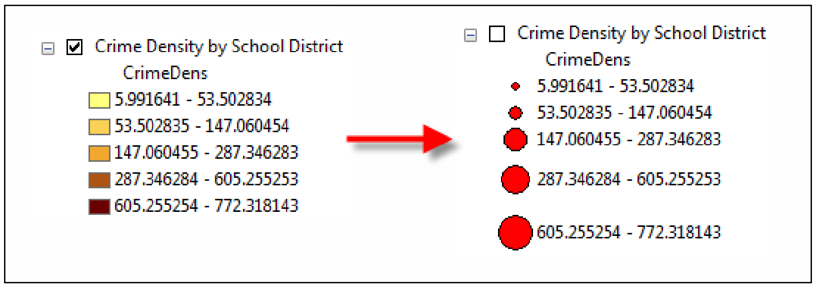
You can also reposition a layer that is already in a data frame or a group layer. The `MoveLayer()` function provides the ability to reposition the layer within a data frame or a group layer. Movement of a layer must be within the same data frame. You can't move a layer from one data frame to another. Just as with `InsertLayer()`, `MoveLayer()` uses a reference layer to reposition the layer.

Updating layer symbology

There may be times when you will want to change the symbology of a layer in a map document. This can be accomplished through the use of the `UpdateLayer()` function, which can be used to change the symbology of a layer as well as various properties of a layer. In this recipe, you will use the `UpdateLayer()` function to update the symbology of a layer.

Getting ready

The `arcpy.mapping` module also gives you the capability of updating layer symbology from your scripts by using the `UpdateLayer()` function. For example, you might want your script to update a layer's symbology from a graduated color to a graduated symbol, as illustrated in the following screenshot. `UpdateLayer()` can also be used to update various layer properties, but the default functionality is to update the symbology. Because `UpdateLayer()` is a robust function capable of altering both symbology and properties, you do need to understand the various parameters that can be supplied as an input.



How to do it...

Follow these steps to learn how to update the symbology of a layer using `UpdateLayer()`:

1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap.
2. Click on the Python window button from the main ArcMap toolbar.

3. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

4. Reference the currently active document (`Crime_Ch3.mxd`), and assign the reference to a variable:

   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Get a reference to the `Crime` data frame:

   ```
   df = mapping.ListDataFrames(mxd, "Crime")[0]
   ```

6. Define the layer that will be updated:

   ```
   updateLayer = mapping.ListLayers(mxd,"Crime Density by School
   District",df)[0]
   ```

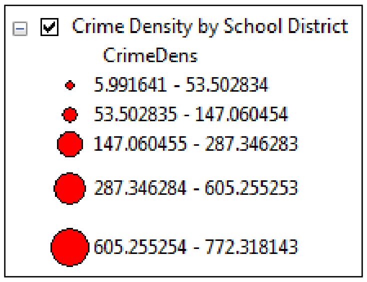
7. Define the layer that will be used to update the symbology:

   ```
   sourceLayer = mapping.Layer(r"C:\ArcpyBook\data\
   CrimeDensityGradSym.lyr")
   ```

8. Call the `UpdateLayer()` function to update the symbology:

   ```
   mapping.UpdateLayer(df,updateLayer,sourceLayer,True)
   ```

9. Run the script. The **Crime Density by School District** layer will now be symbolized with graduated symbols instead of graduated colors, as shown in the following screenshot:



Crime Density by School District
CrimeDens
● 5.991641 - 53.502834
● 53.502835 - 147.060454
● 147.060455 - 287.346283
● 287.346284 - 605.255253
● 605.255254 - 772.318143

How it works...

In this recipe, we used the `UpdateLayer()` function to update the symbology of a layer. We didn't update any properties, but we'll do so in the next recipe. The `UpdateLayer()` function requires that you pass several parameters including a data frame, layer to be updated, and a reference layer from which the symbology will be pulled and applied to update the layer. In our code, the `updateLayer` variable holds a reference to the **Crime Density by School District** layer, which will have its symbology updated. The source layer from which the symbology will be pulled and applied to the update layer is a layer file (`CrimeDensityGradSym.lyr`) containing graduated symbols.

To update the symbology for a layer, you must first ensure that the update layer and the source layer have the same geometry (point, line, and polygon). You also need to check that the attribute definitions are the same, in some cases, depending upon the renderer. For example, graduated color symbology and graduated symbols are based on a particular attribute. In this case, both the layers have polygon geometry and have a `CrimeDens` field containing crime density information.

Once we have references to both layers, we call the `UpdateLayer()` function, passing in the data frame and layers along with a fourth parameter that indicates that we're updating symbology only. We've supplied a `True` value as this fourth parameter, indicating that we only want to update the symbology and not properties.

```
mapping.UpdateLayer(df,updateLayer,sourceLayer,True)
```

There's more...

`UpdateLayer()` also provides the ability to remove one layer and add another layer in its place. The layers can be completely unrelated, so there is no need to ensure the same geometry type and attribute field as you would when redefining the symbology of a layer. This switching of layers essentially executes a call to `RemoveLayer()` and then a call to `AddLayer()` as one operation. To take advantage of this functionality you must set the `symbology_only` parameter to `False`.

Updating layer properties

In the previous recipe, you learned how to update the symbology of a layer. As I mentioned, `UpdateLayer()` can also be used to update various properties of a layer, such as field aliases, query definitions, and others. In this recipe, you will use `UpdateLayer()` to alter various properties of a layer.

Getting ready

You can also use the `UpdateLayer()` function to update a limited number of layer properties. Specific layer properties, such as field aliases, selection symbology, query definitions, label fields, and others, can be updated using `UpdateLayer()`. A common scenario is to have a layer in many map documents that needs to have a specific property changed across all the instances of the layer in all map documents. To accomplish this, you will have to use ArcMap to modify the layer with the appropriate properties and save it to a layer file. This layer file then becomes the source layer, which will be used to update the properties of another layer called `update_layer`. In this recipe, you'll use ArcMap to alter the properties of a layer, save to a layer file (`.lyr`), and then use Python to write a script that uses `UpdateLayer()` to apply the properties to another layer.

How to do it...

Follow these steps to learn how to update layer properties with `UpdateLayer()`:

1. Open `c:\ArcpyBook\Ch3\Crime_Ch3.mxd` with ArcMap. For this recipe, you will be working with the **Burglaries in 2009** feature class, as shown in the following screenshot:

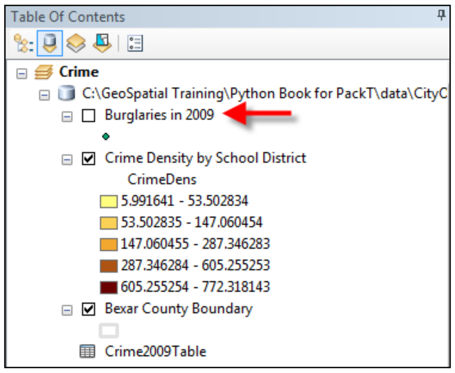
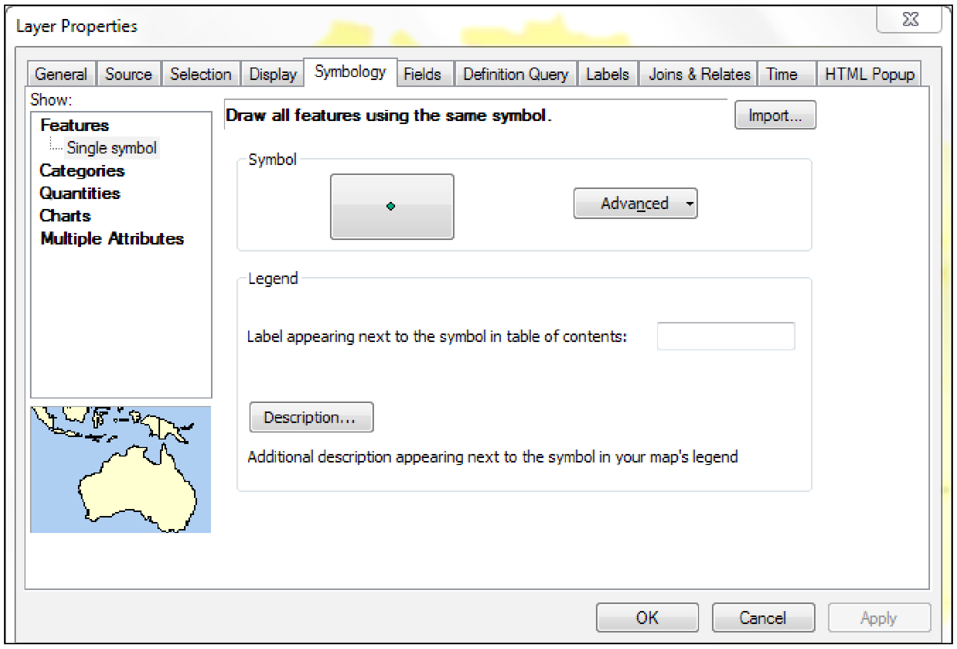


Table Of Contents

- Crime
  - C:\GeoSpatial Training\Python Book for PackT\data\CityC
    - ☐ Burglaries in 2009
      - ●
    - ☑ Crime Density by School District
      - CrimeDens
      - 5.991641 - 53.502834
      - 53.502835 - 147.060454
      - 147.060455 - 287.346283
      - 287.346284 - 605.255253
      - 605.255254 - 772.318143
    - ☑ Bexar County Boundary
      - ☐
    - Crime2009Table

2. Double-click on the **Burglaries in 2009** feature class in the **Crime** data frame to display the **Layer Properties** window, as shown in the following screenshot. Each of the tabs represents properties that can be set for the layer:

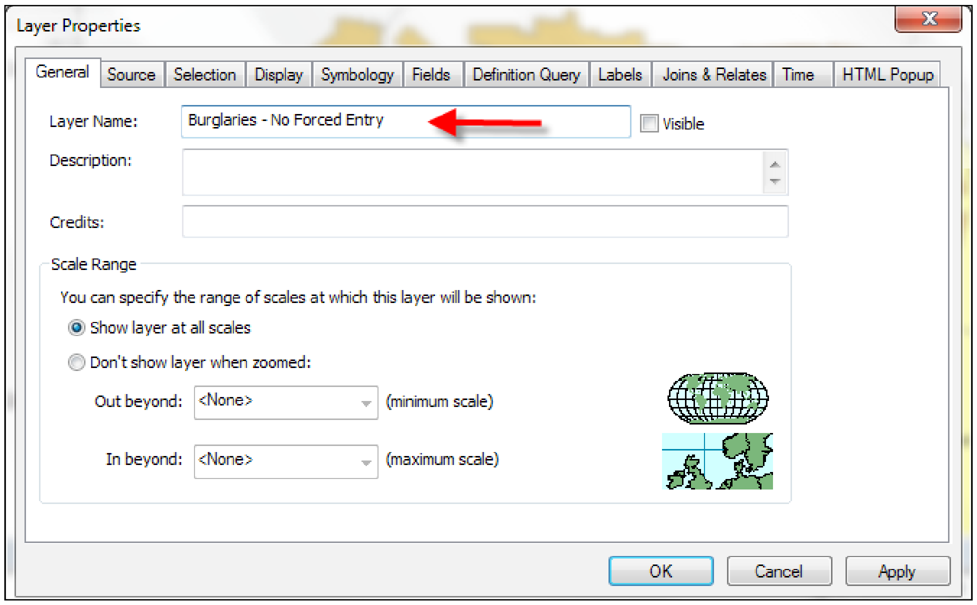


Draw all features using the same symbol.

Label appearing next to the symbol in table of contents:

Additional description appearing next to the symbol in your map's legend

3. Click on the **General** tab and change the value in the **Layer Name** textbox to the name shown in the following screenshot:



Layer Name: Burglaries - No Forced Entry

Description:

Credits:

Scale Range

You can specify the range of scales at which this layer will be shown:

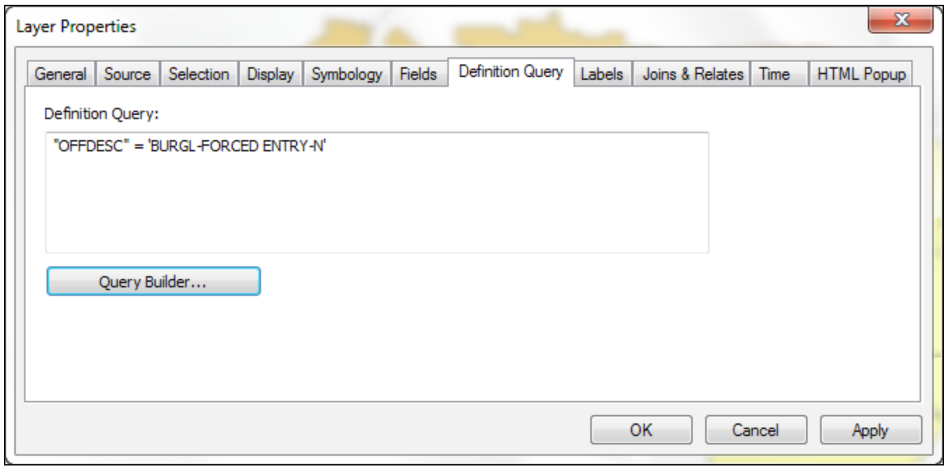
Show layer at all scales

Don't show layer when zoomed:

Out beyond: <None> (minimum scale)

In beyond: <None> (maximum scale)

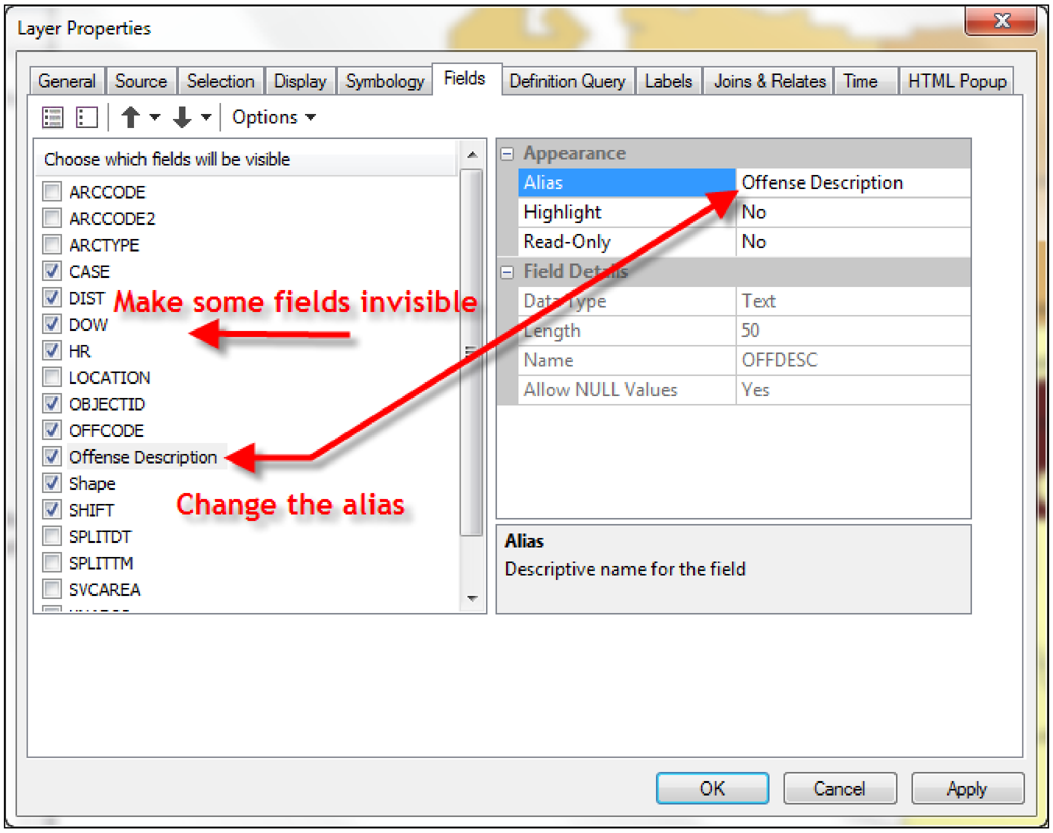
4. Click on the **Definition Query** tab and define the query shown in the following screenshot. You can use the **Query Builder** to define the query or simply type in the query:



Definition Query:

"OFFDESC" = 'BURGL-FORCED ENTRY-N'

5. Change the alias of the `OFFDESC` field to `Offense Description` as shown in the preceding screenshot.
6. Click on the **Fields** tab in **Layer Properties** and make visible only those fields that are selected with a checkmark in the following screenshot. This is done by unchecking the fields you see in the following screenshot.



Choose which fields will be visible

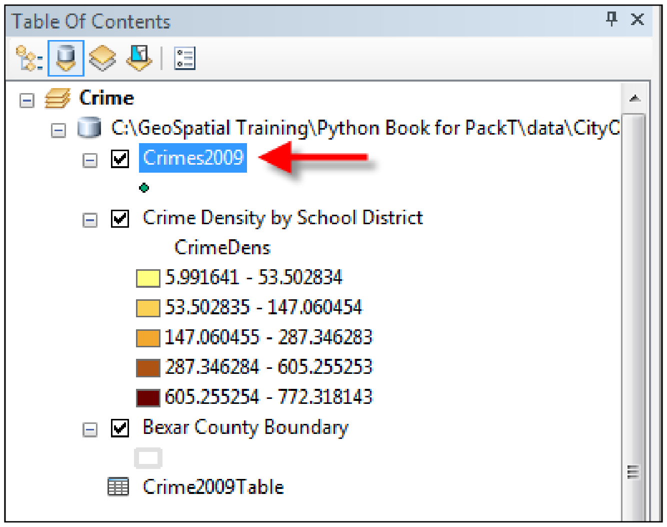
- [ ] ARCCODE
- [ ] ARCCODE2
- [ ] ARCTYPE
- [x] CASE
- [x] DIST
- [x] DOW
- [x] HR
- [ ] LOCATION
- [x] OBJECTID
- [x] OFFCODE
- [x] Offense Description
- [x] Shape
- [x] SHIFT
- [ ] SPLITDT
- [ ] SPLITTM
- [ ] SVCAREA

Appearance	
Alias	Offense Description
Highlight	No
Read-Only	No
Field Details	
Data Type	Text
Length	50
Name	OFFDESC
Allow NULL Values	Yes

Alias
Descriptive name for the field

7. Click on **OK** to dismiss the **Layer Properties** dialog.
8. In the data frame, right-click on **Burglaries – No Forced Entry** and select **Save as Layer File**.
9. Save the file as `c:\ArcpyBook\data\BurglariesNoForcedEntry.lyr`.
10. Right-click on the **Burglaries – No Forced Entry** layer and select **Remove**.

11. Using the **Add Data** button in ArcMap, add the **Crimes2009** feature class from the `CityOfSanAntonio` geodatabase. The `feature` class will be added to the data frame, as shown in the following screenshot:



12. Open the Python window in ArcMap.
13. Import the `arcpy.mapping` module:

```
import arcpy.mapping as mapping
```

14. Reference the currently active document (`Crime_Ch3.mxd`) and assign the reference to a variable:

```
mxd = mapping.MapDocument("CURRENT")
```

15. Get a reference to the `Crime` data frame:

```
df = mapping.ListDataFrames(mxd, "Crime")[0]
```

16. Define the layer that will be updated:

```
updateLayer = mapping.ListLayers(mxd,"Crimes2009",df)[0]
```

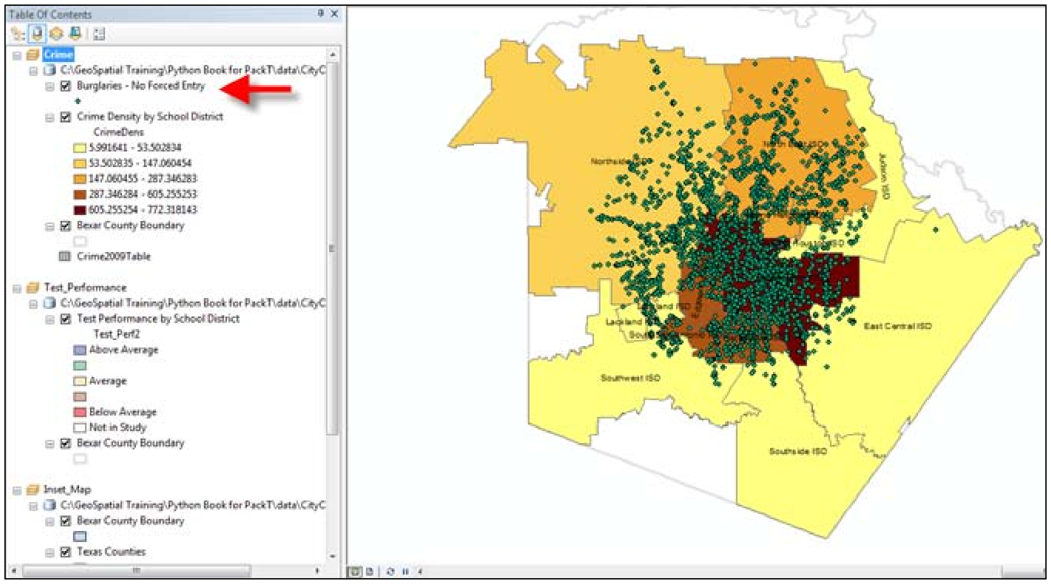
17. Define the layer that will be used to update the properties:

```
sourceLayer = mapping.Layer(r"C:\ArcpyBook\data\
BurglariesNoForcedEntry.lyr")
```

18. Call the `UpdateLayer()` function to update the symbology:

```
mapping.UpdateLayer(df,updateLayer,sourceLayer,False)
```

19. Run the script.
20. The **Crimes2009** layer will be updated with the properties associated with the `BurglariesNoForcedEntry.lyr` file. This is illustrated in the following screenshot. Turn on the layer to view the definition query that has been applied. You can also open the **Layer Properties** dialog to view the property changes that have been applied to the **Crimes2009** feature class:



4 Finding and Fixing Broken Data Links

In this chapter, we will cover the following recipes:

- Finding broken data sources in your map document and layer files
- Fixing broken data sources with MapDocument.findAndReplaceWorkspacePaths()
- Fixing broken data sources with MapDocument.replaceWorkspaces()
- Fixing individual Layer and Table objects with replaceDataSource()
- Finding all broken data sources in all map documents in a folder

Introduction

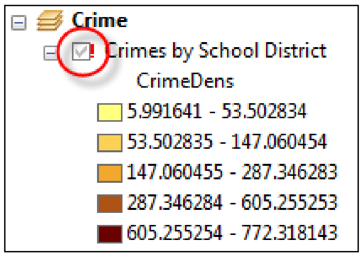
It is not uncommon for your GIS data sources to move, migrate to a new data format or be deleted. The result can be broken data sources in many map document or layer files. These broken data sources can't be used until they have been fixed, which can be an overwhelming process if the same changes need to be made across numerous map documents. You can automate the process of finding and fixing these data sources using `arcpy.mapping`, without ever having to open the affected map documents. Finding broken data sources is a simple process requiring the use of the `ListBrokenDataSources()` function, which returns a Python list of all broken data sources in a map document or layer file. Typically, this function is used as the first step in a script that iterates through the list and fixes the data sources. Fixing broken data sources can be performed on the individual data layer or across all the layers in a common workspace.

Finding broken data sources in your map document and layer files

Broken data sources are a very common problem with map document files. You can use `arcpy.mapping` to identify data sources that have moved, been deleted, or changed format.

Getting ready

In ArcMap, a broken data connection is signified by a red exclamation point just before the layer name. This is illustrated in the following screenshot. The `ListBrokenDataSources()` function in `arcpy.mapping` returns a list of layer objects from a map document or layer file that have a broken data connection:

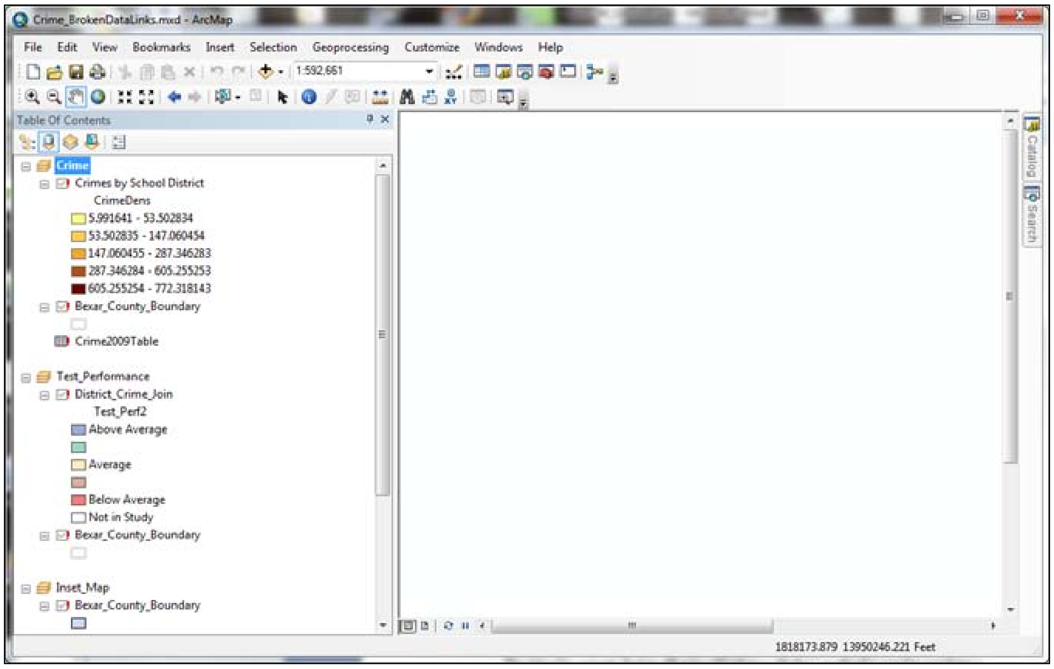


How to do it...

Follow the steps below to learn how to find broken data sources in a map document file.

1. Open `C:\ArcpyBook\Ch4\Crime_BrokenDataLinks.mxd` in ArcMap.

   You will see that each of the data sources have been broken. In this case, the data has been moved to another folder, but you'd see the same indicator if the data had been deleted or migrated to a different format. For example, it is not uncommon to convert data from a personal geodatabase to a file geodatabase:



2. Close ArcMap.
3. Open IDLE and create a new script window.
4. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

5. Reference the `Crime_BrokenDataLinks.mxd` map document file:

   ```
   mxd = mapping.MapDocument(r"c:\ArcpyBook\Ch4\Crime_
   BrokenDataLinks.mxd")
   ```

6. Get a list of the broken data sources:

   ```
   lstBrokenDS = mapping.ListBrokenDataSources(mxd)
   ```

7. Iterate the list and print out the layer names:

   ```
   for layer in lstBrokenDS:
       print layer.name
   ```

The ouptput will be printed as follows:

```
District_Crime_Join
Bexar_County_Boundary
District_Crime_Join
Bexar_County_Boundary
Bexar_County_Boundary
Texas_Counties_LowRes
School_Districts
Crime_surf
Bexar_County_Boundary
Crime2009Table
```

8. Save your script as `FindFixBrokenData.py` in the `c:\ArcpyBook\Ch4` folder.

How it works...

The `ListBrokenDataSources()` function returns a Python list of layers that have a broken data source. We then use a `for` loop to iterate this list and perform some sort of action for each layer. In this case, we printed out the layer names simply to illustrate the data returned by this function. In a later recipe, we'll build on this code by fixing these broken data sources.

There's more...

In addition to returning a list of broken data sources from a map document file, the `ListBrokenDataSources()` function can also find broken data sources in a layer file (`.lyr`). Simply pass in the path to the layer file to have the function examine the file for broken data sources. Keep in mind that these functions are not needed with `Map` or `Layer` packages, since the data is bundled with these files, unlike a layer file.

Fixing broken data sources with MapDocument. findAndReplaceWorkspacePaths()

The `MapDocument.findAndReplaceWorkspacePaths()` method is used to perform a global find and replace of workspace paths for all the layers and tables in a map document. You can also replace the paths to multiple workspace types at once.

Getting ready

We need to cover some definitions before examining the methods used to fix datasets. You'll see these terms used frequently when discussing the methods used to fix broken data sources, so you'll need to understand what they mean in this context. A **workspace** is simply a container for data. This can be a folder (in the case of shapefiles), personal geodatabase, file geodatabase, or ArcSDE connection. A **workspace path** is the system path to the **workspace**. In the case of file geodatabases, this would include the name of the **geodatabase**. A dataset is simply a feature class or table within a workspace, and finally, a data source is the combination of the workspace and dataset. Don't confuse a dataset with a feature dataset. The former is just a generic term for data, while the latter is an object within a geodatabase that serves as a container for feature classes and other datasets.

There are three `acpy.mapping` classes involved in fixing broken data sources. They are `MapDocument`, `Layer`, and `TableView`. Each class contains methods that can be used to fix data sources. In this recipe, we'll examine how you can use the `findAndReplaceWorkspacePaths()` method in the `MapDocument` class to perform a global find and replace operation on the layers and tables in a map document.

How to do it...

Follow these steps to learn how to fix layers and tables in a map document using `findAndReplaceWorkspacePaths()`:

1. Open `c:\ArcpyBook\Ch4\Crime_BrokenDataLinks.mxd` in ArcMap.
2. Right-click on any of the layers and select **Properties**.
3. Go to the **Source** tab and you will notice that the location for the layer refers to `C:\ArcpyBook\Ch4\Data\OldData\CityOfSanAntonio.gdb`. This is a file geodatabase but the location no longer exists; it has moved to the `C:\ArcpyBook\data` folder.
4. Open IDLE and create a new script window.
5. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

6. Reference the `Crime_BrokenDataLinks.mxd` map document file:

   ```
   mxd = mapping.MapDocument(r"c:\ArcpyBook\Ch4\Crime_
   BrokenDataLinks.mxd")
   ```

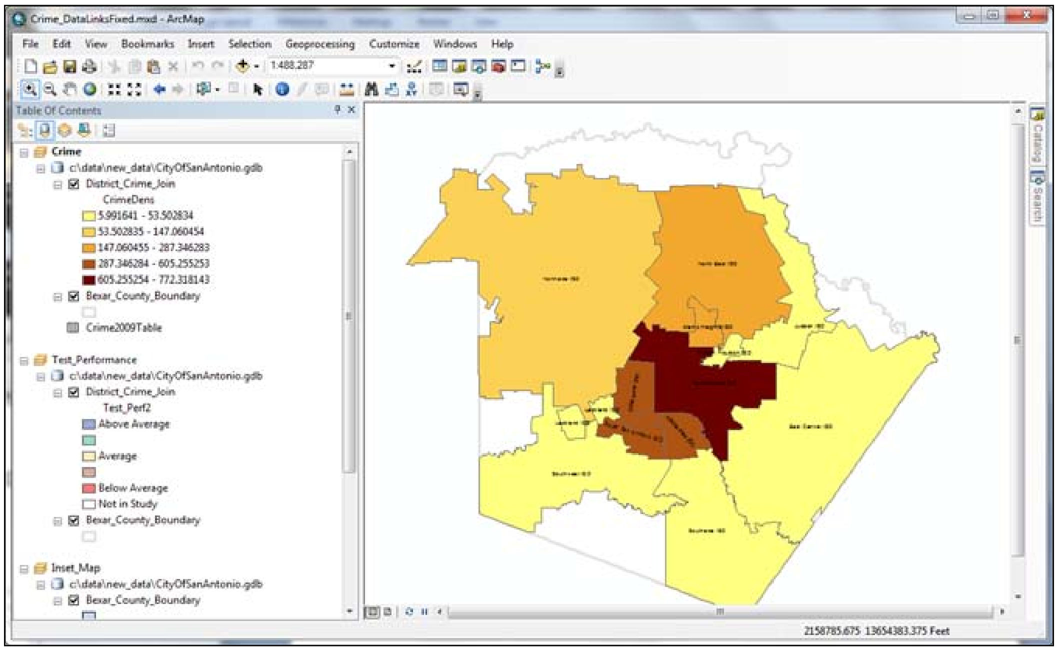
7. Use `MapDocument.findAndReplaceWorkspacePaths()` to fix the source path for each data source in the map document:

   ```
   mxd.findAndReplaceWorkspacePaths(r"C:\ArcpyBook\Ch4\Data\OldData\
   CityOfSanAntonio.gdb", r"C:\ArcpyBook\Data\CityOfSanAntonio.gdb")
   ```

8. Save the results to a new `.mxd` file:

   ```
   mxd.saveACopy(r"C:\ArcpyBook\Ch4\Crime_DataLinksFixed.mxd")
   ```

9. Save the script as `C:\ArcpyBook\Ch4\MapDocumentFindReplace.py`.
10. Run the script.
11. In ArcMap, open the `C:\ArcpyBook\Ch4\Crime_DataLinksFixed.mxd` file. You will notice that all the data sources get fixed, as shown in the following screenshot:



How it works...

The `MapDocument.findAndReplaceWorkspacePaths()` method is used to perform a global find and replace of workspace paths for all layers and tables in a map document. You can replace the paths to multiple workspace types at once.

There's more...

The `Layer` and `TableView` objects also have a `findAndReplaceWorkspacePaths()` method that performs the same type of operation. The difference is that this method, on the `Layer` and `TableView` objects, is used to fix an individual broken data source rather than a global find and replace of all broken data sources in a map document.

Fixing broken data sources with MapDocument.replaceWorkspaces()

During the course of normal GIS operations, it is a fairly common practice to migrate data from one file type to another. For example, many organizations migrate data from the older personal geodatabase formats to the new file geodatabase types or perhaps even enterprise ArcSDE geodatabases. You can automate the process of updating your datasets to a different format with `MapDocument.replaceWorkspaces()`.

Getting ready

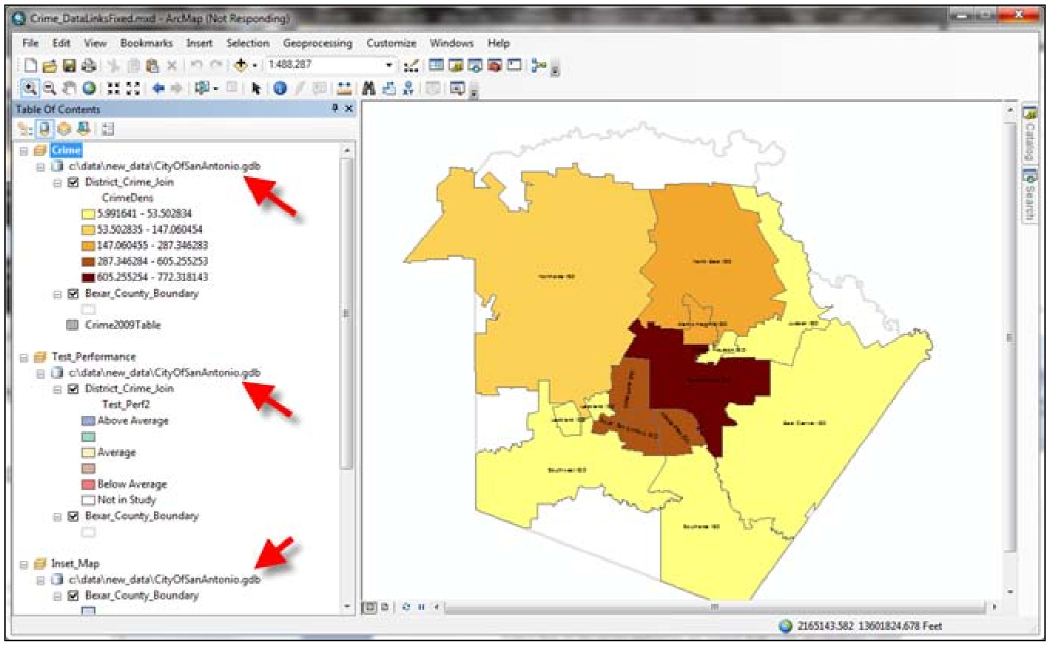
`MapDocument.replaceWorkspaces()` is similar to `MapDocument.findAndReplaceWorkspacePaths()`, but it also allows you to switch from one workspace type to another. For example, you can switch from a file geodatabase to a personal geodatabase. However, it only works on one workspace at a time. In this recipe, we'll use `MapDocument.replaceWorkspaces()` to switch our data source from a file geodatabase to a personal geodatabase.

How to do it...

Follow these steps to learn how to fix broken data sources using `MapDocument.replaceWorkspaces()`:

1. Open `c:\ArcpyBook\Ch4\Crime_DataLinksFixed.mxd` in ArcMap.

2. Notice that all the layers and tables are loaded from a file geodatabase called `CityOfSanAntonio.gdb`, as shown in the following screenshot:



3. Open IDLE and create a new script window.
4. Import the `arcpy.mapping` module:

```
import arcpy.mapping as mapping
```

5. Reference the `Crime_BrokenDataLinks.mxd` map document file:

```
mxd = mapping.MapDocument(r"c:\ArcpyBook\Ch4\Crime_
BrokenDataLinks.mxd.mxd")
```

6. Call `replaceWorkspaces()`, passing a reference to the old geodatabase type as well as the new geodatabase type:

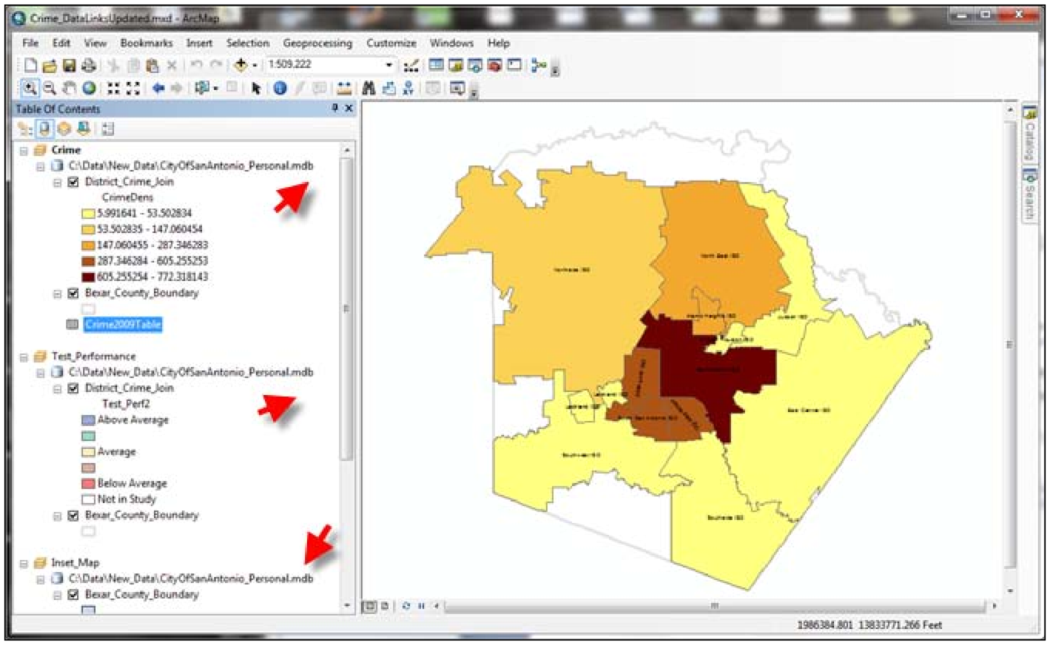
```
mxd.replaceWorkspaces(r"c:\ArcpyBook\data\CityOfSanAntonio.gdb",
"FILEGDB_WORKSPACE",r"c:\ArcpyBook\new_data\CityOfSanAntonio_
Personal.mdb","ACCESS_WORKSPACE")
```

7. Save a copy of the map document file.

```
mxd.saveACopy(r"c:\ArcpyBook\Ch4\Crime_DataLinksUpdated.mxd")
```

8. Save the script as `c:\ArcpyBook\Ch4\MapDocumentReplaceWorkspace.py`.
9. Run the script.

10. In ArcMap, open the `c:\ArcpyBook\Ch4\Crime_DataLinksUpdated.mxd` file. As shown in the following screenshot, all data sources now reference a personal geodatabase (note the `.mdb` extension):



How it works...

The `MapDocument.replaceWorkspaces()` method accepts several parameters including old and new workspace paths along with the old and new workspace types. Paths to the workspaces are self-explanatory, but some discussion of the workspace types is helpful. The workspace types are passed into the method as string keywords. In this case, the old workspace type was a file geodatabase so its keyword is `FILEGDB_WORKSPACE`. The new workspace type is `ACCESS_WORKSPACE`, which indicates a personal geodatabase. Personal geodatabases are stored in Microsoft Access files. There are a number of different workspace types that can store GIS data. Make sure you provide the workspace type that is appropriate for your dataset. The following is a list of valid workspace types (many people still work with shapefiles, so in this case the workspace type would be `SHAPEFILE_WORKSPACE`):

- `ACCESS_WORKSPACE`: A personal geodatabase or Access workspace
- `ARCINFO_WORKSPACE`: An ArcInfo coverage workspace

- `CAD_WORKSPACE`: A CAD file workspace
- `EXCEL_WORKSPACE`: An Excel file workspace
- `FILEGDB_WORKSPACE`: A file geodatabase workspace
- `NONE`: Used to skip the parameter
- `OLEDB_WORKSPACE`: An OLE database workspace
- `PCCOVERAGE_WORKSPACE`: A PC ARC/INFO Coverage workspace
- `RASTER_WORKSPACE`: A raster workspace
- `SDE_WORKSPACE`: An SDE geodatabase workspace
- `SHAPEFILE_WORKSPACE`: A shapefile workspace
- `TEXT_WORKSPACE`: A text file workspace
- `TIN_WORKSPACE`: A TIN workspace
- `VPF_WORKSPACE`: A VPF workspace

Fixing individual Layer and Table objects with replaceDataSource()

The previous recipes in this chapter have used various methods on the `MapDocument` object to fix broken data links. The `Layer` and `Table` objects also have methods that can be used to fix broken data links at the individual object level rather than working on all datasets in a map document file.

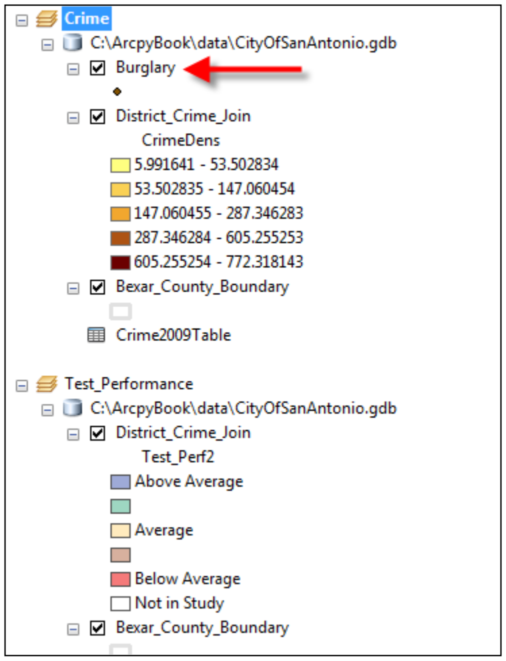
Getting ready

Both the `Layer` and `TableView` classes have a `replaceDataSource()` method. This method can be used to change the workspace path, workspace type, and/or dataset name for a single layer or table. In this recipe, you'll write a script that changes the workspace path and workspace type for a single layer. The `replaceDataSource()` method is available to the `Layer` and `TableView` classes. In the case of a layer it can be in either a map document or layer file. For a table, it can refer to the map document only, since `Table` objects can't be contained inside a layer file.

How to do it...

Follow these steps to learn how to fix individual `Layer` and `Table` objects in a map document using `replaceDataSource()`:

1. Open `c:\ArcpyBook\Ch4\Crime_DataLinksLayer.mxd` in ArcMap. The **Crime** data frame contains a layer called **Burglary**, which is a feature class in the `CityOfSanAntonio` file geodatabase. You're going to replace this feature class with a shapefile layer containing the same data:



```
Crime
  C:\ArcpyBook\data\CityOfSanAntonio.gdb
    ☑ Burglary
      ◆
    ☑ District_Crime_Join
        CrimeDens
      🟨 5.991641 - 53.502834
      🟨 53.502835 - 147.060454
      🟧 147.060455 - 287.346283
      🟫 287.346284 - 605.255253
      🟥 605.255254 - 772.318143
    ☑ Bexar_County_Boundary
      ☐
    Crime2009Table

Test_Performance
  C:\ArcpyBook\data\CityOfSanAntonio.gdb
    ☑ District_Crime_Join
        Test_Perf2
      🟦 Above Average
      🟩
      🟨 Average
      🟫
      🟥 Below Average
      ☐ Not in Study
    ☑ Bexar_County_Boundary
```

2. Open IDLE and create a new script window.
3. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

4. Reference the `Crime_BrokenDataLinks.mxd` map document file:

   ```
   mxd = mapping.MapDocument(r"c:\ArcpyBook\Ch4\ Crime_
   DataLinksUpdated.mxd")
   ```

5. Get a reference to the `Crime` data frame:

   ```
   df = mapping.ListDataFrames(mxd,"Crime")[0]
   ```

6. Find the `Burglary` layer and store it in a variable:

   ```
   lyr = mapping.ListLayers(mxd,"Burglary",df)[0]
   ```

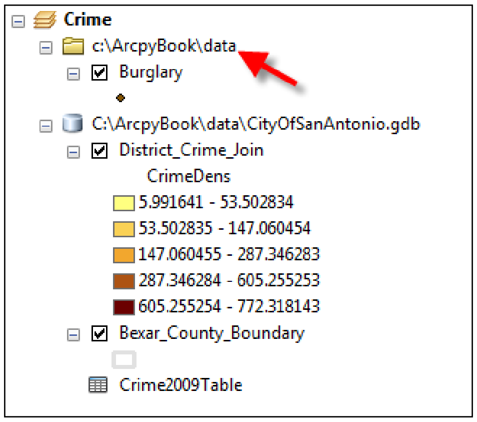
7. Call the `replaceDataSource()` method on the `Layer` object and pass in the path to the shapefile, a keyword indicating that this will be a shapefile workspace, and the name of the shapefile:

   ```
   lyr.replaceDataSource(r"c:\ArcpyBook\data","SHAPEFILE_
   WORKSPACE","Burglaries_2009")
   ```

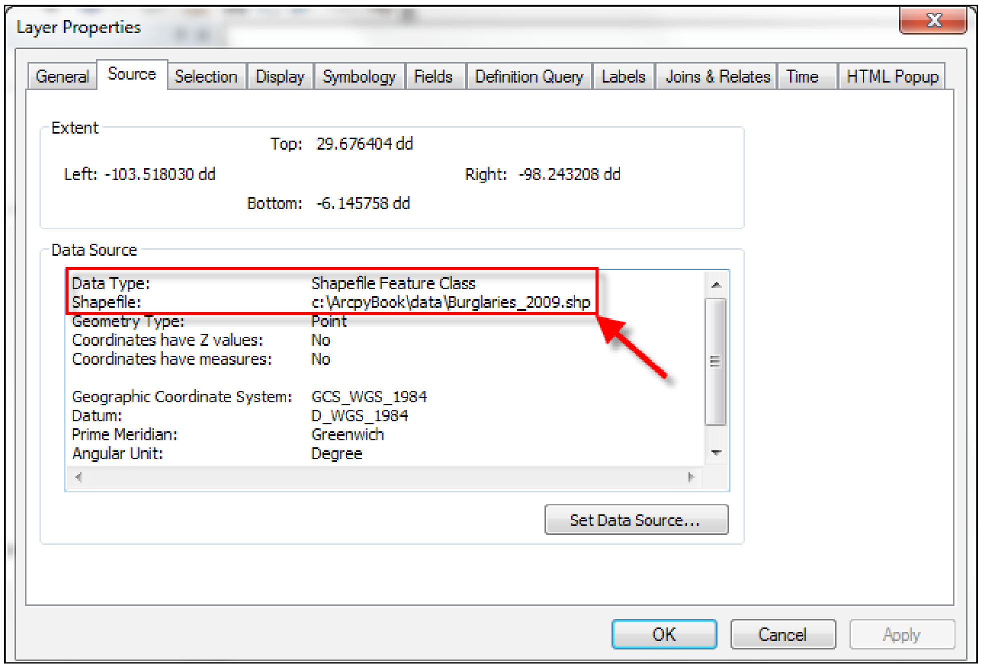
8. Save the results to a new map document file.

   ```
   mxd.saveACopy(r"c:\ArcpyBook\Ch4\Crime_DataLinksNewLayer.mxd")
   ```

9. Save the script as `c:\ArcpyBook\Ch4\LayerReplaceDataSource.py`.
10. Run the script.
11. Open `C:\ArcpyBook\Ch4\Crime_DataLinksNewLayer.mxd` in ArcMap. You should see that the **Burglary** layer now references a new workspace:



12. Right-click on the **Burglary** layer and select **Properties**.
13. Click on the **Source** tab and note the new workspace, workspace type, and dataset name:



Extent
Top: 29.676404 dd
Left: -103.518030 dd
Right: -98.243208 dd
Bottom: -6.145758 dd

Data Source

```
Data Type:                          Shapefile Feature Class
Shapefile:                          c:\ArcpyBook\data\Burglaries_2009.shp
Geometry Type:                      Point
Coordinates have Z values:          No
Coordinates have measures:          No

Geographic Coordinate System:       GCS_WGS_1984
Datum:                              D_WGS_1984
Prime Meridian:                     Greenwich
Angular Unit:                       Degree
```

How it works...

The `replaceDataSource()` method accepts two required parameters and two optional parameters. The first two parameters define the workspace path and workspace type for the layer that will be used as the replacement. The third parameter, `dataset_name`, is an optional parameter that defines the name of the dataset that will be used as the replacement layer. This name needs to be an exact match. For example, in this recipe, we passed in a `dataset_name` attribute `Burglaries_2009`, which is the name of the shapefile that will now be used as the replacement layer in the data frame. If a name is not provided, the method will attempt to replace the dataset by finding a table with the same name as the current layer's dataset property. The final optional parameter is `validate`. By default, this value is set to `true`. When set to `true`, a workspace will only be updated if the `workspace_path` value is a valid workspace. If it is not a valid workspace, then the workspace will not be replaced. If set to `false`, the method will set the source to match `workspace_path` regardless of whether it is a valid match or not. This can result in a broken data source, but can be useful if you are creating or modifying a map document in preparation for data that does not yet exist.

There's more...

The `Layer` and `TableView` classes also contain a `findAndReplaceWorkspacePath()` method. This method is very similar to the `MapDocument.findAndReplaceWorkspacePaths()` method. The only difference is that it works against a single `Layer` or `TableView` class instead of iterating the entire map document or layer file.

Finding all broken data sources in all map documents in a folder

A common scenario in many organizations involves the movement of data from one workspace to another or from one workspace type to another. When this happens, any map documents or layers that reference these data sources become broken. Finding each of these data sources can be a huge task if undertaken manually. Fortunately, you can create a geoprocessing script that will find all broken data sources in a folder or list of folders.

Getting ready

In this recipe, you will learn how to recursively search directories for map document files, find any broken data sources within those map documents, and write the names of the broken data layers to a file.

How to do it...

Follow these steps to learn how to find all broken data sources in all map documents in a folder:

1. Open IDLE and create a new script window.
2. Import the `arcpy` and `os` packages:

   ```
   import arcpy.mapping as mapping, os
   ```

3. Define a path where you'd like to begin the search. In this case, we're going to begin the search from the `C:` directory, which will then recursively search all directories inside the `C:` drive. You may want to define a more specific path:

   ```
   path = r"C:"
   ```

4. Open a file that you will use to which you will write the broken layer names:

   ```
   f = open('BrokenDataList.txt','w')
   ```

5. Use the `os.walk()` method along with a `for` loop to walk the directory tree:

   ```
   for root,dirs,files in os.walk(path):
   ```

6. Inside the `for` loop, create a second `for` loop that loops through all the files returned. For each file, use the `os.path.splitext()` method to obtain the base file name as well as the extension:

   ```
   for filename in files:
       basename, extension = os.path.splitext(filename)
   ```

7. Test the file extension to see if it is a map document file. If so, get the full path to the map document file, create a new map document object instance using the path, write the map document name, loop through each of the broken data sources, and write to a file:

   ```
   if extension == ".mxd":
       fullPath = os.path.join(path,filename)
       mxd = mapping.MapDocument(fullPath)
       f.write("MXD: " + filename + "\n")
       brknList = mapping.ListBrokenDataSources(mxd)
       for brknItem in brknList:
           f.write("\t" + brknItem.name + "\n")
   ```

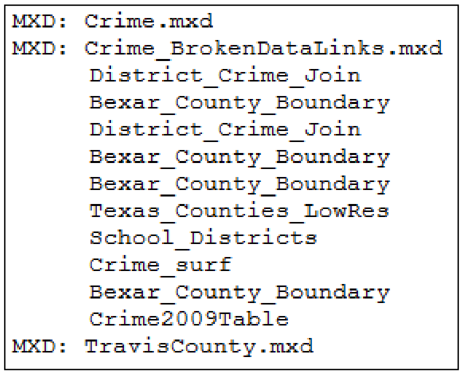
8. Close the file:

   ```
   f.close()
   ```

9. The entire script should appear as follows:

   ```
   import arcpy.mapping as mapping, os
   path = r"C:"
   f = open('filename_here.txt','w')
   for root,dirs,files in os.walk(path):
       for filename in files:
           basename, extension = os.path.splitext(filename)
           if extension == ".mxd":
               fullPath = os.path.join(path,filename)
               mxd = mapping.MapDocument(fullPath)
               f.write("MXD: " + filename + "\n")
               brknList = mapping.ListBrokenDataSources(mxd)
               for brknItem in brknList:
                   f.write("\t" + brknItem.name + "\n")
   f.close()
   ```

10. Run the script to generate the file.
11. Open the file to see the results. Your output will vary depending upon the path you've defined. The following screenshot shows my output file:



```
MXD: Crime.mxd
MXD: Crime_BrokenDataLinks.mxd
      District_Crime_Join
      Bexar_County_Boundary
      District_Crime_Join
      Bexar_County_Boundary
      Bexar_County_Boundary
      Texas_Counties_LowRes
      School_Districts
      Crime_surf
      Bexar_County_Boundary
      Crime2009Table
MXD: TravisCounty.mxd
```

How it works...

This script uses a combination of methods from the Python `os` package and `arcpy.mapping` package. The `os.walk()` method walks a directory tree and returns the path, a list of directories, and a list of files for each directory starting with a root directory that you have defined as the `c:` directory. This root directory could have been any directory. The `os.walk()` method returns a three item tuple consisting of the root directory, a list of directories, and a list of files. We then loop through this list of files and use the `os.path.splitext()` method to split each file into a base file name and file extension. The extension is tested to see if it matches the string `.mxd`, which indicates a map document file. Files identified as map documents have their filenames written to a text file, and a new `MapDocument` object instance is created. The `ListBrokenDataSources()` method is then used with a reference to the map document to generate a list of broken data sources within the file, and these broken data sources are written to the file as well.

5
Automating Map Production and Printing

In this chapter, we will cover the following recipes:

- Creating a list of layout elements
- Assigning a unique name to layout elements
- Restricting the layout elements returned by ListLayoutElements()
- Updating layout element properties
- Getting a list of available printers
- Printing maps with PrintMap()
- Exporting a map to a PDF file
- Exporting a map to an image file
- Creating a map book with PDFDocumentCreate() and PDFDocumentOpen()

Introduction

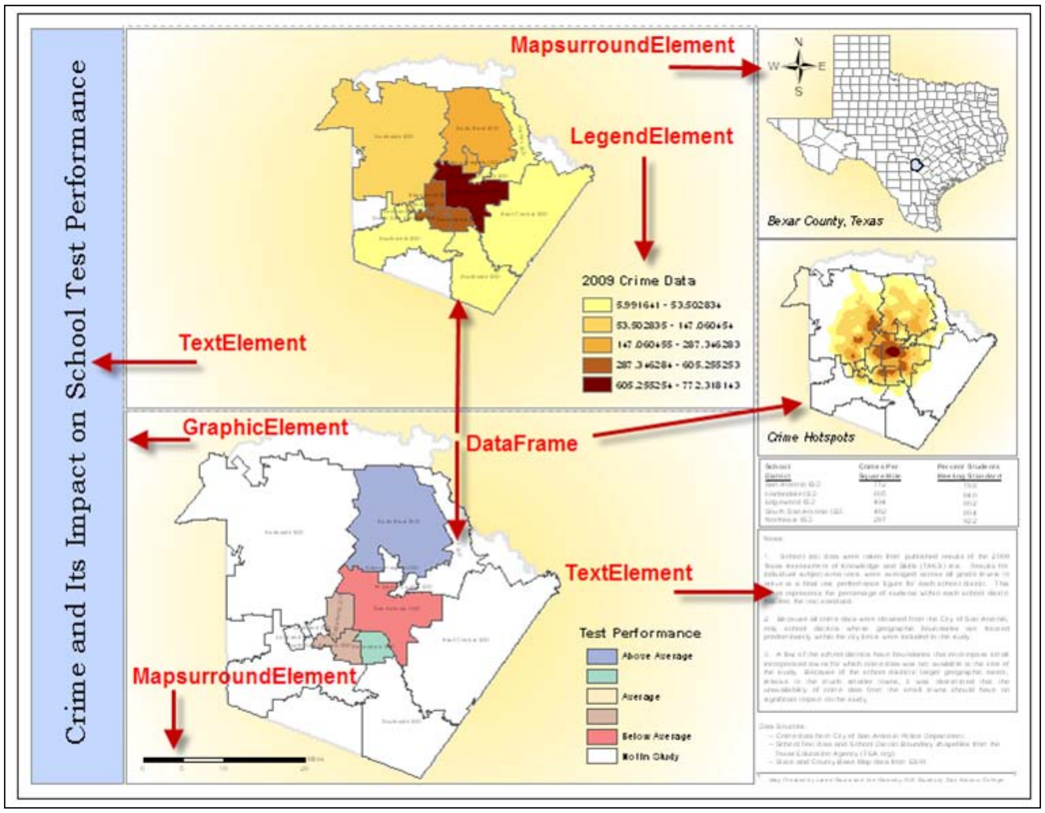
The `arcpy.mapping` module, released with ArcGIS 10, provides a number of capabilities related to the automation of map production. `arcpy.mapping` can be used to automate map production, build map books, export to image or PDF files, and create and manage PDF files. In this chapter, you'll learn how to use the `arcpy.mapping` module to automate various geoprocessing tasks related to map production and printing.

Creating a list of layout elements

Often, the first step in a geoprocessing script that automates the production of maps is to generate a list of the available layout elements. For example, you might need to update the title of your map before printing or creating a PDF file. In this case, the title would likely be stored in a `text` element. You can generate a list of `text` elements in your map layout view and then change the title in it. The first step is to generate a list of `text` elements.

Getting ready...

In ArcMap, two views are available, namely, data view and layout view. **Data view** is used to view geographic and tabular data, analyze data, symbolize layers, and manage data without regard for any particular map page size or layout. **Layout view** shows the map as printed on a page, and is used to create production-quality maps through the addition of map elements. These elements include map frames, layers, legends, titles, north arrows, scale bars, and title blocks. Each object in the layout is represented in `arcpy.mapping` as a layout element class. Examples of many of these layout element classes are displayed in the following screenshot:



Each element can be assigned a unique name that can then be used to access the element programmatically. This unique name is defined in ArcMap. The `arcpy.mapping` module provides a `ListLayoutElements()` function that returns a list of all these elements. In this recipe, you will learn to use the `ListLayoutElements()` function to generate a list of map layout elements.

How to do it...

Follow these steps to learn how to generate a list of layout elements:

1. Open `C:\ArcpyBook\Ch5\Crime_Ch5.mxd` in ArcMap.
2. Open the Python window.
3. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

4. Reference the currently active document (`Crime_Ch5.mxd`) and assign the reference to a variable:

   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Generate a list of layout elements and print them to the screen if the name property is not empty:

   ```
   for el in mapping.ListLayoutElements(mxd):
     if el.name != '':
       print el.name
   ```

6. The entire script should appear as follows:

   ```
   import arcpy.mapping as mapping
   mxd = mapping.MapDocument("CURRENT")
   for el in mapping.ListLayoutElements(mxd):
     if el.name != '':
       print el.name
   ```

7. Run the script to see the following output:

   ```
   Crime_Inset
   Alternating Scale Bar
   Legend Test Performance
   Crime Legend
   North Arrow
   Inset_Map
   Test_Performance
   Crime
   ```

How it works...

`ListLayoutElements()` returns a list of layout elements in the form of various layout classes. Each element can be one of the following object instances: `GraphicElement`, `LegendElement`, `PictureElement`, `TextElement`, or `MapSurroundElement`. Each element can have a unique name. You don't have to assign a name to each element, but it is helpful to do so if you plan to access these elements programmatically in your scripts. In this script, we first made sure that the element had a name assigned to it before printing the name. This was done because ArcMap does not require that an element be assigned a name.

Assigning a unique name to layout elements

It's a good practice to assign a unique name to all your layout elements using ArcMap. This is important in the event that your geoprocessing scripts need to access a particular element to make changes. For example, you might need to update the icon that displays your corporate logo. Rather than making this change manually in all your map document files, you could write a geoprocessing script that updates all your map document files programmatically with the new logo. However, in order for this to be possible, a unique name will need to be assigned to your layout elements. This gives you the ability to access the elements of your layout individually.

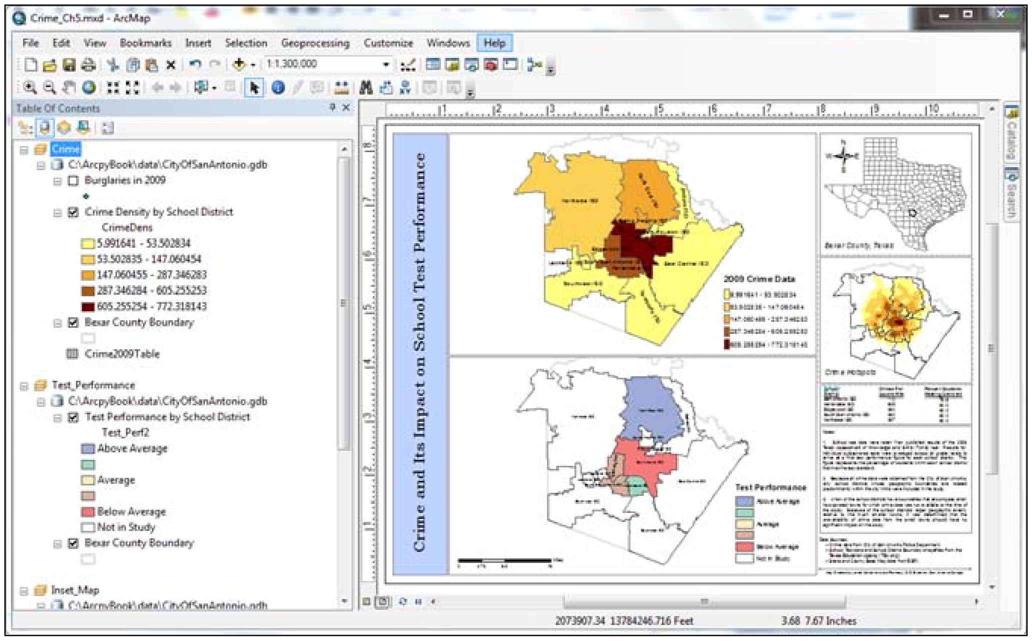
Getting ready

As I mentioned in the previous recipe, each layout element will be one of a number of element types and each can be assigned a name. This element name can then be used when you need to reference a particular element in your Python script. You can use ArcMap to assign unique names to each layout element. In this recipe, you will use ArcMap to assign names to the elements.

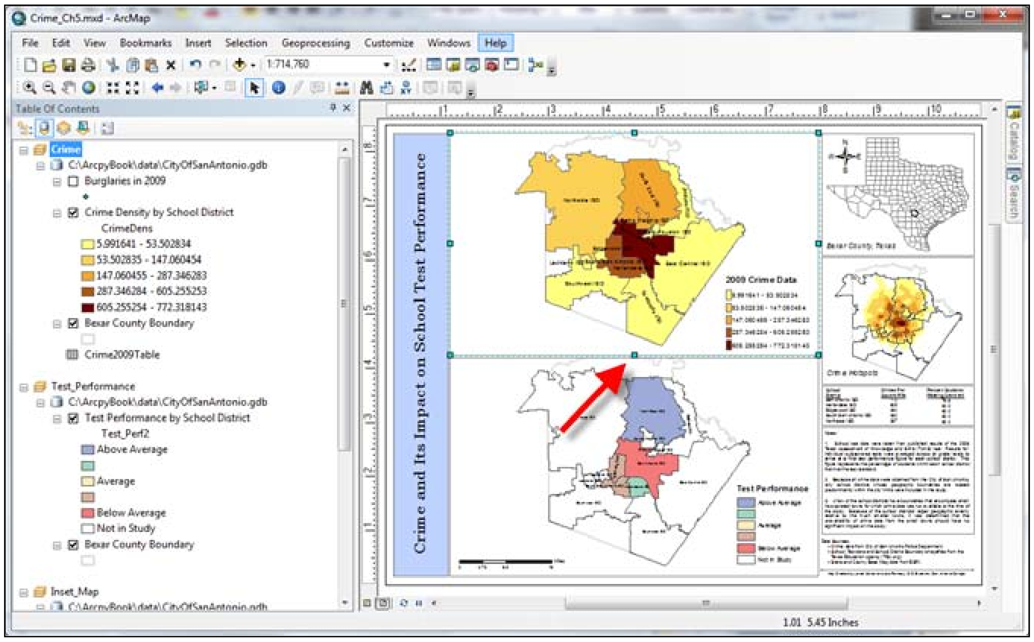
How to do it...

Follow these steps to learn how to assign unique names to each layout element using ArcMap:

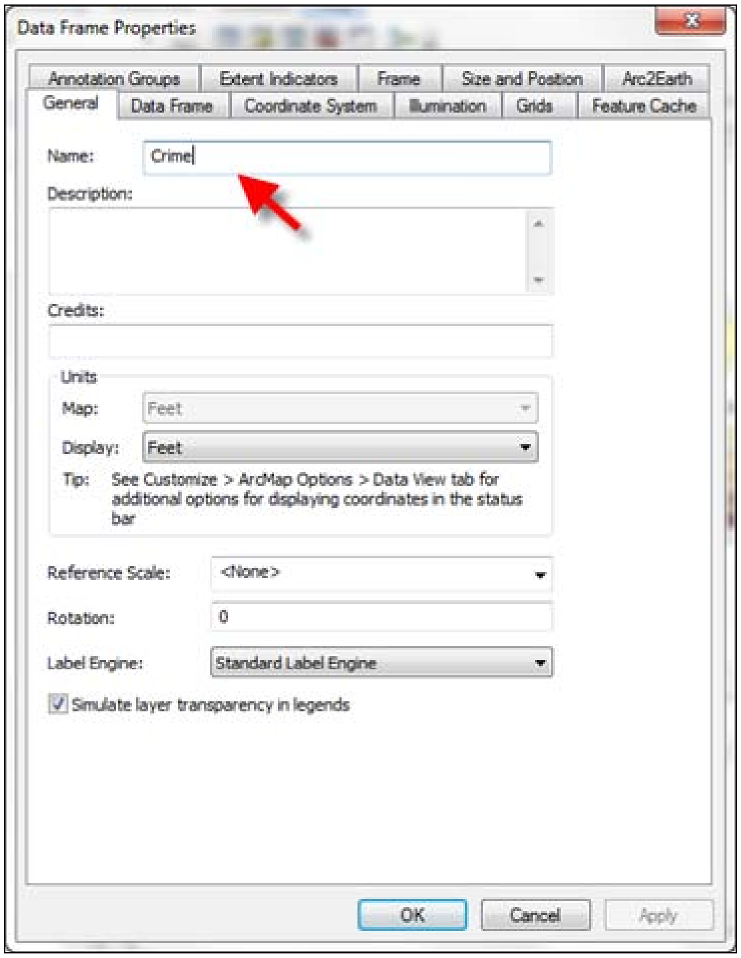
1. Open `C:\ArcpyBook\Ch5\Crime_Ch5.mxd` in ArcMap.
2. Switch to layout view and you should something similar to the following screenshot:



3. Names are assigned differently depending upon the element type. Click on the uppermost data frame, which should be **Crime**, to select it. The selection handles should appear as follows:


Crime_Ch5.mxd - ArcMap
File Edit View Bookmarks Insert Selection Geoprocessing Customize Windows Help
1:714,760
Table Of Contents
Crime
Crime and Its Impact on School Test Performance
2009 Crime Data
Test Performance
1.01 5.45 Inches

4. Right-click on the upper data frame and select **Properties** to display the **Data Frame Properties** window, as shown in the following screenshot. The **Name** property defines the unique name for the element. In this case, the element name is `Crime`:



Name: Crime

Description:

Credits:

Units

Map: Feet

Display: Feet

Tip: See Customize > ArcMap Options > Data View tab for additional options for displaying coordinates in the status bar

Reference Scale: <None>

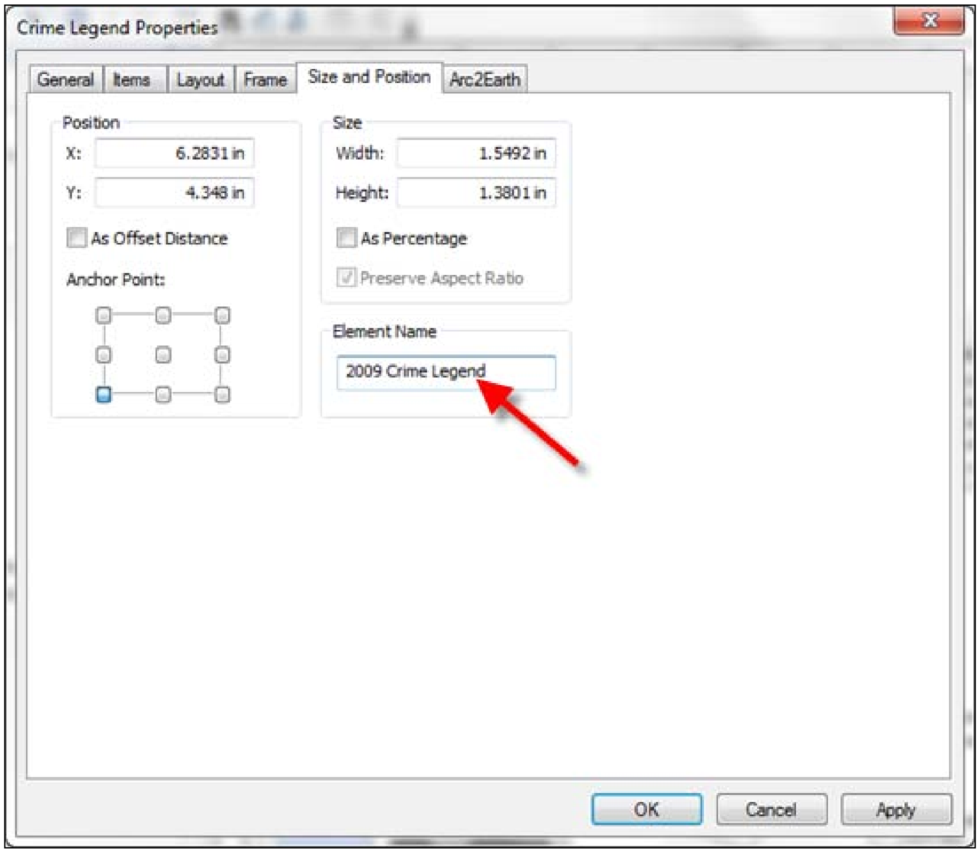
Rotation: 0

Label Engine: Standard Label Engine

[✓] Simulate layer transparency in legends

5. Close the **Data Frame Properties** window.
6. Select **2009 Crime Legend** in the layout view and open the **Properties** window by right-clicking on the legend and selecting **Properties**.

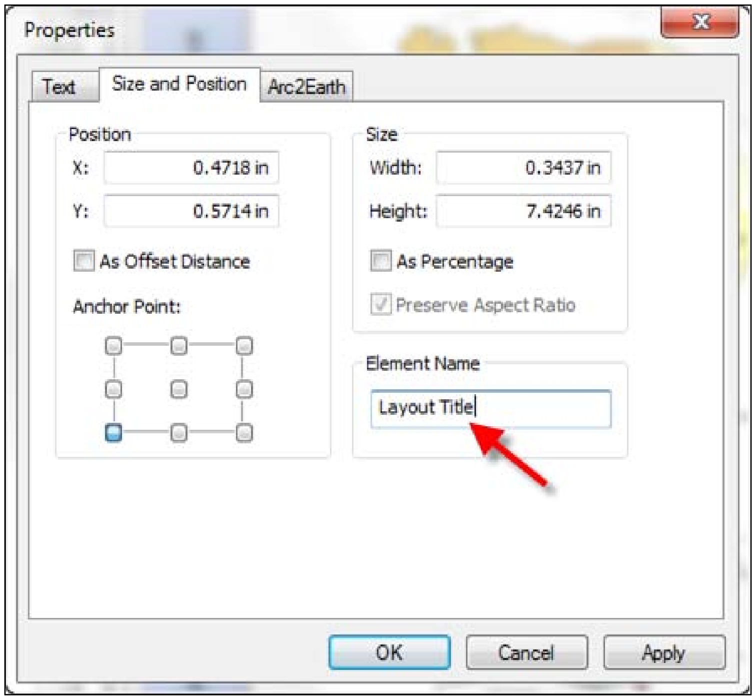
7. Setting the element name for the legend is different from setting the element name for a data frame. Click on the **Size and Position** tab.
8. The **Element Name** textbox is used in this case. Change the current value to `2009 Crime Legend`, as shown in the following:



X: 6.2831 in
Y: 4.348 in
As Offset Distance
Width: 1.5492 in
Height: 1.3801 in
As Percentage
Preserve Aspect Ratio
Element Name: 2009 Crime Legend

9. You can also define unique names for text elements. Select the title element (`Crime and Its Impact on School Test Performance`), right-click on the element, and select **Properties**.

10. Select the **Size and Position** tab and define a unique name for this element, as shown in the following screenshot:



How it works...

Each element in the layout view can be assigned a name, which can then be used in your geoprocessing script to retrieve the specific element. You should strive to define unique names for each element. It isn't required that you define a unique name for each element, nor is it required that you even define a name at all. However, it is a best practice to give each element a name and ensure that each element name is unique if you intend to access these elements from your Python scripts. In terms of naming practices for your elements, you should strive to include only letters and underscores in the name.

There's more...

You can use element names in conjunction with the `ListLayoutElements()` function to restrict the elements that are returned by the function through the use of a wildcard parameter. In the next recipe, you'll learn how to restrict the list of layout elements returned through the use of wildcards and element type.

Restricting the layout elements returned by ListLayoutElements()

Layouts can contain a large number of elements, many of which you won't need for a particular geoprocessing script. The `ListLayoutElements()` function can restrict the layout elements returned by passing in a parameter that defines the type of element that should be returned along with an optional wildcard, which finds elements using a portion of the name.

Getting ready

There are many different types of layout elements including graphics, legends, pictures, text, and data frames. When you return a list of layout elements, you can restrict (filter) the types of elements that are returned. In this recipe, you will write a script that filters the layout elements returned by element type and wildcard.

How to do it...

Follow these steps to learn how to restrict the list of layers returned by the `ListLayoutElements()` function through the use of optional parameters, which define the type of element that should be returned along with a wildcard that can also restrict the elements returned.

1. Open `C:\ArcpyBook\Ch5\Crime_Ch5.mxd` in ArcMap.
2. Open the Python window.
3. Import the `arcpy.mapping` module:
   ```
   import arcpy.mapping as mapping
   ```
4. Reference the currently active document (`Crime_Ch5.mxd`) and assign the reference to a variable:
   ```
   mxd = mapping.MapDocument("CURRENT")
   ```
5. Use the `ListLayoutElements()` function with a restriction of only legend elements as well as a wildcard that returns elements with a name containing the text `Crime` anywhere in the name:
   ```
   for el in mapping.ListLayoutElements(mxd,"LEGEND_
   ELEMENT","*Crime*")
     print el.name
   ```
6. Run the script. In this case, only a single layout element will be returned:
   ```
   2009 Crime Legend
   ```

How it works...

ListLayoutElements() is a versatile function, which in its most basic form is used to return a list of all the layout elements on the page layout of a map document. However, there are two optional parameters that you can supply to filter this list. The first type of filter is an element type filter in which you specify that you only want to return one of the layout element types. You can also apply a wildcard to filter the returned list. These two types of filters can be used in combination. For example, in this recipe we are specifying that we only want to return LEGEND_ELEMENT objects with the text "Crime" anywhere in the element name. This results in a highly filtered list that only contains a single layout element.

`ListLayoutElements()` can be filtered by one of the following element types: `DATAFRAME_ELEMENT`, `GRAPHIC_ELEMENT`, `LEGEND_ELEMENT`, `MAPSURROUND_ELEMENT`, `PICTURE_ELEMENT`, `TEXT_ELEMENT`.

Updating layout element properties

Each layout element has a set of properties that you can update programmatically. For example, `LegendElement` includes properties that allow you to change the position of the legend on the page, update the legend title, and access the legend items.

Getting ready

There are many different types of layout elements including graphics, legends, text, maps, and pictures. Each of these elements is represented by a class in the `arcpy.mapping` package. These classes provide various properties that you can use to programmatically alter the element.

The `DataFrame` class provides access to data frame properties in the map document file. This object can work with both map units and page layout units, depending upon the property being used. Page layout properties, such as positioning and sizing, can be applied to the properties including `elementPositionX`, `elementPositionY`, `elementWidth`, and `elementHeight`.

The `GraphicElement` object is a generic object for various graphics that can be added to the page layout including tables, graphs, neatlines, markers, lines, and area shapes. You'll want to make sure that you set the `name` property for each graphic element (and any other element for that matter), if you intend to access it through a Python script.

`LegendElement` provides operations for positioning of the legend on the page layout, modification of the legend title, and also provides access to the legend items and the parent data frame. `LegendElement` can be associated with only a single data frame.

MapSurroundElement can refer to north arrows, scale bars, and scale text, and like LegendElement is associated with a single data frame. Properties on this object enable repositioning on the page.

PictureElement represents a raster or image on the page layout. The most useful property on this object allows for getting and setting the data sources, which can be extremely helpful when you need to change a picture, such as a logo, in multiple map documents. For example, you could write a script that iterates through all your map document files and replaces the current logo with a new logo. You can also reposition or resize the object.

TextElement represents text on a page layout including inserted text, callouts, rectangle text, and titles, but does not include legend titles or text that are part of a table or chart. Properties enable modifying the text string, which can be extremely useful in situations where you need to make the same text string change in multiple places in the page layout or over multiple map documents, and of course repositioning of the object is also available.

Each element in the page layout is returned as an instance of one of the element objects. In this recipe, we're going to use the title property on the Legend object to programmatically change the title of the Crime legend and obtain a list of the layers that are part of the legend.

How to do it...

Follow these steps to learn how to update the properties of a layout element:

1. Open C:\ArcpyBook\Ch5\Crime_Ch5.mxd in ArcMap.
2. Open the Python window.
3. Import the arcpy.mapping module:

   ```
   import arcpy.mapping as mapping
   ```

4. Reference the currently active document (Crime_Ch5.mxd), and assign the reference to a variable:

   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Use the ListLayoutElements() method with a wildcard and restriction of only legend elements to return only the Crime legend and store it in a variable:

   ```
   elLeg = mapping.ListLayoutElements(mxd, "LEGEND_
   ELEMENT","*Crime*")[0]
   ```

6. Use the title property to update the title of the legend:

   ```
   elLeg.title = "Crimes by School District"
   ```

7. Get a list of the layers that are a part of the legend and print the names:

   ```
   for item in elLeg.listLegendItemLayers():
     print item.name
   ```

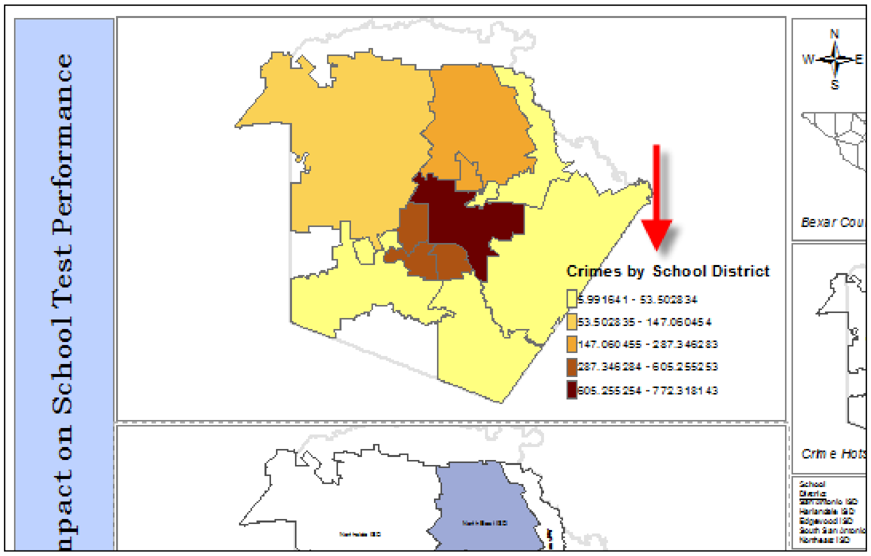
8. The entire script should appear as follows:

```
import arcpy.mapping as mapping
mxd = mapping.MapDocument("CURRENT")
elLeg = mapping.ListLayoutElements(mxd, "LEGEND_
ELEMENT","*Crime*")[0]
elLeg.title = "Crimes by School District"
for item in elLeg.listLegendItemLayers():
  print item.name
```

9. Run the script. You should see the following layers printed:

```
Burglaries in 2009
Crime Density by School District
```

10. The change is displayed in the following screenshot:



How it works...

Each of the layout elements has a set of properties and methods. In this particular case, we've used the `title` property on the `Legend` object. Other properties of this object allow you to set the width and height, positioning, and others. Methods on the `Legend` object give you the ability to adjust the column count, list the legend items, and remove and update items.

Getting a list of available printers

Yet another list function provided by `arcpy` is `ListPrinterNames()`, which generates a list of the available printers. As is the case with the other list functions that we've examined, `ListPrinterNames()` is often called as a preliminary step in a multi-step script.

Getting ready

Before printing maps with the `PrintMap()` function, it is a common practice to call the `ListPrinterNames()` function, which returns a list of the available printers for the local computer. A particular printer can then be found by iterating through the list of printers and using it as an input to the `PrintMap()` function.

How to do it...

Follow these steps to learn how to use the `ListPrinterNames()` function to return a list of the available printers for your script:

1. Open `C:\ArcpyBook\Ch5\Crime_Ch5.mxd` in ArcMap.
2. Open the Python window.
3. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

4. Reference the currently active document (`Crime_Ch5.mxd`) and assign the reference to a variable:

   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Call the `ListPrinterNames()` function and print each printer:

   ```
   for printerName in mapping.ListPrinterNames():
     print printerName
   ```

6. Run the script. The output will vary depending upon the list of the available printers for your computer. However, it should print something similar to the following code snippet:

   ```
   HP Photosmart D110 series
   HP Deskjet 3050 J610 series (Network)
   HP Deskjet 3050 J610 series (Copy 1)
   HP Deskjet 3050 J610 series
   Dell 968 AIO Printer
   ```

How it works...

The `ListPrinterNames()` function returns a Python list containing all the printers available to use in your script. You can then use the `PrintMap()` function, which we'll examine in the next recipe, to send a print job to a particular printer that is available to your computer.

Printing maps with PrintMap()

Sending your map layout to a printer is easy with the `PrintMap()` function. By default, the print job will be sent to the default printer saved with the map document, but you can also define a specific printer where the job should be sent.

Getting ready

`arcpy.mapping` provides a `PrintMap()` function for printing page layouts or data frames from ArcMap. Before calling `PrintMap()`, it is a common practice to call the `ListPrinterNames()` function. A particular printer can then be found by iterating the list of printers and used as an input to the `PrintMap()` function.

`PrintMap()` can print either a specific data frame or the page layout of a map document. By default, this function will use the printer saved with the map document or, if not present in the map document, the default system printer. As I mentioned, you can also use `ListPrinterNames()` to get a list of the available printers, and select one of these printers as an input to `PrintMap()`. In this recipe, you will learn how to use the `PrintMap()` function to print the layout.

How to do it...

Follow these steps to learn how to use the `PrintMap()` function to print the layout view in ArcMap:

1. Open `C:\ArcpyBook\Ch5\Crime_Ch5.mxd` in ArcMap.
2. Open the Python window.
3. Import the `arcpy.mapping` module:
   ```
   import arcpy.mapping as mapping
   ```
4. Reference the currently active document (`Crime_Ch5.mxd`), and assign the reference to a variable:
   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Look for the `Test_Performance` data frame and print if it is found:

   ```
   for df in mapping.ListDataFrames(mxd):
     if df.name == "Test_Performance":
       mapping.PrintMap(mxd,"",df)
   ```

How it works...

The `PrintMap()` function accepts one required parameter and a handful of optional parameters. The required parameter is a reference to the map document. The first optional parameter is the printer name. In this case, we haven't specified a particular printer to use. Because we haven't provided a specific printer, it will use the printer saved with the map document or the default system printer if a printer is not part of the map document. The second optional parameter is the data frame that we'd like to print, which in this instance is `Test_Performance`. Other optional parameters, not supplied in this case, are an output print file and the image quality.

Exporting a map to a PDF file

Rather than sending your map or layout view to a printer, you may want to simply create PDF files that can be shared. Arcpy mapping provides an `ExportToPDF()` function, which you can use to do this.

Getting ready

PDF is a very popular interchange format designed to be viewable and printable from many different platforms. The arcpy mapping `ExportToPDF()` function can be used to export data frames or the page layout to PDF format. By default, the `ExportToPDF()` function exports the page layout, but you can pass in an optional parameter that references a particular data frame, which can be printed instead of the page layout. In this recipe, you will learn how to export the page layout as well as a specific data frame to a PDF file.

How to do it...

Follow these steps to learn how to export a map to a PDF fil:

1. Open `C:\ArcpyBook\Ch5\Crime_Ch5.mxd` in ArcMap.
2. Open the Python window.
3. Import the `arcpy.mapping` module:

   ```
   import arcpy.mapping as mapping
   ```

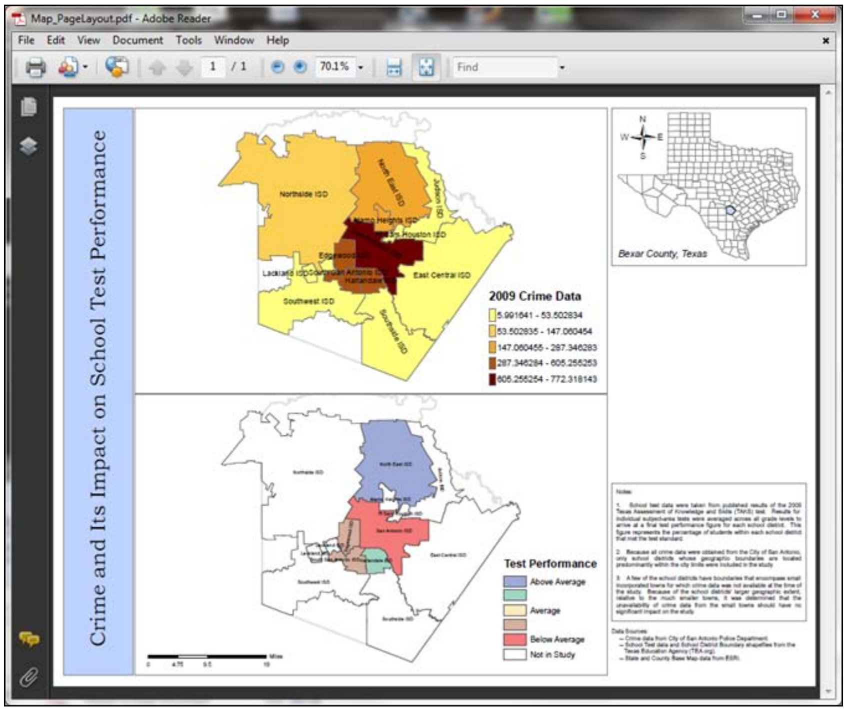
4. Reference the currently active document (`Crime_Ch5.mxd`), and assign the reference to a variable:

```
mxd = mapping.MapDocument('CURRENT')
```

5. Export the page layout with the `ExportToPDF()` function:

```
mapping.ExportToPDF(mxd,r"c:\ArcpyBook\Ch5\Map_PageLayout.pdf").
```

6. Run the script.
7. Open the `Map_PageLayout.pdf` file that was created, and you should see something similar to the following screenshot:



8. Now, we'll print a specific data frame from our map document file. Alter your script, so that it appears as follows:

```
import arcpy.mapping as mapping
mxd = mapping.MapDocument('CURRENT')
for df in mapping.ListDataFrames(mxd):
  if df.name == "Crime":
    mapping.ExportToPDF(mxd,r"c:\ArcpyBook\Ch5\DataFrameCrime.
pdf",df)
```

9. Run the script and examine the output PDF.

How it works...

The `ExportToPDF()` function requires two parameters, including a reference to the map document and the file that serves as the output PDF file. The first script that we developed, passed in a reference to the map document along with an output PDF file. Since we didn't pass in an optional parameter specifying the data frame, the `ExportToPDF()` function will export the page layout. There are also many optional parameters that can be passed into this method, including a specific data frame and a number of parameters, mostly related to the quality of the output content and file. Our second script passed in a specific data frame that should be exported. You can reference the ArcGIS help pages for more information on each of the optional parameters.

Exporting a map to an image file

You can also export the contents of the map or layout view to an image file using one of the many functions provided by `arcpy.mapping`. Each image export function will differ in name depending upon the type of image file you'd like to create. The parameters passed into the function will also vary slightly.

Getting ready

In addition to providing the ability to export data frames and the page layout to a PDF format, you can also use one of the many export functions provided by `arcpy.mapping` to export an image file. Available formats include AI, BMP, EMF, EPS, GIF, JPEG, SVG, and TIFF. The parameters provided for each function will vary depending upon the type of image. Some examples of these function names include `ExportToJPEG()`, `ExportToGIF()`, and `ExportToBMP()`. In this recipe, you'll learn how to export your maps to images.

How to do it...

Follow these steps to learn how to export your data or layout view to an image file:

1. Open `C:\ArcpyBook\Ch5\Crime_Ch5.mxd` in ArcMap.
2. Open the Python window.
3. Import the `arcpy.mapping` module:
   ```
   import arcpy.mapping as mapping
   ```
4. Reference the currently active document (`Crime_Ch5.mxd`), and assign the reference to a variable:
   ```
   mxd = mapping.MapDocument("CURRENT")
   ```

5. Export the `Crime` data frame as a JPEG image:

```
for df in mapping.ListDataFrames(mxd):
  if df.name == "Crime":
    mapping.ExportToJPEG(mxd,r"c:\ArcpyBook\Ch5\DataFrameCrime.
jpg",df)
```

6. Run the script and examine the output file.
7. Now, we'll use an optional parameter that outputs a `world` file along with the image. Alter your script, so that it appears as follows:

```
import arcpy.mapping as mapping
mxd = mapping.MapDocument("CURRENT")
for df in mapping.ListDataFrames(mxd):
  if df.name == "Crime":
    mapping.ExportToPDF(mxd,r"c:\ArcpyBook\Ch5\DataFrameCrime2.
jpg",df,world_file=True)
```

8. Run the script. A new file called `DataFrameCrime.jpg` will be created. Open this file in a text editor, and you should see the following content:

```
470.520239851286190
0.000000000000000
0.000000000000000
-496.256971063348490
1972178.761771137800000
13815440.387677660000000
```

How it works...

Notice that the `ExportToJPEG()` function looks virtually the same as `ExportToPDF()`. Keep in mind though that the optional parameters will be different for all the export functions. Each `ExportTo<Type>` function will vary depending upon the parameters that can be used in the creation of the image file.

Creating a map book with PDFDocumentCreate() and PDFDocumentOpen()

One common scenario for many GIS professionals is the need to create a map book that can then be shared with others. A map book is simply a collection of maps for a particular area and usually contains an index map as well. Map books are often created with PDF files, as they are a common interchange format.

Getting ready

In addition to exporting your maps to PDF, you can also manipulate existing PDF documents or create new PDF documents. You can merge pages, set document open behavior, add file attachments, and create or change document security settings. The `PDFDocumentOpen()` function is used to open an existing PDF file for manipulation. `PDFDocumentCreate()` creates a new PDF document. These functions are often used in the creation of map books, which is what we'll do in this recipe.

How to do it...

1. Open IDLE and create a new script window.
2. Import the `arcpy.mapping` and `os` modules:
   ```
   import arcpy.mapping as mapping
   import os
   ```
3. Set the path and filename for the map book. Remove the file if it already exists:
   ```
   pdfPath = r"C:\ArcpyBook\Ch5\CrimeMapBook.pdf"
   if os.path.exists(pdfPath):
     os.remove(pdfPath)
   ```
4. Create the PDF document:
   ```
   pdfDoc = mapping.PDFDocumentCreate(pdfPath)
   ```
5. Append existing PDF pages to the document:
   ```
   pdfDoc.appendPages(r"c:\ArcpyBook\Ch5\Map_PageLayout.pdf")
   pdfDoc.appendPages(r"c:\ArcpyBook\Ch5\Map_DataFrameCrime.pdf")
   ```
6. Commit changes to the map book:
   ```
   pdfDoc.saveAndClose()
   ```
7. The entire script should appear as follows:
   ```
   import arcpy.mapping as mapping
   import os
   pdfPath = r"C:\ArcpyBook\Ch5\CrimeMapBook.pdf"
   if os.path.exists(pdfPath):
     os.remove(pdfPath)
   pdfDoc = mapping.PDFDocumentCreate(pdfPath)
   pdfDoc.appendPages(r"c:\ArcpyBook\Ch5\Map_PageLayout.pdf")
   pdfDoc.appendPages(r"c:\ArcpyBook\Ch5\Map_DataFrameCrime.pdf")
   pdfDoc.saveAndClose()
   ```

8. Run the script.
9. Examine the new map book at `c:\ArcpyBook\Ch5\CrimeMapBook.pdf`. The book should now contain two pages consisting of the PDF files that we created in an earlier recipe.

How it works...

The `PDFDocumentCreate()` function is used to create a new PDF document by providing a path and filename for the document. The PDF is not actually created on disk until you insert or append pages and then call `PDFDocument.saveAndClose()`. The `appendPages()` and `insertPages()` functions are used to insert and append pages.

`PDFDocumentOpen()` accepts a parameter that specifies the path to a PDF file and returns an instance of the `PDFDocument` class. Once open, you can make modifications to PDF file properties, add or insert files, and attach documents. Make sure you call `PDFDocument.saveAndClose()` after all operations, to save the changes.

There's more...

A number of properties can be set on a PDF document through the `PDFDocument` object, including getting the page count, attaching files, updating title, author, subject, keywords, open behavior, and the layout. You can also update document security by calling `PDFDocument.updateDocSecurity()` to set a password, encryption, and security restrictions.

6 Executing Geoprocessing Tools from Scripts

In this chapter, we will cover the following recipes:

- Finding geoprocessing tools
- Retrieving a toolbox alias
- Executing geoprocessing tools from a script
- Using the output of a tool as an input to another tool
- Setting environment variables

Introduction

ArcGIS Desktop contains over 800 geoprocessing tools, which can be used in your Python scripts. In this chapter, you will learn to use these tools in your scripts. Each tool has unique characteristics. The syntax for executing each will differ depending upon the type of input required to successfully execute the tool. We'll examine how you can determine the input parameters for any tool using the ArcGIS Desktop help system. The execution of a tool results in the creation of one or more output datasets along with a set of messages that are generated while the tool is running. We'll examine how you can use these messages. Finally, we'll look at how you can get and set environment variables for your script.

Finding geoprocessing tools

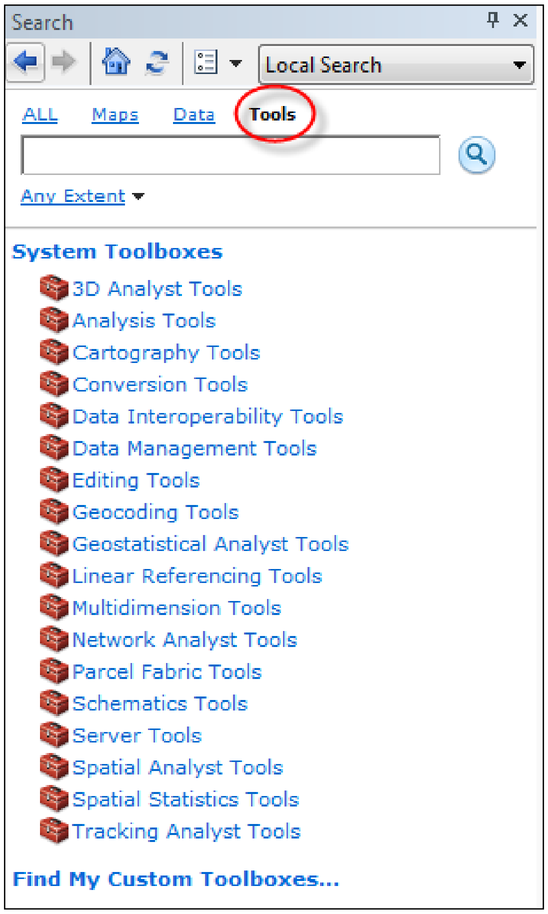
Before using a tool in your geoprocessing script, you will need to make sure that you have access to that tool, based on the current license level of ArcGIS Desktop that you are running or that your end users will run. This information is contained within the ArcGIS Desktop help system.

Getting ready

The availability of geoprocessing tools for your script is dependent upon the ArcGIS license level you are using. At version 10.1 of ArcGIS Desktop, there are three license levels, namely **Basic**, **Standard**, and **Advanced**. These were formerly known as **ArcView**, **ArcEditor**, and **ArcInfo**, respectively. It is important for you to understand the license level required for the tool that you want to use in your script. In addition to this, the use of extensions in ArcGIS Desktop can result in the availability of additional tools for your script. There are two primary ways to find tools in ArcGIS Desktop. The first is to use the search window and the second is to simply browse the contents of ArcToolbox. In this recipe, you will learn to use the search window to find available geoprocessing tools that can be used in your scripts.

How to do it…

1. Open `C:\ArcpyBook\Ch6\Crime_Ch6.mxd` in ArcMap.
2. From the **Geoprocessing** menu item select **Search For Tools**. This will display the **Search** window, as shown in the following screenshot. By default, you will be searching for **Tools**:



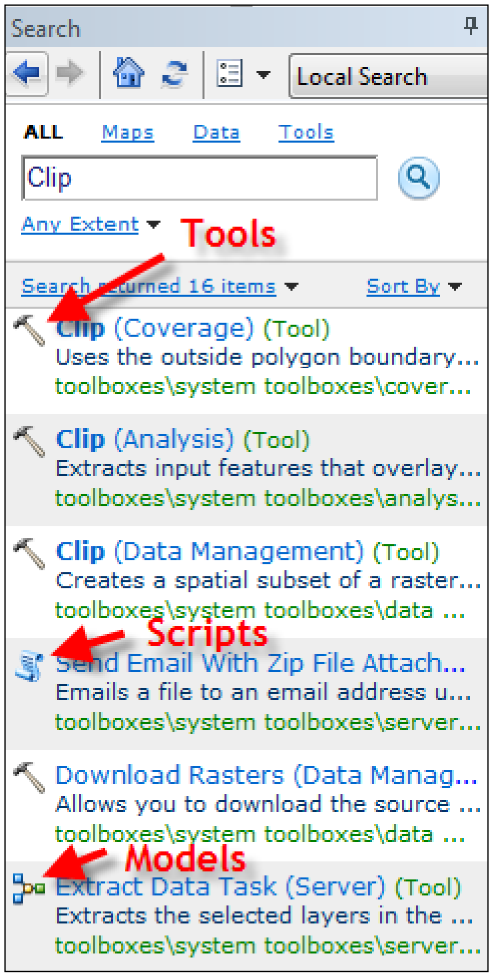
System Toolboxes

- 3D Analyst Tools
- Analysis Tools
- Cartography Tools
- Conversion Tools
- Data Interoperability Tools
- Data Management Tools
- Editing Tools
- Geocoding Tools
- Geostatistical Analyst Tools
- Linear Referencing Tools
- Multidimension Tools
- Network Analyst Tools
- Parcel Fabric Tools
- Schematics Tools
- Server Tools
- Spatial Analyst Tools
- Spatial Statistics Tools
- Tracking Analyst Tools

Find My Custom Toolboxes...

3. Type the term `Clip` into the search text box. As you begin typing this word, the **Search** textbox will automatically filter the results based upon the first few letters you type. You'll notice that for the word `Clip`, there are three possible tools: `clip(analysis)`,`clip(coverage)`, `clip(data_management)`. There are a number of cases where there are several geoprocessing tools with the same name. To uniquely define a tool, the toolbox alias is attached to the tool name. We'll examine toolbox aliases in greater detail in the next recipe.

4. For now, click on the **Search** button to generate a list of matching tools. The search should generate a list similar to what you see in the following screenshot. Tools are indicated with a hammer icon in the search results. You'll also see a couple of other icons in the search results. The scroll icon indicates a Python script, and an icon containing multi-colored squares indicates a model:



Clip (Coverage) (Tool)
Uses the outside polygon boundary...
toolboxes\system toolboxes\cover...

Clip (Analysis) (Tool)
Extracts input features that overlay...
toolboxes\system toolboxes\analys...

Clip (Data Management) (Tool)
Creates a spatial subset of a raster...
toolboxes\system toolboxes\data ...

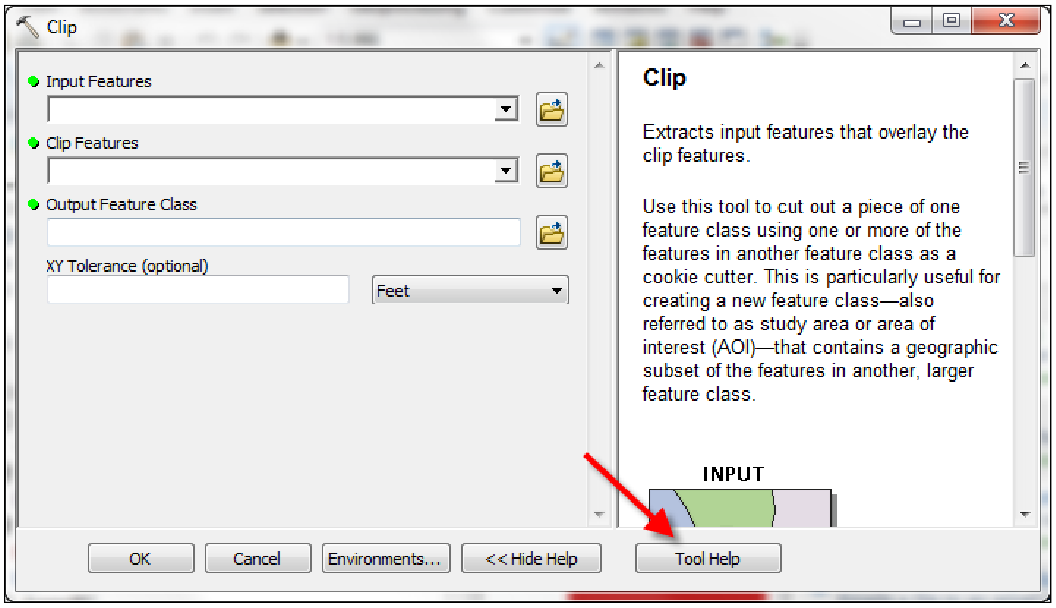
Send Email With Zip File Attach...
Emails a file to an email address u...
toolboxes\system toolboxes\server...

Download Rasters (Data Manag...
Allows you to download the source ...
toolboxes\system toolboxes\data ...

Extract Data Task (Server) (Tool)
Extracts the selected layers in the ...
toolboxes\system toolboxes\server...

5. Select the **Clip (Analysis)** tool. This will open the dialog box for the **Clip (Analysis)** tool. This isn't all that useful to you as a script programmer. You will probably be more interested in the ArcGIS Desktop help for a particular tool.

6. Click on the **Tool Help** button at the bottom of the tool dialog box to display detailed information about this particular tool:



Clip

Extracts input features that overlay the clip features.

Use this tool to cut out a piece of one feature class using one or more of the features in another feature class as a cookie cutter. This is particularly useful for creating a new feature class—also referred to as study area or area of interest (AOI)—that contains a geographic subset of the features in another, larger feature class.

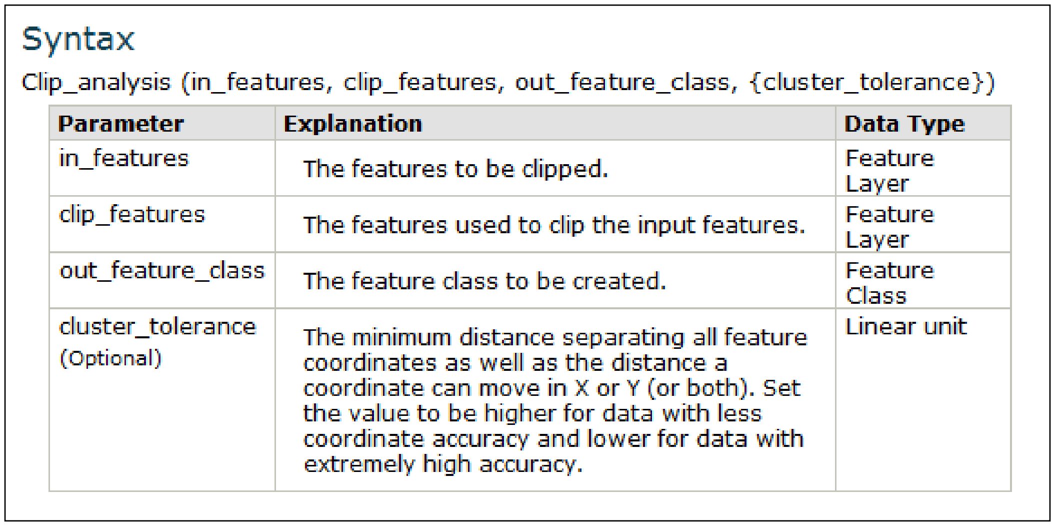
7. Scroll down to the bottom of the help page for the **Clip** tool to examine the syntax for this particular tool.

How it works...

The help system contains a summary, illustration, usage, syntax, code samples, available environment variables, related topics, and licensing information for each tool. As a geoprocessing script programmer, you will primarily be interested in the syntax, code samples, and licensing information sections near the bottom.

You should always examine the licensing information section at the bottom of the help documentation for each tool, to make sure you have the appropriate license level to use the tool.

The syntax section contains information about how this tool should be called from your Python script, including the name of the tool and the required and optional input parameters. All the parameters will be enclosed within parentheses. The required parameters for the `Clip` tool are `in_features`, `clip_features`, and `out_feature_class`. When you call this tool from your script, you will be required to provide these parameters to the tool for it to execute correctly. The fourth parameter is an optional parameter called `cluster_tolerance`. Parameters marked as optional in the syntax are surrounded by curly braces. The following screenshot provides an example of an optional parameter surrounded by curly braces. This doesn't mean that you enclose the parameter in curly braces when you call the tool. It is in the help section simply to indicate that this parameter is optional when being called from your geoprocessing script:



Syntax

Clip_analysis (in_features, clip_features, out_feature_class, {cluster_tolerance})

Parameter	Explanation	Data Type
in_features	The features to be clipped.	Feature Layer
clip_features	The features used to clip the input features.	Feature Layer
out_feature_class	The feature class to be created.	Feature Class
cluster_tolerance (Optional)	The minimum distance separating all feature coordinates as well as the distance a coordinate can move in X or Y (or both). Set the value to be higher for data with less coordinate accuracy and lower for data with extremely high accuracy.	Linear unit

Retrieving a toolbox alias

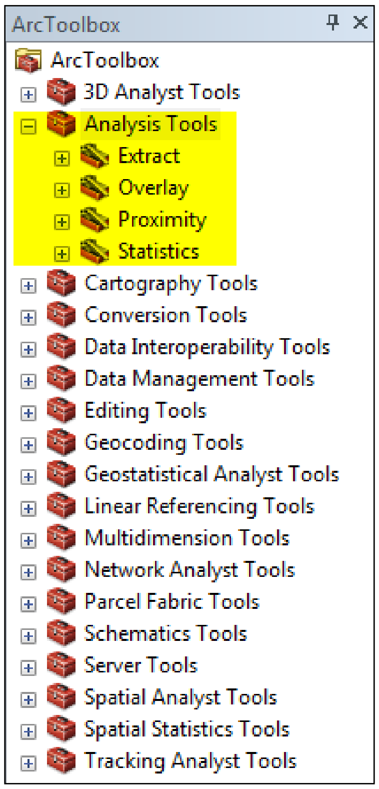
All toolboxes have an alias which, when combined with the tool name, provides a unique reference to any tool in ArcGIS Desktop. This alias is necessary because a number of tools have the same name. When referencing a tool from your Python script, it is necessary to reference both the tool name and tool alias.

Getting ready

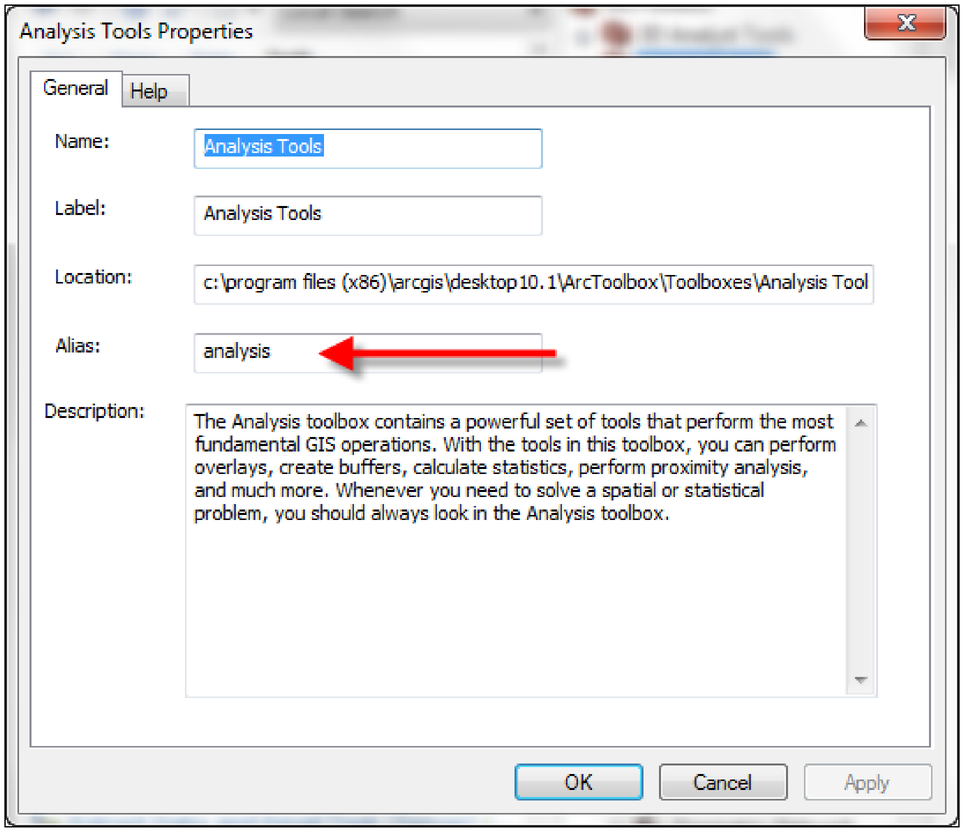
In the last recipe we looked at the **Clip** tool. There are actually three Clip tools which can be found in the **Analysis, Coverage**, and **Data Management** toolboxes. Each **Clip** tool performs a different function. For instance, the **Clip** tool in the **Analysis** toolbox clips a vector feature class using an input feature, while the **Clip** tool in the **Data Management** toolbox is used to create a spatial subset of a raster. Since it is possible to have multiple tools with the same name, we can uniquely identify a particular tool by providing both the toolname and the toolbox alias where the tool resides. In this recipe you will learn how to find the alias of a toolbox.

How to do it...

1. Open `C:\ArcpyBook\Ch6\Crime_Ch6.mxd` in ArcMap.
2. If necessary, open **ArcToolbox**.
3. Find the **Analysis Tools** toolbox, as shown in the following screenshot:



4. Right-click on the **Analysis Tools** toolbox and select **Properties**. This will display the **Analysis Tools Properties** dialog, as shown in the following screenshot. The **Alias** textbox will contain the alias:



Name: Analysis Tools

Label: Analysis Tools

Location: c:\program files (x86)\arcgis\desktop10.1\ArcToolbox\Toolboxes\Analysis Tool

Alias: analysis

Description: The Analysis toolbox contains a powerful set of tools that perform the most fundamental GIS operations. With the tools in this toolbox, you can perform overlays, create buffers, calculate statistics, perform proximity analysis, and much more. Whenever you need to solve a spatial or statistical problem, you should always look in the Analysis toolbox.

How it works...

You can follow this process to find the alias name of any toolbox. In a Python script, you can execute a tool by referring to the tool with the syntax `<toolname>_<toolbox alias>`. For example, if you were calling the **Buffer** tool, it would be `buffer_analysis`. Toolbox aliases are invariably simple. They are typically one word and do not include dashes or special characters. In the next recipe, we'll create a simple script that follows this format for executing a tool.

Executing geoprocessing tools from a script

Once you have determined the toolbox alias and then verified the accessibility of the tool based on your current license level, you are ready to add the execution of the tool to a script.

Getting ready

Now that you understand how to find the tools that are available and how to uniquely reference them, the next step is to put this together and execute a tool from a geoprocessing script.
In this recipe, you can then execute the tool from your script.

How to do it...

1. Open `C:\ArcpyBook\Ch6\Crime_Ch6.mxd` in ArcMap.
2. Click on the **Add Data** button and add the `EdgewoodSD.shp` file to the table of contents.
3. Turn off the **Crime Density by School District** and **Burglaries in 2009** layers to get a better view of the **EdgewoodSD** layer. There is only one polygon feature in this file. It represents the Edgewood School District. Now we're going to write a script that clips the **Burglaries in 2009** features to this school district.
4. Open the Python window in ArcMap.
5. Import the `arcpy` module:

   ```
   import arcpy
   ```

6. Create a variable that references the input feature class to be clipped:

   ```
   in_features = "c:/ArcpyBook/data/CityOfSanAntonio.gdb/Burglary"
   ```

7. Create a variable that references the layer to be used for the clip:

   ```
   clip_features = "c:/ArcpyBook/Ch6/EdgewoodSD.shp"
   ```

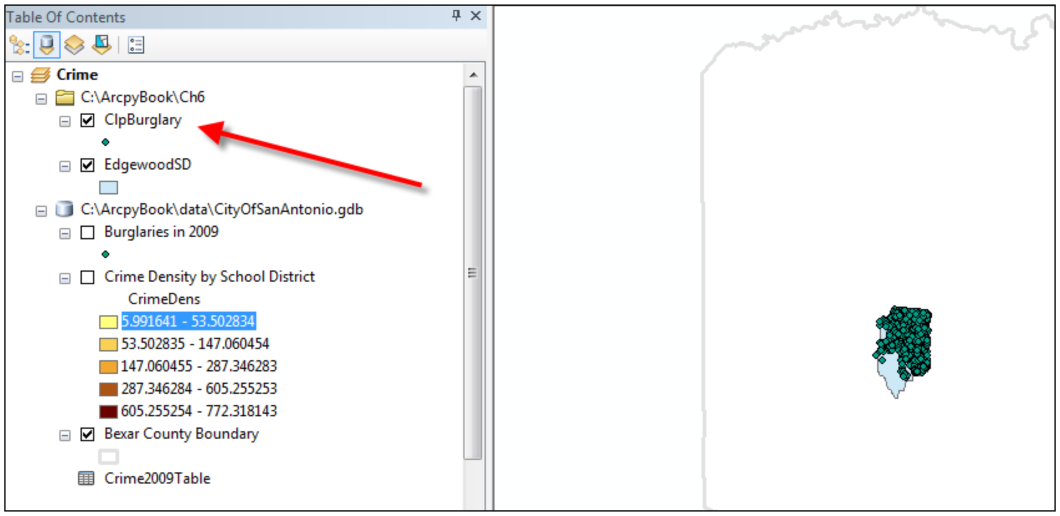
8. Create a variable that references the output feature class:

   ```
   out_feature_class = "c:/ArcpyBook/Ch6/ClpBurglary.shp"
   ```

9. Execute the `Clip` tool from the **Analysis Tools** toolbox:

   ```
   arcpy.Clip_analysis(in_features,clip_features,out_feature_class)
   ```

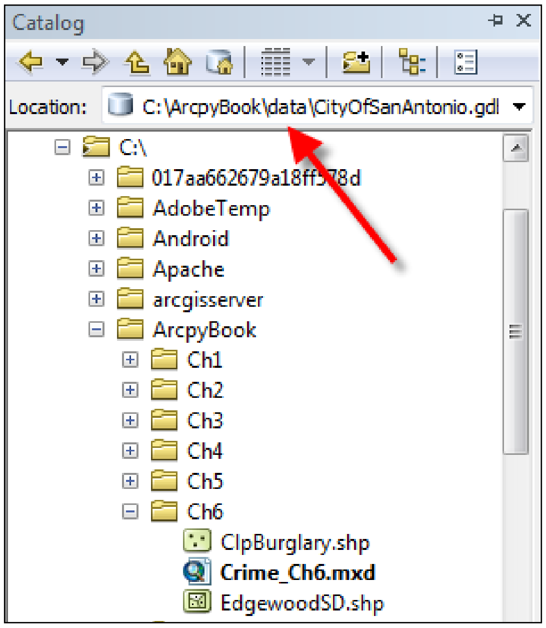
10. Run the script. The output feature class containing only those burglary points within the Edgewood school district should be added to the data frame, as shown in the following screenshot:



How it works...

The primary line of code of interest in this recipe is the final line that executes the `Clip` tool. Notice that we called this tool by specifying a syntax of `Clip_analysis`, which gives us a reference to the `Clip` tool in the **Analysis Tools** toolbox, which has an alias of `analysis`. We've also passed in three parameters that reference the input feature class, clip feature class, and output feature class. I should point out that we hardcoded the paths to each of the datasets. This is not a good programming practice, but in this particular instance I just wanted to illustrate how you execute a tool. A future chapter will illustrate how you can remove the hardcoding in your scripts and make them much more versatile.

Most tools that you use will require paths to data sources. This path must be the same as the path reported on the ArcCatalog **Location** toolbar, as shown in the following screenshot:



Tools use ArcCatalog to find geographic data using an ArcCatalog path. This path is a string and is unique to each dataset. The path can include folder locations, database connections, or a URL. So, it is important to check the path using ArcCatalog before attempting to write Python scripts against the data. ArcSDE paths require special consideration. Many ArcSDE users do not have standardized connection names, which can cause issues when running models or scripts.

There's more...

Geoprocessing tools are organized in two ways. You can access tools as functions on `arcpy` or as modules matching the toolbox alias name. In the first case, when tools are accessible as functions from `arcpy`, they are called in the format that you followed in this recipe. The tool name is followed by an underscore and then the toolbox alias. In the second form, tools are called as functions of a module, which takes the name of the toolbox alias. Here, `analysis` is the toolbox alias, so it becomes a module. `Clip` is a function of that module and is called follows:

```
arcpy.analysis.Clip (in_features,clip_features,out_feature_class)
```

Which method you use is really a matter of preference. They both accomplish the same thing, which is the execution of a geoprocessing tool.

Using the output of a tool as an input to another tool

There will be many occasions when you will need to use the output of one tool as an input to another tool. This is called tool chaining. An example of tool chaining could involve buffering a streams layer and then finding all residential properties that fall within the buffer. In this case, the Buffer tool would output a new layer, which would then be used as an input to the Select by Location tool or one of the other overlay tools. In this recipe, you will learn how to obtain the output of a tool and use it as input to another tool.

Getting ready

The Buffer tool creates an output feature class from an input feature layer using a specified distance. This output feature class can be stored in a variable, which can then be used as an input to another tool, such as the `Select Layer by Location` tool. In this recipe, you will learn how to use the output from the `Buffer` tool as an input to the `Select Layer by Location` tool to find all schools that are within a half mile of a stream.

How to do it…

Follow these steps to learn how to access the currently active map document in ArcMap:

1. Open ArcMap with a new map document file (`.mxd`).
2. Click on the **Add Data** button and add the Streams and Schools shapefiles from `c:\ArcpyBook\data\TravisCounty`.
3. Click on the Python window button.
4. Import the `arcpy` module:
   ```
   import arcpy
   ```
5. Set the workspace:
   ```
   arcpy.env.workspace = "c:/ArcpyBook/data/TravisCounty"
   ```
6. Start a `try` statement and add variables for the streams, buffered streams layer, distance, and schools:
   ```
   try:
     # Buffer areas of impact around major roads
     streams = "Streams.shp"
     streamsBuffer = "StreamsBuffer.shp"
     distance = "2640 Feet"
     schools2mile = "Schools.shp"
   ```

7. Call the `Buffer` tool and pass in variables for the streams layer, the buffered stream layer, and the distance along with several variables that control the look of the buffer:

```
arcpy.Buffer_analysis(streams, streamsBuffer,
distance,'FULL','ROUND','ALL')
```

8. Create a temporary layer for the schools using the `Make Feature Layer` tool:

```
arcpy.MakeFeatureLayer_management(schools2mile, 'Schools2Mile_lyr')
```

9. Select all schools within a half mile of a stream using the `Select Layer by Location` tool:

```
arcpy.SelectLayerByLocation_management('Schools2Mile_lyr',
'intersect', streamsBuffer)
```

10. Add the `except` block to catch any errors:

```
except:
  print 'Error in script'
```

11. The entire script should appear as follows:

```
import arcpy
arcpy.env.workspace = "c:/ArcpyBook/data/TravisCounty"
try:
  # Buffer areas of impact around major roads
  streams = "Streams.shp"
  streamsBuffer = "StreamsBuffer.shp"
  distance = "2640 Feet"
  schools2mile = "Schools.shp"


  arcpy.Buffer_analysis(streams, streamsBuffer,
distance,'FULL','ROUND','ALL')

  # Make a layer
  arcpy.MakeFeatureLayer_management(schools2mile, 'Schools2Mile_
lyr')
  arcpy.SelectLayerByLocation_management('Schools2Mile_lyr',
'intersect', streamsBuffer)
except:
  print 'Error in script'
```

How it works...

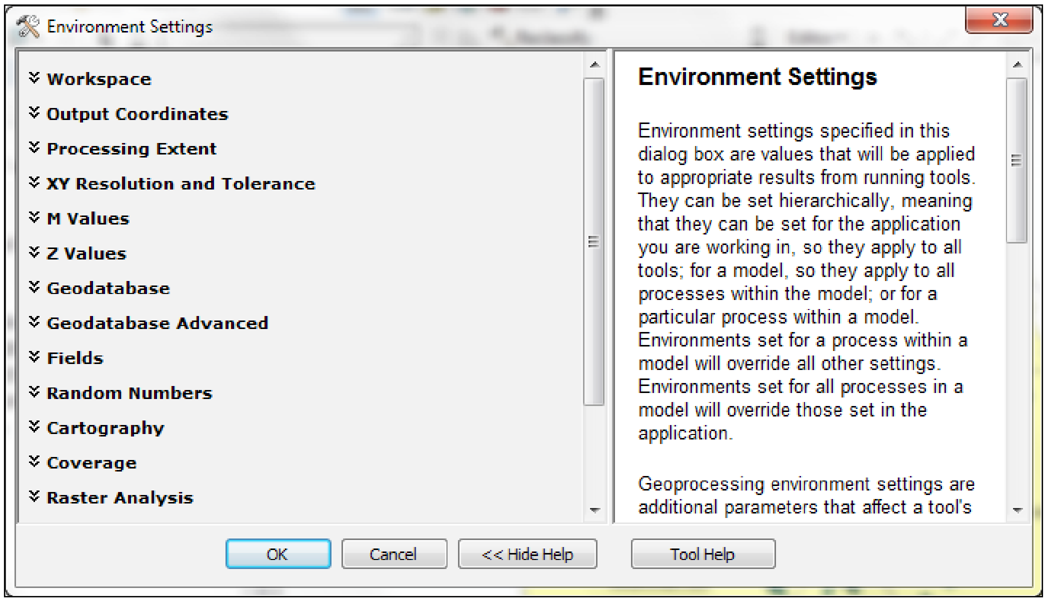
The `Buffer` tool creates an output feature class, which we called `StreamsBuffer.shp` and stored in a variable called `streamsBuffer`. This `streamsBuffer` variable is then used as an input to the `Select Layer by Location` tool as the third parameter being passed in to the function. Using the output of one tool simply requires that you create a variable to hold the output data and then it can be re-used as needed in other tools.

Setting environment variables and examining tool messages

Environment variables provide additional parameters that can be set, which serve as global variables accessible at various levels including your scripts. Your scripts can get environment variable values as well as set values. You need to have an understanding of the environment variables that are available to your script as well as how they can be accessed. In addition to this, tools generate messages during execution. These messages come in several varieties.

Getting ready

Environment settings are additional parameters available to your script during execution. These are the values that you set once at the ArcGIS Desktop application level using a separate dialog box accessed through the **Geoprocessing - Environments** menu item, and are organized by category:



Environment Settings

Environment settings specified in this dialog box are values that will be applied to appropriate results from running tools. They can be set hierarchically, meaning that they can be set for the application you are working in, so they apply to all tools; for a model, so they apply to all processes within the model; or for a particular process within a model. Environments set for a process within a model will override all other settings. Environments set for all processes in a model will override those set in the application.

Geoprocessing environment settings are additional parameters that affect a tool's

These settings are very similar to environment variable settings that you can set at an operating-system level, but they are specific to the ArcGIS geoprocessing framework. These application-level environment settings are the highest level, and default settings for all the tools are applied during the execution of any tool. In addition to application-level environment settings, you can also apply environment settings at the tool level. Environment settings at the tool level directly inherit the settings applied at the application level. However, these settings can be overridden at the tool level. Tool-level settings are applicable only to the current tool execution. Your Python scripts can get and set environment settings through the `env` class in `arcpy`. These are read/write properties. Both application- and tool-level settings are passed into the script and will apply to any tool that you run from within the script. You may also override any environment settings from within your script and these will be applied during the execution of the script. Please keep in mind that just as with tools, environment settings set at the script level only apply to the current execution of a script. There are, however, two occasions when environment settings are not passed to the script. These include scripts running outside an ArcGIS application, such as when they are run from the operating system command prompt. In addition, when a script calls another script, environment settings are not passed through. In this recipe, you will learn to set environment settings from your Python script and view various messages generated by the tool during execution.

How to do it...

Follow these steps to learn how to set environment settings and generate messages in your script:

1. Create a new IDLE script and save it as `c:\ArcpyBook\Ch6\SetEnvVariables.py`.
2. Import the `arcpy` module:
   ```
   import arcpy
   ```
3. Set the workspace using an environment variable:
   ```
   arcpy.env.workspace = "c:/ArcpyBook/Ch6"
   ```
4. Call the `Buffer` tool, passing in the input dataset of `Streams.shp`, an output dataset of `Streams_Buff.shp`, and a distance of 200 feet.
   ```
   arcpy.Buffer_analysis("Streams.shp","Streams_Buff,’200 Feet’")
   ```
5. Save the script.

How it works...

Environment variables can be set at both the application level and the tool level. Application-level environment settings are similar to global environment settings; in that they affect all tools. On the other hand, environment settings defined at the tool level affect only the current run of a tool. Both can be set using ArcGIS Desktop. Your scripts can also set environment variables, which are applicable only during the execution of the script. They are similar to environment variables set at the tool level. Probably, the most commonly set environment variable used in scripts is the `env.workspace` variable, which defines the current working directory for the script. Setting this variable at the top of your script can make your code less wordy, since you won't have to constantly refer to the full path to a dataset, but rather simply the dataset name as defined in the current workspace.

7
Creating Custom Geoprocessing Tools

In this chapter, we will cover the following recipe:

- Creating a custom geoprocessing tool

Introduction

In addition to accessing the system tools provided by ArcGIS, you can also create your own custom tools. These tools work in the same way that system tools do and can be used in ModelBuilder, Python window, or in standalone Python scripts. Many organizations build their own library of tools that perform geoprocessing operations specific to their data.

Creating a custom geoprocessing tool

In addition to being able to execute any of the available tools in your scripts, you can also create your own custom tools, which can also be called from a script. Custom tools are frequently created to handle geoprocessing tasks that are specific to an organization. These tools can easily be shared as well.

Getting ready

In this recipe, you will learn to create custom geoprocessing script tools by attaching a Python script to a custom toolbox in ArcToolbox. There are a number of advantages of creating a custom script tool. When you take this approach, the script becomes a part of the geoprocessing framework, which means that it can be run from a model, command line, or another script. In addition to this, the script has access to ArcMap environment settings and help documentation. Other advantages include a nice, easy-to-use user interface and error-prevention capabilities. Error-prevention capabilities provided include a dialog box that informs the user of certain errors.

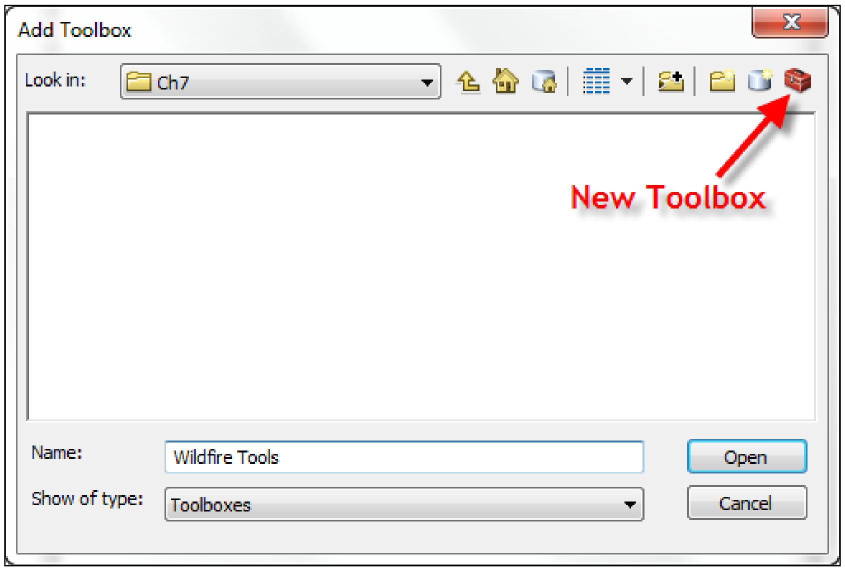
These custom developed script tools must be added to a custom toolbox that you create because the system toolboxes provided with ArcToolbox are read-only toolboxes and thus can't accept new tools.

In this recipe, you are going to be provided with a pre-written Python script that reads wildfire data from a comma-delimited text file, and writes this information to a point feature class called `FireIncidents`. References to these datasets have been hardcoded, so you are going to alter the script to accept dynamic variable input. You'll then attach the script to a custom tool in ArcToolbox to give your end users a visual interface for using the script.

How to do it...

The custom Python geoprocessing scripts that you write can be added to ArcToolbox inside custom toolboxes. You are not allowed to add your scripts to any of the system toolboxes, such as **Analysis** or **Data Management**. However, by creating a new custom toolbox, you can add these scripts.

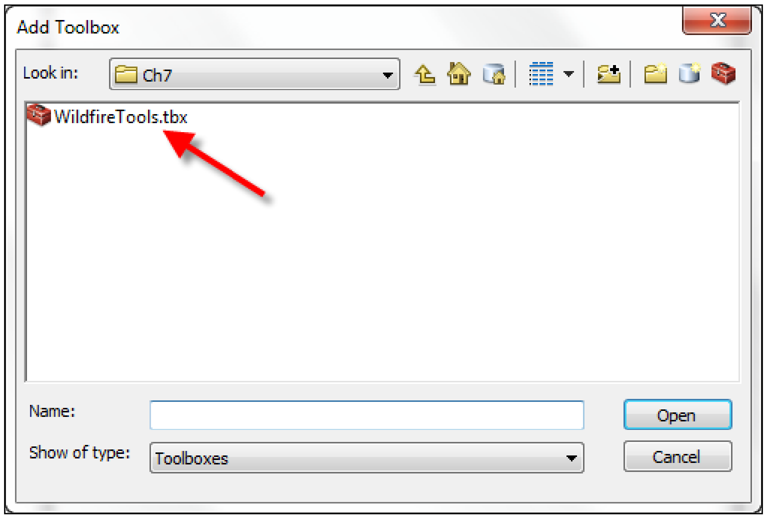
1. Open ArcMap with an empty map document file and open the ArcToolbox window.
2. Right-click anywhere in the white space area of ArcToolbox and select **Add Toolbox**. On the **Add Toolbox** dialog box, click on the **New Toolbox** button. This will create a new toolbox with a default name of `Toolbox.tbx`; you will rename the toolbox in the next step:



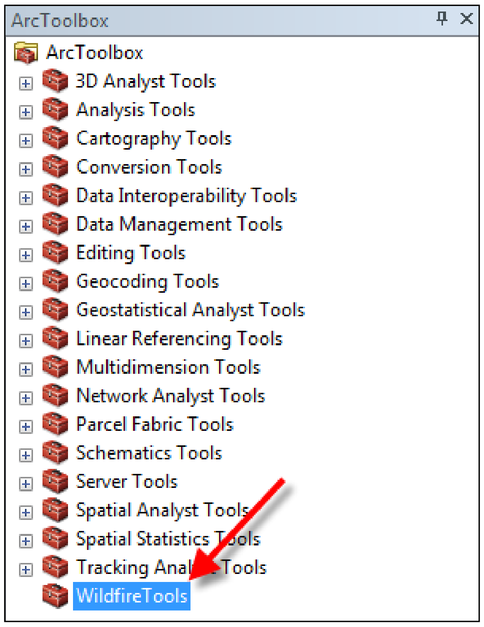
Name: Wildfire Tools

Show of type: Toolboxes

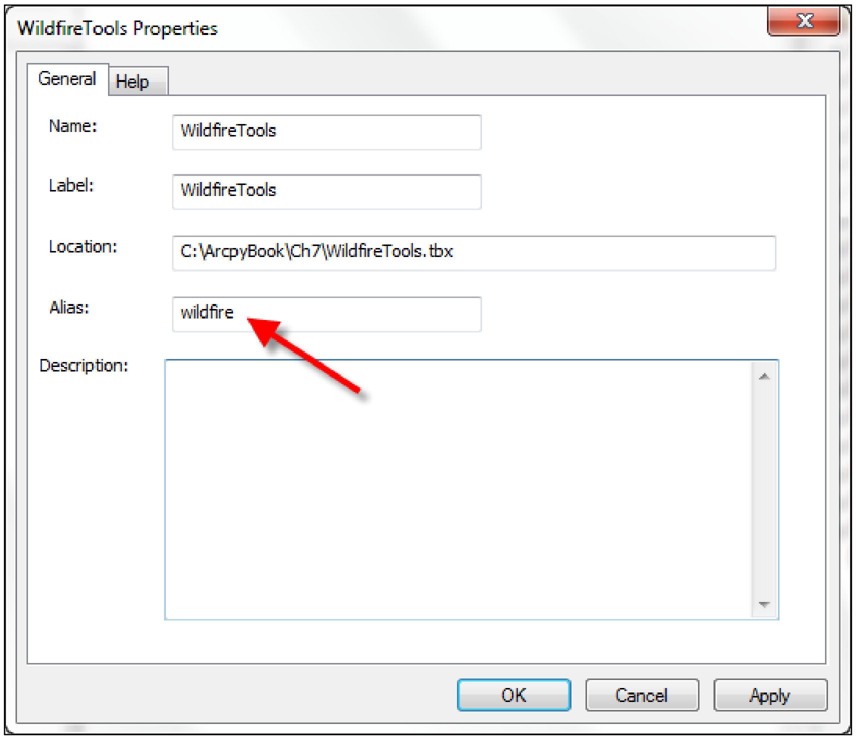
3. Navigate to the `c:\ArcpyBook\Ch7` folder and name the toolbox `Wildfire Tools`:



4. Open the toolbox by selecting **WildfireTools.tbx** and clicking on the **Open** button. The toolbox should now be displayed in ArcToolbox as shown in the following screenshot:



5. Each toolbox should be given a name and an alias. The alias will be used to uniquely define your custom tool. Alias names should be kept short and should not include any special characters. Right-click on the new toolbox and select **Properties**. Add an alias of `wildfire` as shown in the following screenshot:



Name: WildfireTools
Label: WildfireTools
Location: C:\ArcpyBook\Ch7\WildfireTools.tbx
Alias: wildfire
Description:

You can optionally create a new toolset inside this toolbox by right-clicking on the toolbox and selecting **New** | **Toolset**. Toolsets allow you to functionally group your scripts. In this example, it won't be necessary to do this, but if you need to group your scripts in the future, then this is how you can accomplish it.

6. In this next step, we will alter an existing Python script called `InsertWildfires.py` to accept dynamic inputs that will be provided by the user of the tool through the ArcToolbox interface. Open `c:\ArcpyBook\Ch7\InsertWildfires.py` in IDLE.

   Notice that we have hardcoded the path to our workspace as well as the comma-delimited text file containing the wildland fire incidents:

```
arcpy.env.workspace = "C:/ArcpyBook/data/Wildfires/WildlandFires.
mdb"
  f = open("C:/ArcpyBook/data/Wildfires/
NorthAmericaWildfires_2007275.txt","r")
```

7. Delete the preceding two lines of code.

   In addition, we have also hardcoded the output feature class name:

   ```
   cur = arcpy.InsertCursor("FireIncidents")
   ```

   This hardcoding limits the usefulness of our script. If the datasets move or are deleted, the script will no longer run. In addition to this, the script lacks the flexibility to specify different input and output datasets. In the next step, we will remove this hardcoding and replace it with the ability to accept dynamic input.

8. We will use the `GetParameterAsText()` function found in `arcpy` to accept dynamic input from the user. Add the following lines of code to your `try` block, so that your code appears as follows:

   ```
   try:
     # the output feature class name
     outputFC = arcpy.GetParameterAsText(0)

     # the template feature class that defines the attribute schema
     fClassTemplate = arcpy.GetParameterAsText(1)

     # open the file to read
     f = open(arcpy.GetParameterAsText(2),'r')

     arcpy.CreateFeatureclass_management(os.path.split(outputFC)[0],
   os.path.split(outputFC[1]), "point", fClassTemplate)
   ```

   Notice that we call the `CreateFeatureClass` tool, found in the **Data Management Tools** toolbox, passing in the `outputFC` variable along with the template feature class (`fClassTemplate`). This tool will create the empty feature class containing the output feature class defined by the user.

9. You will also need to alter the line of code that creates an `InsertCursor` object. Change the line as follows:

   ```
   cur = arcpy.InsertCursor(outputFC)
   ```

10. The entire script should appear as follows:

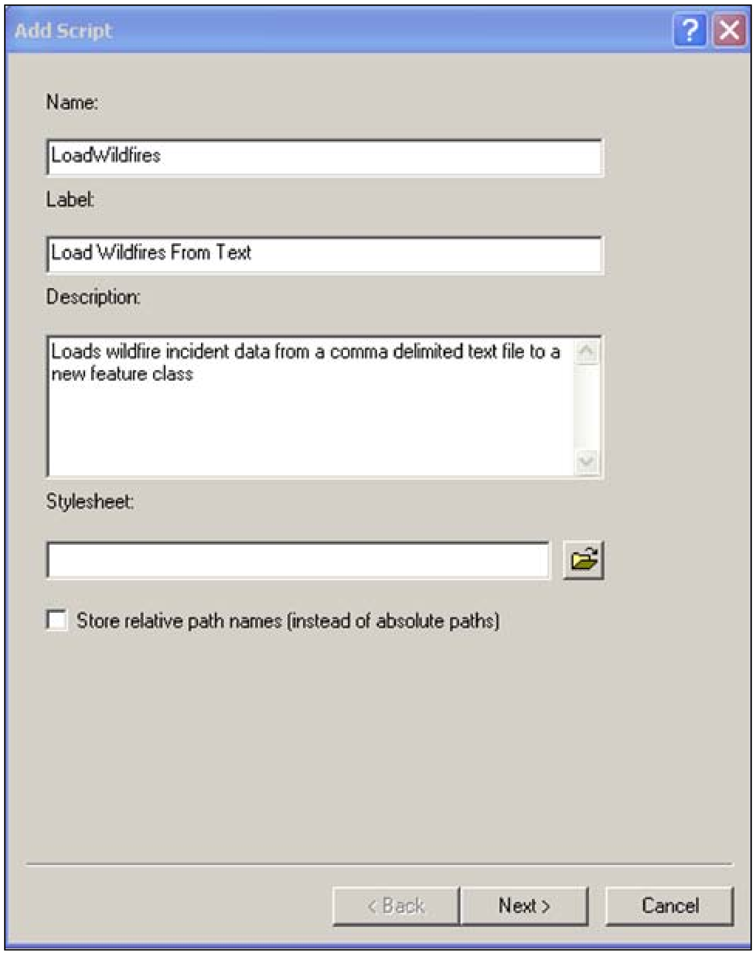
    ```
    import arcpy, os
    try:
          outputFC = arcpy.GetParameterAsText(0)
    ```

```
  fClassTemplate = arcpy.GetParameterAsText(1)
       f = open(arcpy.GetParameterAsText(2),'r')
       arcpy.CreateFeatureclass_management(os.path.split(outputFC)
[0], os.path.split(outputFC[1]), "point", fClassTemplate)
  lstFires = f.readlines()
  cur = arcpy.InsertCursor(outputFC)
  cntr = 1
  for fire in lstFires:
    if 'Latitude' in fire:
      continue
    vals = fire.split(",")
    latitude = float(vals[0])
    longitude = float(vals[1])
    confid = int(vals[2])
    pnt = arcpy.Point(longitude, latitude)
    feat = cur.newRow()
    feat.shape = pnt
    feat.setValue("CONFIDENCEVALUE", confid)
    cur.insertRow(feat)
    arcpy.AddMessage("Record number: " + str(cntr) + " written to
feature class")
    cntr = cntr + 1
except:
  print arcpy.GetMessages()
finally:
  del cur
  f.close()
```

In the next step, we will add the script that we just created to the **Wildfire Tools** toolbox as a script tool.

11. In ArcToolbox, right-click on the **Wildfire Tools** custom toolbox that you created earlier and select **Add | Script**. This will display the **Add Script** dialog, as shown in the following screenshot. Give your script a name, label, and description. The **Name** field can not contain any spaces or special characters. **Label** is the name that shows up next to the script. For this example, give it a label of `Load Wildfires From Text`. Finally, add some descriptive information that details the operations the script will perform.

12. See the following screenshot for **Name**, **Label**, and **Description** details:



Name:
LoadWildfires

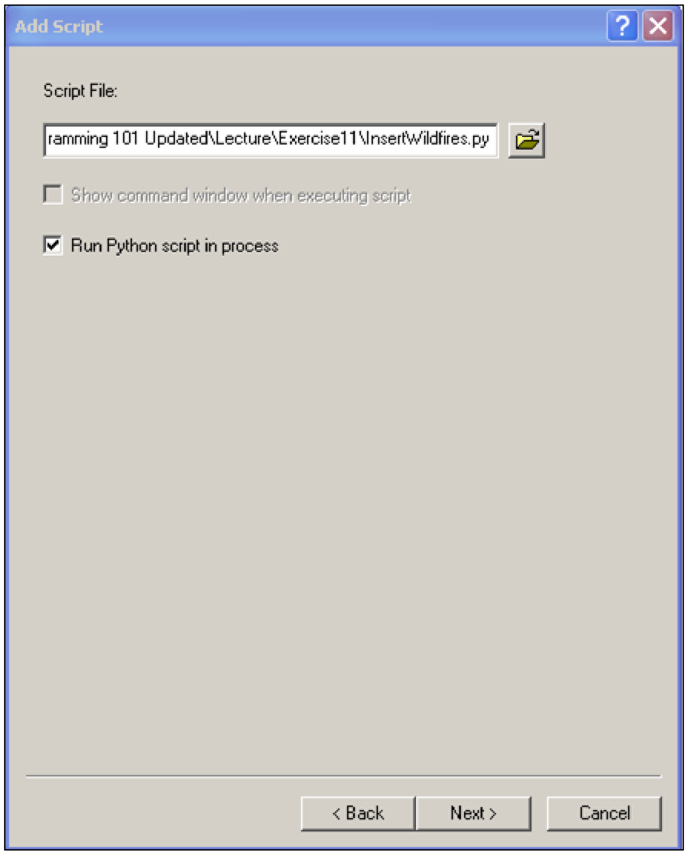
Label:
Load Wildfires From Text

Description:
Loads wildfire incident data from a comma delimited text file to a new feature class

☐ Store relative path names (instead of absolute paths)

13. Click on **Next** to display the next input dialog box for **Add Script**.
14. In this dialog box, you will specify the script that will be attached to the tool. Navigate to `c:\ArcpyBook\Ch7\InsertWildfires.py` and add `InsertWildfires.py` as the script.

15. You will also want to make sure that the **Run Python script in process** checkbox is selected, as shown in the following screenshot. Running a Python script "in process" increases the performance of your script.



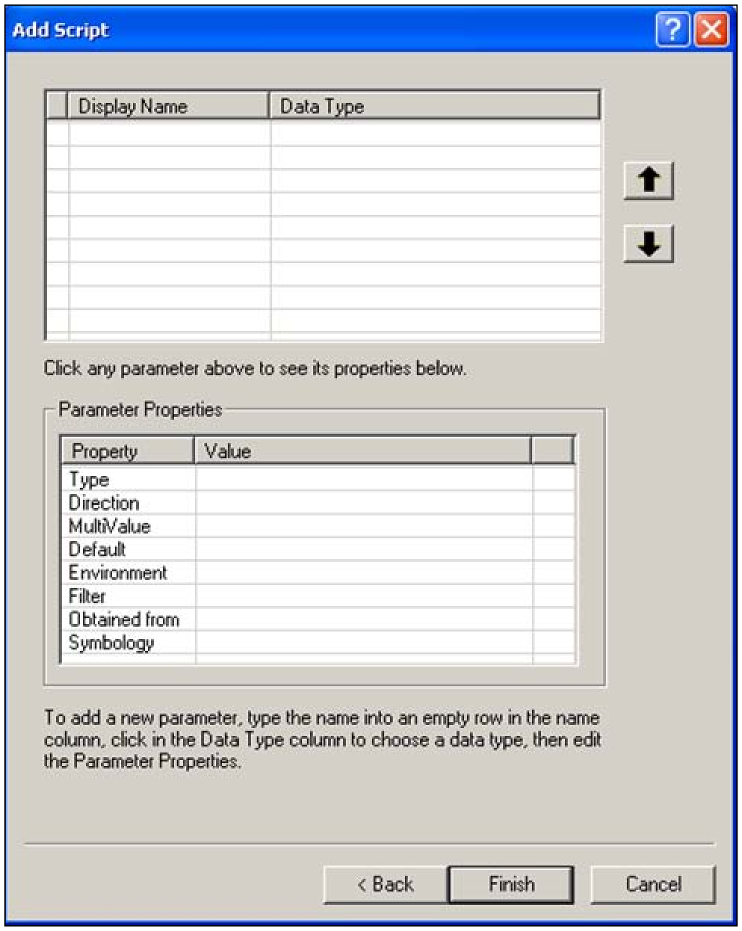
ramming 101 Updated\Lecture\Exercise11\InsertWildfires.py

[ ] Show command window when executing script

[✓] Run Python script in process

Running a script out of process requires ArcGIS to create a separate process to execute the script. The time it takes to start this process and execute the script leads to performance problems. Always run your scripts in process. Running a script in process means that ArcGIS does not have to spawn a second process to run the script. It runs in the same process space as ArcGIS.

16. Click on **Next** to display the parameter window, as shown in the following screenshot:

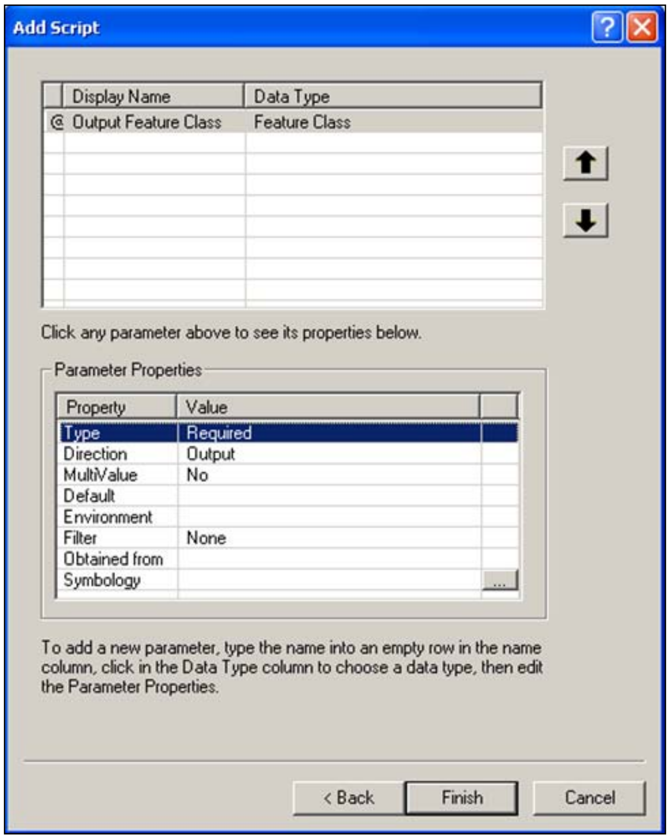

Add Script
Display Name
Data Type
Click any parameter above to see its properties below.
Parameter Properties
Property
Value
Type
Direction
MultiValue
Default
Environment
Filter
Obtained from
Symbology
To add a new parameter, type the name into an empty row in the name column, click in the Data Type column to choose a data type, then edit the Parameter Properties.
< Back
Finish
Cancel

Each parameter that you enter in this dialog box corresponds to a single call to `GetParameterAsText()`. Earlier, you altered your script to accept dynamic parameters through the `GetParameterAsText()` method. The parameters should be entered in this dialog box in the same order that your script expects to receive them. For instance, you inserted the following line of code in your code:

```
outputFC = arcpy.GetParameterAsText(0)
```

The first parameter that you add to the dialog box will need to correspond to this line. In our code, this parameter represents the feature class that will be created as a result of running this script. You add parameters by clicking on the first available row under **Display Name**. You can enter in any text in this row. This text will be displayed to the user. You will also need to select a corresponding datatype for the parameter. In this case, Data Type should be set to **Feature Class**, since this is the expected data that will be gathered from the user. Each parameter also has a number of properties that can be set. Some of the more important properties include **Type**, **Direction**, and **Default**.

17. Enter the information, as shown in the following screenshot, into your dialog box, for the output feature class. Make sure that you set **Direction** to **Output**:



Display Name	Data Type
Output Feature Class	Feature Class

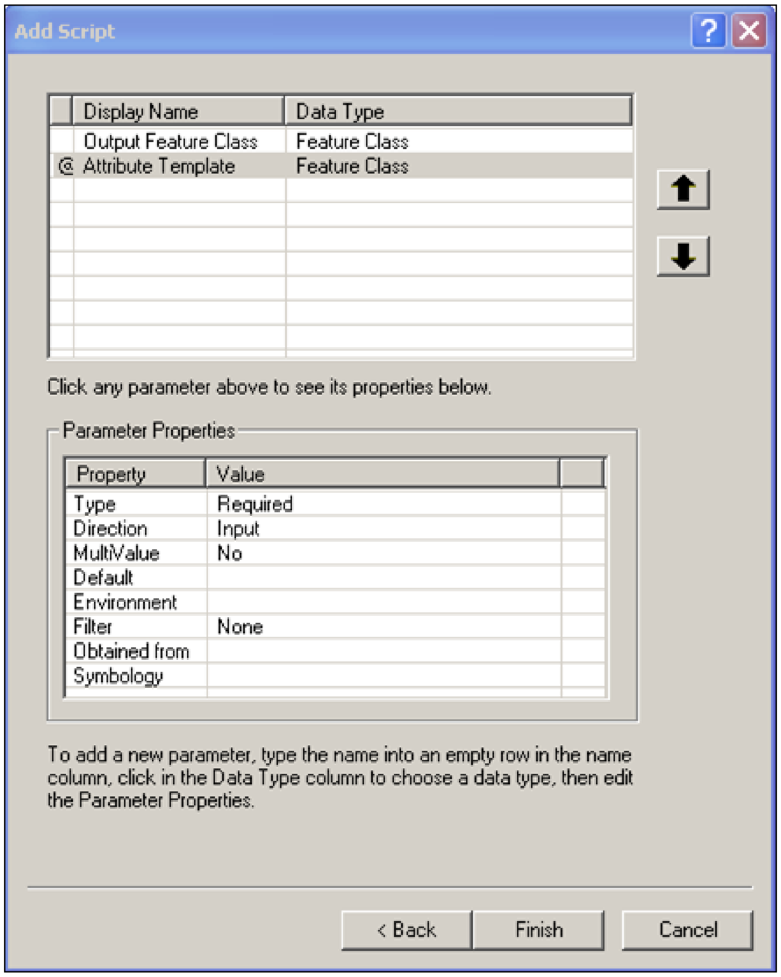
Click any parameter above to see its properties below.

Parameter Properties

Property	Value
Type	Required
Direction	Output
MultiValue	No
Default	
Environment	
Filter	None
Obtained from	
Symbology	

To add a new parameter, type the name into an empty row in the name column, click in the Data Type column to choose a data type, then edit the Parameter Properties.

18. Next, we need to add a parameter that defines the feature class that will be used as the attribute template for our new feature class. Enter the following information in your dialog box:



	Display Name	Data Type
	Output Feature Class	Feature Class
@	Attribute Template	Feature Class

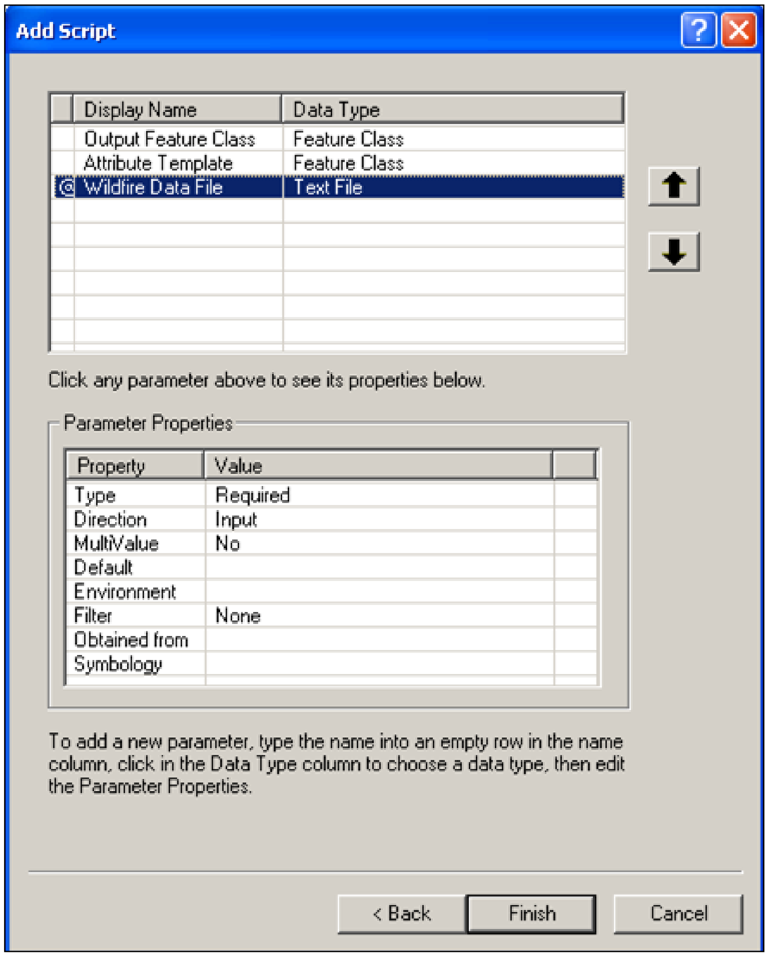
Click any parameter above to see its properties below.

Parameter Properties

Property	Value
Type	Required
Direction	Input
MultiValue	No
Default	
Environment	
Filter	None
Obtained from	
Symbology	

To add a new parameter, type the name into an empty row in the name column, click in the Data Type column to choose a data type, then edit the Parameter Properties.

19. Finally, we need to add a parameter that will be used to specify the comma-delimited text file that will be used as an input in the creation of our new feature class. Enter the following information into your dialog box:



Display Name	Data Type
Output Feature Class	Feature Class
Attribute Template	Feature Class
Wildfire Data File	Text File

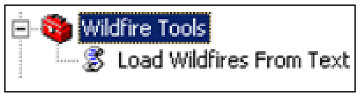
Click any parameter above to see its properties below.

Parameter Properties

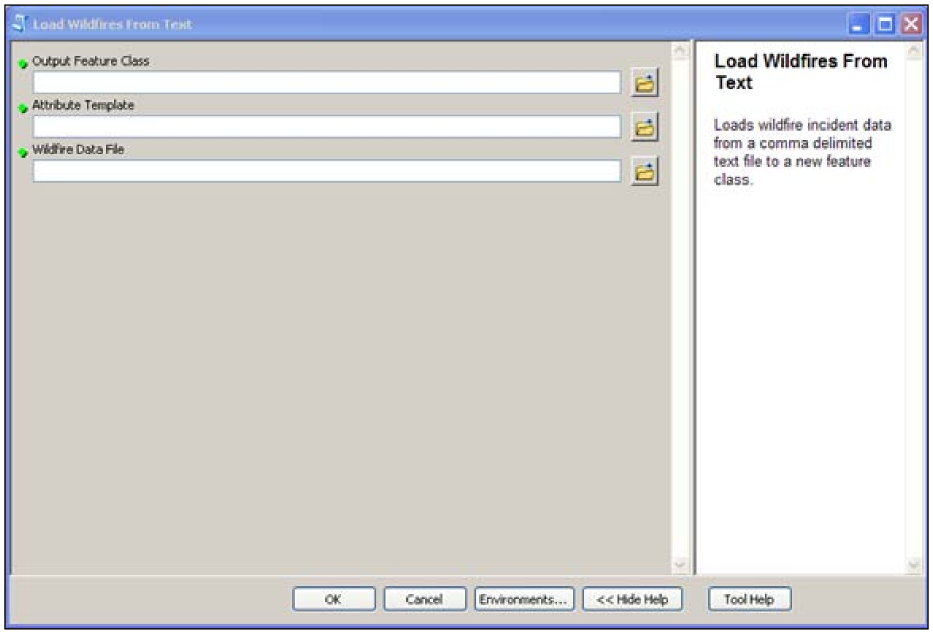
Property	Value
Type	Required
Direction	Input
MultiValue	No
Default	
Environment	
Filter	None
Obtained from	
Symbology	

To add a new parameter, type the name into an empty row in the name column, click in the Data Type column to choose a data type, then edit the Parameter Properties.

20. Click on **Finish**. The new script tool will be added to your **Wildfire Tools** toolbox, as shown in the following screenshot:



21. Now, we'll test the tool to make sure it works. Double-click on the script tool to display the dialog box, as shown in the following screenshot:

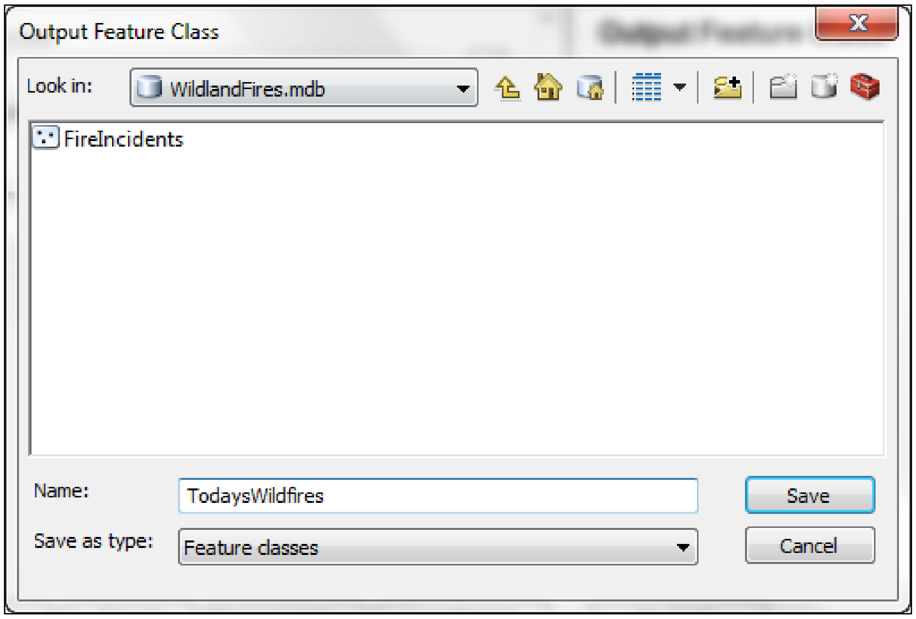


Load Wildfires From Text

Loads wildfire incident data from a comma delimited text file to a new feature class.

22. Define a new output feature class, which should be loaded inside the existing `WildlandFires.mdb` personal geodatabase, as shown in the next screenshot. Click on the open folder icon and navigate to the `WildlandFires.mdb` personal geodatabase, which should be located in `c:\ArcpyBook\data\Wildfires`.

23. You will also need to give your new feature class a name. In this case, we'll name the feature class `TodaysWildfires`, but the name can be whatever you'd like. In the following screenshot, you can see an example of how this should be done. Click on the **Save** button:



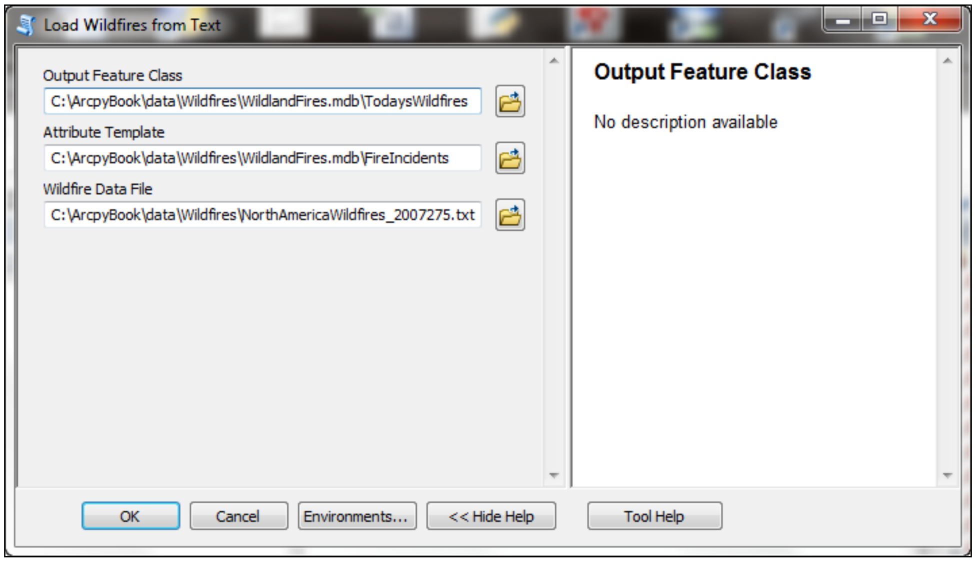
Name: TodaysWildfires

Save as type: Feature classes

24. For the attribute template, you will want to point to the `FireIncidents` feature class that has already been created for you. This feature class contains a field called `CONFIDENCEVAL`. This field will be created in our new feature class. Click on the **Browse** button, navigate to `c:\ArcpyBook\data\Wildfires\WildlandFires.mdb`, and you should see the `FireIncidents` feature class. Select it and click on **Add**.

25. Finally, the last parameter needs to point to our comma-delimited text file containing wildland fires. This file can be found at: `c:\ArcpyBook\data\Wildfires\NorthAmericaWildfires_2007275.txt`. Click on the **Browse** button and navigate to `c:\ArcpyBook\data\Wildfires`. Click on `NorthAmericaWildfires_2007275.txt` and click on the **Add** button.

Your tool should appear as follows:

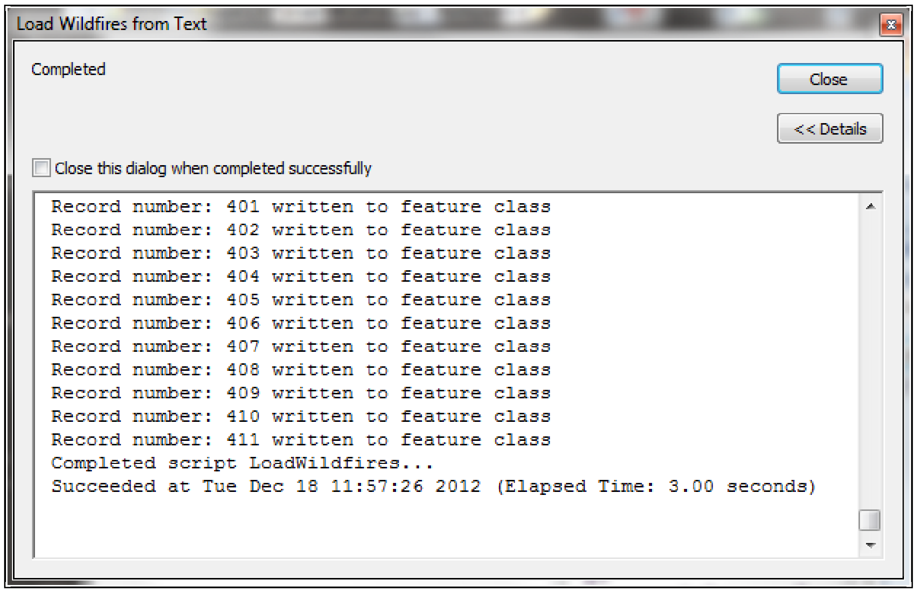


Output Feature Class
C:\ArcpyBook\data\Wildfires\WildlandFires.mdb\TodaysWildfires
Attribute Template
C:\ArcpyBook\data\Wildfires\WildlandFires.mdb\FireIncidents
Wildfire Data File
C:\ArcpyBook\data\Wildfires\NorthAmericaWildfires_2007275.txt

Output Feature Class

No description available

26. Click on **OK** to execute the tool. Any messages will be written to the dialog box as shown in the following screenshot. This is a standard dialog box for any geoprocessing tool. If everything has been set up correctly, you should see the following screenshot, which shows that a new feature class will be added to the ArcMap display:

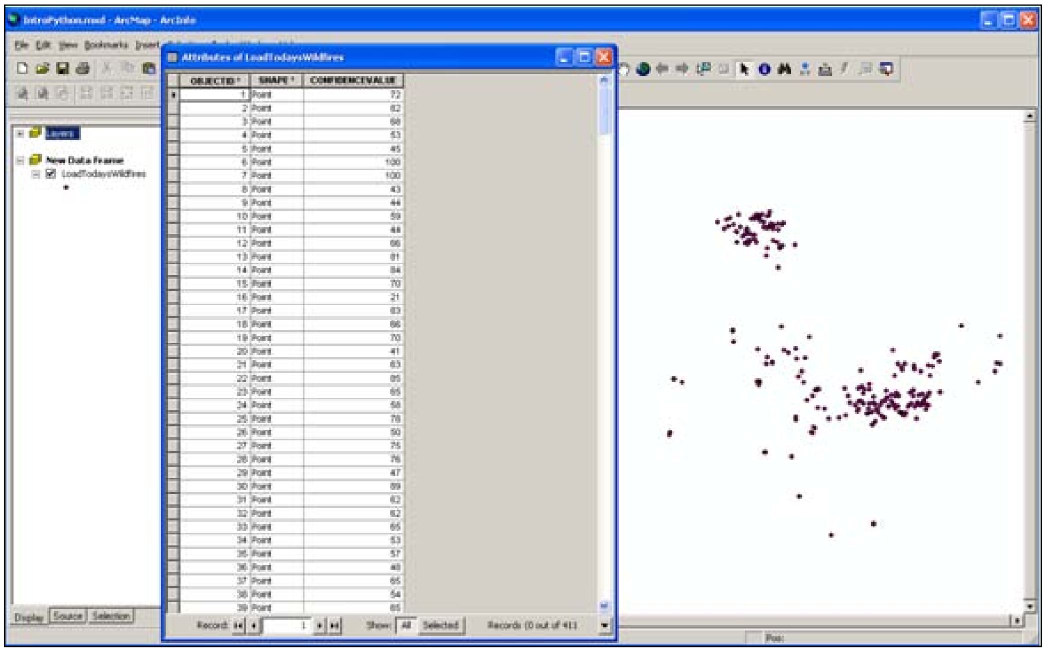


Completed

☐ Close this dialog when completed successfully

```
Record number: 401 written to feature class
Record number: 402 written to feature class
Record number: 403 written to feature class
Record number: 404 written to feature class
Record number: 405 written to feature class
Record number: 406 written to feature class
Record number: 407 written to feature class
Record number: 408 written to feature class
Record number: 409 written to feature class
Record number: 410 written to feature class
Record number: 411 written to feature class
Completed script LoadWildfires...
Succeeded at Tue Dec 18 11:57:26 2012 (Elapsed Time: 3.00 seconds)
```

If everything is set up correctly, you should see the following screenshot, which shows that a new feature class will be added to the ArcMap display:



OBJECTID *	SHAPE *	CONFIDENCEVALUE
1	Point	72
2	Point	62
3	Point	68
4	Point	53
5	Point	45
6	Point	100
7	Point	100
8	Point	43
9	Point	44
10	Point	59
11	Point	44
12	Point	66
13	Point	81
14	Point	84
15	Point	70
16	Point	21
17	Point	63
18	Point	66
19	Point	70
20	Point	41
21	Point	63
22	Point	85
23	Point	65
24	Point	58
25	Point	78
26	Point	50
27	Point	75
28	Point	76
29	Point	47
30	Point	89
31	Point	62
32	Point	62
33	Point	65
34	Point	53
35	Point	57
36	Point	48
37	Point	65
38	Point	54
39	Point	65

How it works...

Almost all script tools have parameters, and the values are set on the tool dialog box. When the tool is executed, the parameter values are sent to your script. Your script reads these values and then proceeds with its work. Python scripts can accept parameters as input. Parameters, also known as arguments, allow your scripts to become dynamic. Up to this point, all of our scripts have used hard coded values. By specifying input parameters for a script, you are able to supply the name of the feature class at run time. This capability makes your scripts more versatile.

The `GetParameterAsText()` method, which is used to capture parameter input, is zero-based with the first parameter entered occupying index `0`. Each successive parameter is incremented by 1. The output feature class that will be created by reading the comma-delimited text file is specified in the variable `outputFC`, which is retrieved by `GetParameterAsText(0)`. With `GetParameterAsText(1)`, we capture a feature class that will act as a template for the output feature class attribute schema. The attribute fields in the template feature class are used to define the fields that will populate our output feature class. Finally, `GetParameterAsText(2)` is used to create a variable called `f`, which will hold the comma-delimited text file that will be read.

There's more...

The `arcpy.GetParameterAsText()` method is not the only way to capture information passed into your script. When you call a Python script from the command line, you can pass in a set of arguments. When passing arguments to a script, each word must be separated by a space. These words are stored in a zero-based list object called `sys.argv`. With `sys.argv`, the first item in the list, referenced by index `0`, stores the name of the script. Each successive word is referenced by the next integer. Therefore, the first parameter will be stored in `sys.argv[1]`, the second in `sys.argv[2]`, and so on. These arguments can then be accessed from within your script.

It is recommended that you use the `GetParameterAsText()` function rather than `sys.argv`, because `GetParameterAsText()` does not have a character limit whereas `sys.argv` has a limit of 1,024 characters per parameter. In either case, once parameters have been read into the script, your script can continue execution using the input values.

8
Querying and Selecting Data

In this chapter, we will cover the following recipes:

- Constructing proper attribute query syntax
- Creating feature layers and table views
- Selecting features and rows with the Select Layer by Attribute tool
- Selecting features with the Select by Location tool
- Combining spatial and attribute queries with the Select by Location tool

Introduction

Selecting features from a geographic layer or rows from a standalone attribute table is one of the most common GIS operations. Queries are created to enable these selections, and can be either attribute or spatial queries. **Attribute queries** use SQL statements to select features or rows through the use of one or more fields or columns in a dataset. An example attribute query would be "Select all land parcels with a property value greater than $500,000". **Spatial queries** are used to select features based on some type of spatial relationship. An example might be "Select all land parcels that intersect a 100 year floodplain" or perhaps "Select all streets that are completely within Travis County, Texas". It is also possible to combine attribute and spatial queries. An example might be "Select all land parcels that intersect the 100 year floodplain and have a property value greater than $500,000".

Constructing proper attribute query syntax

The construction of property attribute queries is critical to your success in creating geoprocessing scripts that query data from feature classes and tables. All attribute queries that you execute against feature classes and tables will need to have the correct SQL syntax and also follow various rules depending upon the datatype that you execute the queries against.

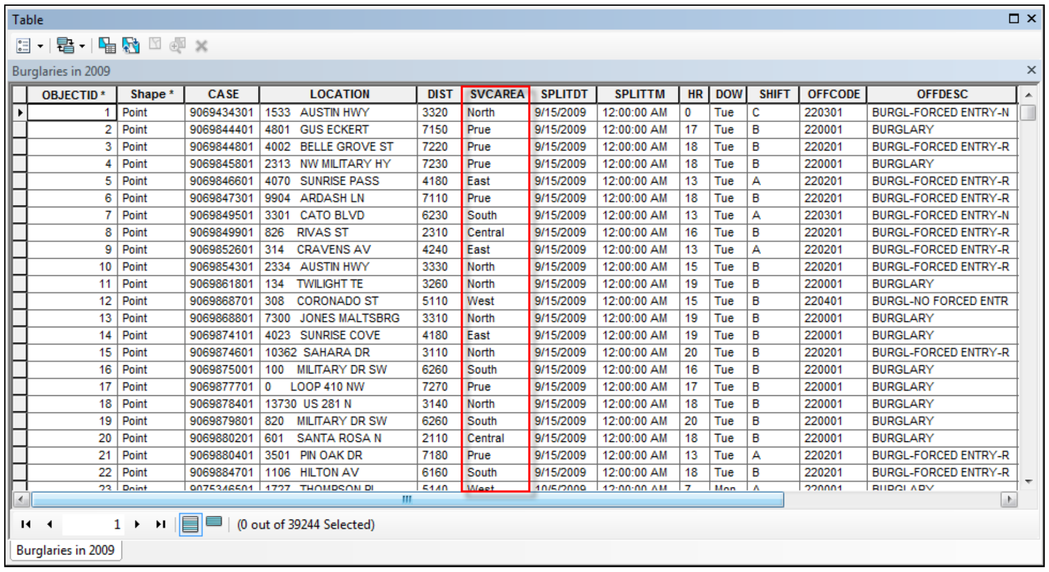
Getting ready

Creating the syntax for attribute queries is one of the most difficult and time-consuming tasks that you'll need to master when creating Python scripts that incorporate the use of the **Select by Attributes** tool. These queries are basically SQL statements along with a few idiosyncrasies that you'll need to master. If you already have a good understanding of creating queries in ArcMap or perhaps an experience with creating SQL statements in other programming languages, then this will be a little easier for you. In addition to creating valid SQL statements, you also need to be aware of some specific Python syntax requirements and some datatype differences that will result in a slightly altered formatting of your statements for some datatypes. In this recipe, you'll learn how to construct valid query syntax and understand the nuances of how different datatypes alter the syntax as well as some Python-specific constructs.

How to do it...

Initially, we're going to take a look at how queries are constructed in ArcMap, so that you can get a feel of how they are structured.

1. In ArcMap, open `C:\ArcpyBook\Ch8\Crime_Ch8.mxd`.
2. Right-click on the **Burglaries in 2009** layer and select **Open Attribute Table**. You should see an attribute table similar to the following screenshot. We're going to be querying the **SVCAREA** field:



Table

Burglaries in 2009

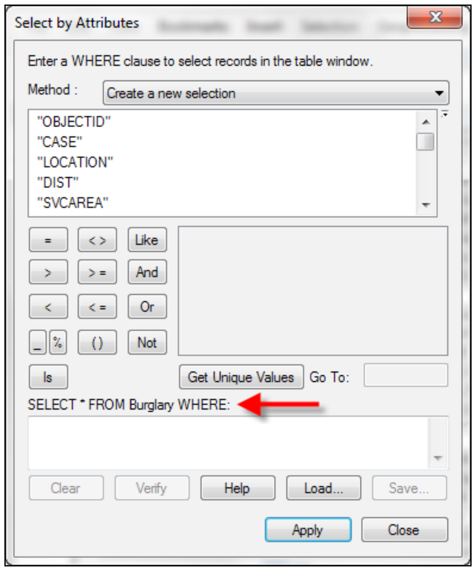
OBJECTID *	Shape *	CASE	LOCATION	DIST	SVCAREA	SPLITDT	SPLITTM	HR	DOW	SHIFT	OFFCODE	OFFDESC
1	Point	9069434301	1533 AUSTIN HWY	3320	North	9/15/2009	12:00:00 AM	0	Tue	C	220301	BURGL-FORCED ENTRY-N
2	Point	9069844401	4801 GUS ECKERT	7150	Prue	9/15/2009	12:00:00 AM	17	Tue	B	220001	BURGLARY
3	Point	9069844801	4002 BELLE GROVE ST	7220	Prue	9/15/2009	12:00:00 AM	18	Tue	B	220201	BURGL-FORCED ENTRY-R
4	Point	9069845801	2313 NW MILITARY HY	7230	Prue	9/15/2009	12:00:00 AM	18	Tue	B	220001	BURGLARY
5	Point	9069846601	4070 SUNRISE PASS	4180	East	9/15/2009	12:00:00 AM	13	Tue	A	220201	BURGL-FORCED ENTRY-R
6	Point	9069847301	9904 ARDASH LN	7110	Prue	9/15/2009	12:00:00 AM	18	Tue	B	220201	BURGL-FORCED ENTRY-R
7	Point	9069849501	3301 CATO BLVD	6230	South	9/15/2009	12:00:00 AM	13	Tue	A	220301	BURGL-FORCED ENTRY-N
8	Point	9069849901	826 RIVAS ST	2310	Central	9/15/2009	12:00:00 AM	16	Tue	B	220201	BURGL-FORCED ENTRY-R
9	Point	9069852601	314 CRAVENS AV	4240	East	9/15/2009	12:00:00 AM	13	Tue	A	220201	BURGL-FORCED ENTRY-R
10	Point	9069854301	2334 AUSTIN HWY	3330	North	9/15/2009	12:00:00 AM	15	Tue	B	220201	BURGL-FORCED ENTRY-R
11	Point	9069861801	134 TWILIGHT TE	3260	North	9/15/2009	12:00:00 AM	19	Tue	B	220001	BURGLARY
12	Point	9069868701	308 CORONADO ST	5110	West	9/15/2009	12:00:00 AM	15	Tue	B	220401	BURGL-NO FORCED ENTR
13	Point	9069868801	7300 JONES MALTSBRG	3310	North	9/15/2009	12:00:00 AM	19	Tue	B	220001	BURGLARY
14	Point	9069874101	4023 SUNRISE COVE	4180	East	9/15/2009	12:00:00 AM	19	Tue	B	220001	BURGLARY
15	Point	9069874601	10362 SAHARA DR	3110	North	9/15/2009	12:00:00 AM	20	Tue	B	220201	BURGL-FORCED ENTRY-R
16	Point	9069875001	100 MILITARY DR SW	6260	South	9/15/2009	12:00:00 AM	16	Tue	B	220001	BURGLARY
17	Point	9069877701	0 LOOP 410 NW	7270	Prue	9/15/2009	12:00:00 AM	17	Tue	B	220001	BURGLARY
18	Point	9069878401	13730 US 281 N	3140	North	9/15/2009	12:00:00 AM	18	Tue	B	220001	BURGLARY
19	Point	9069879801	820 MILITARY DR SW	6260	South	9/15/2009	12:00:00 AM	20	Tue	B	220001	BURGLARY
20	Point	9069880201	601 SANTA ROSA N	2110	Central	9/15/2009	12:00:00 AM	18	Tue	B	220001	BURGLARY
21	Point	9069880401	3501 PIN OAK DR	7180	Prue	9/15/2009	12:00:00 AM	13	Tue	A	220201	BURGL-FORCED ENTRY-R
22	Point	9069884701	1106 HILTON AV	6160	South	9/15/2009	12:00:00 AM	18	Tue	B	220201	BURGL-FORCED ENTRY-R
23	Point	9075346501	1727 THOMPSON PL	5140	West	10/5/2009	12:00:00 AM	7	Mon	A	220001	BURGLARY

1 (0 out of 39244 Selected)

Burglaries in 2009

3. With the attribute table open, select the **Table Options** button and then **Select by Attributes** to display a dialog box that will allow you to construct an attribute query.

   Notice the **Select * FROM Burglary WHERE:** statement on the query dialog box (shown in the following screenshot). This is a basic SQL statement that will return all the columns from the attribute table for **Burglary** that meet the condition that we define through the query builder. The asterisk (*****) simply indicates that all fields will be returned:



Select by Attributes

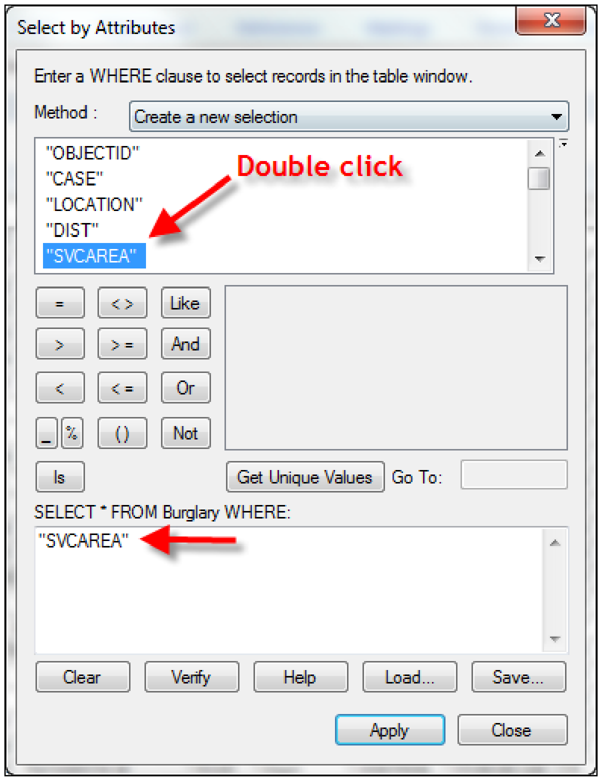
Enter a WHERE clause to select records in the table window.

Method : Create a new selection

"OBJECTID"
"CASE"
"LOCATION"
"DIST"
"SVCAREA"

SELECT * FROM Burglary WHERE:

4. Make sure that **Create a new selection** is the selected item in the **Method** dropdown list. This will create a new selection set.
5. Double-click on **SVCAREA** from the list of fields to add the field to the SQL statement builder, as follows:



Select by Attributes

Enter a WHERE clause to select records in the table window.

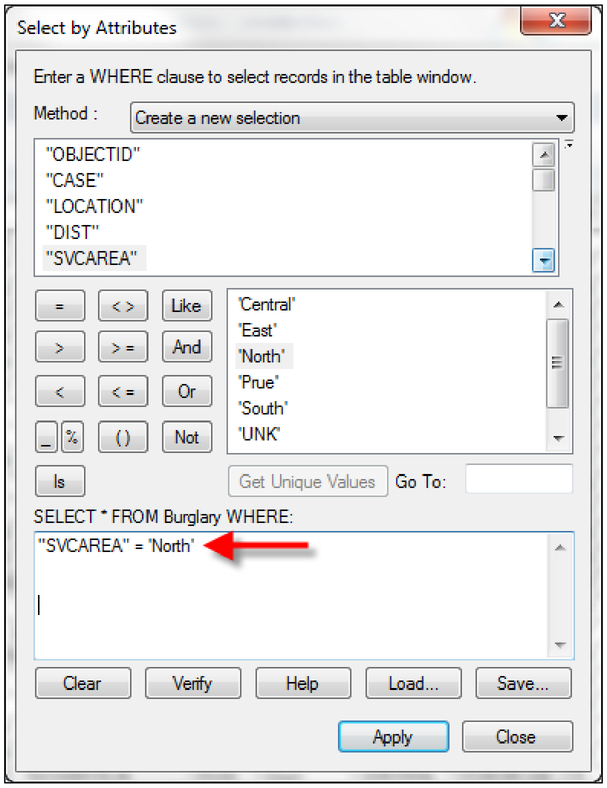
Method : Create a new selection

"OBJECTID"
"CASE"
"LOCATION"
"DIST"
"SVCAREA"

SELECT * FROM Burglary WHERE:

"SVCAREA"

6. Click on the **=** button.
7. Click on the **Get Unique Values** button.
8. From the list of values generated, double-click on **'North'** to complete the SQL statement, as shown in the following screenshot:



```
"SVCAREA" = 'North'
```

9. Click on the **Apply** button to execute the query. This should select 7520 records.

   Many people mistakenly assume that you can simply take a query that has been generated in this fashion and paste it into a Python script. That is not the case. There are some important differences that we'll cover next.

10. Close the **Select by Attributes** window and the **Burglaries in 2009** table.
11. Clear the selected feature set by clicking on **Selection** | **Clear Selected Features**.
12. Open the Python window and add the code to import `arcpy`.

```
import arcpy
```

13. Create a new variable to hold the query and add the exact same statement that you created earlier:

```
qry = "SVCAREA" = 'North'
```

14. Press *Enter* on your keyboard and you should see an error message similar to the following:

    ```
    Runtime error SyntaxError: can't assign to literal (<string>, line
    1)
    ```

Python interprets **SVCAREA** and **North** as strings but the equal to sign between the two is not part of the string used to set the `qry` variable. There are several things we need to do to generate a syntactically correct statement for the Python interpreter.

One important thing has already been taken care of though. Each field name used in a query needs to be surrounded by double quotes. In this case, **SVCAREA** is the only field used in the query and it has already been enclosed by double quotes. This will always be the case when you're working with shapefiles, file geodatabases, or ArcSDE geodatabases. Here is where it gets a little confusing though. If you're working with data from a personal geodatabase, the field names will need to be enclosed by square brackets instead of double quotes as shown in the following code example. This can certainly leads to confusion for script developers.

```
qry = [SVCAREA] = 'North'
```

Now, we need to deal with the single quotes surrounding **'North'**. When querying data from fields that have a `text` datatype, the string being evaluated must be enclosed by quotes. If you examine the original query, you'll notice that we have in fact already enclosed the word `North` with quotes, so everything should be fine right? Unfortunately, it's not that simple with Python. Quotes, along with a number of other characters, must be escaped with a forward slash followed by the character being escaped. In this case, the escape sequence would be `\'`.

1. Alter your query syntax to incorporate the escape sequence:

   ```
   qry = "SVCAREA" = \'North\'
   ```

2. Finally, the entire query statement should be enclosed with quotes:

   ```
   qry = '"SVCAREA" = \'North\''
   ```

In addition to the `=` sign, which tests for equality, there are a number of additional operators that you can use with strings and numeric data, including not equal (`< >`), greater than (`>`), greater than or equal to (`>=`), less than (`<`), and less than or equal to (`<=`).

Wildcard characters including `%` and `_` can also be used for shapefiles, file geodatabases, and ArcSDE geodatabases. These include `%` for representing any number of characters. The `LIKE` operator is often used with wildcard characters to perform partial string matching. For example, the following query would find all records with a service area that begins with `N` and has any number of characters after.

```
qry = '"SVCAREA" LIKE \'N%\''
```

The underscore character (_) can be used to represent a single character. For personal geodatabases the asterisk (`*`) is used to represent a wildcard character for any number of characters, while (`?`) represents a single character.

You can also query for the absence of data, also known as `NULL` values. A `NULL` value is often mistaken for a value of zero, but that is not the case. `NULL` values indicate the absence of data, which is different from a value of zero. Null operators include **IS NULL** and **IS NOT NULL**. The following code example will find all records where the `SVCAREA` field contains no data:

```
qry = '"SVCAREA" IS NULL'
```

The final topic that we'll cover in this section are operators used for combining expressions where multiple query conditions need to be met. The `AND` operator requires that both query conditions be met for the query result to be true, resulting in selected records. The `OR` operator requires that at least one of the conditions be met.

How it works...

The creation of syntactically correct queries is one of the most challenging aspects of programming ArcGIS with Python. However, once you understand some basic rules, it gets a little easier. In this section, we'll summarize these rules. One of the more important things to keep in mind is that field names must be enclosed with double quotes for all datasets, with the exception of personal geodatabases, which require braces surrounding field names.

There is also an `AddFieldDelimiters()` function that you can use to add the correct delimiter to a field based on the datasource supplied as a parameter to the function. The syntax for this function is as follows:

```
AddFieldDelimiters(dataSource,field)
```

Additionally, most people, especially those new to programming with Python, struggle with the issue of adding single quotes to string values being evaluated by the query. In Python, quotes have to be escaped with a single forward slash followed by the quote. Using this escape sequence will ensure that Python does in fact see that as a quote rather than the end of the string.

Finally, take some time to familiarize yourself with the wildcard characters. For datasets other than personal geodatabases, you'll use the (`%`) character for multiple characters and an underscore (_) character for a single character. If you're using a personal geodatabase, the (`*`) character is used to match multiple characters and the (`?`) character is used to match a single character. Obviously, the syntax differences between personal geodatabases and all other types of datasets can lead to some confusion.

Creating feature layers and table views

Feature layers and table views serve as intermediate datasets held in memory for use specifically with tools such as Select by Location and Select Attributes. Although these temporary datasets can be saved, they are not needed in most cases.

Getting ready

Feature classes are physical representations of geographic data and are stored as files (shapefiles, personal geodatabases, and file geodatabases) or within a geodatabase. ESRI defines a feature class as "a collection of features that shares a common geometry (point, line, or polygon), attribute table, and spatial reference."

Feature classes can contain default and user-defined fields. Default fields include the `SHAPE` and `OBJECTID` fields. These fields are maintained and updated automatically by ArcGIS. The `SHAPE` field holds the geometric representation of a geographic feature, while the `OBJECTID` field holds a unique identifier for each feature. Additional default fields will also exist depending on the type of feature class. A line feature class will have a `SHAPE_LENGTH` field. A polygon feature class will have both, a `SHAPE_LENGTH` and a `SHAPE_AREA` field.

Optional fields are created by end users of ArcGIS and are not automatically updated by GIS. These contain attribute information about the features. These fields can also be updated by your scripts.

Tables are physically represented as standalone DBF tables or within a geodatabase. Both, tables and feature classes, contain attribute information. However, a table contains only attribute information. There isn't a `SHAPE` field associated with a table, and they may or may not contain an `OBJECTID` field.

Standalone Python scripts that use the **Select by Attributes** or **Select by Location** tool require that you create an intermediate dataset rather than using feature classes or tables. These intermediate datasets are temporary in nature and are called **Feature Layers** or **Table Views**. Unlike feature classes and tables, these temporary datasets do not represent actual files on disk or within a geodatabase. Instead, they are "in memory" representations of feature classes and tables. These datasets are active only while a Python script is running. They are removed from memory after the tool has executed. However, if the script is run from within ArcGIS as a script tool, then the temporary layer can be saved either by right-clicking on the layer in the table of contents and selecting **Save As Layer File** or simply by saving the map document file.

Feature layers and table views must be created as a separate step in your Python scripts, before you can call the **Select by Attributes** or **Select by Location** tools. The **Make Feature Layer** tool generates the "in-memory" representation of a feature class, which can then be used to create queries and selection sets, as well as to join tables. After this step has been completed, you can use the **Select by Attributes** or **Select by Location** tool. Similarly, the **Make Table View** tool is used to create an "in-memory" representation of a table. The function of this tool is the same as **Make Feature Layer**. Both the **Make Feature Layer** and **Make Table View** tools require an input dataset, an output layer name, and an optional query expression, which can be used to limit the features or rows that are a part of the output layer. In addition, both tools can be found in the **Data Management** Tools toolbox.

The syntax for using the **Make Feature Layer** tool is as follows:

```
arcpy.MakeFeatureLayer_management(<input feature layer>, <output layer
name>,{where clause})
```

The syntax for using the **Make Table View** tool is as follows:

```
Arcpy.MakeTableView_management(<input table>, <output table name>,
{where clause})
```

In this recipe, you will learn how to use the **Make Feature Layer** and **Make Table View** tools. These tasks will be done inside ArcGIS, so that you can see the in-memory copy of the layer that is created.

How to do it...

Follow these steps to learn how to use the **Make Feature Layer** and **Make Table View** tools:

1. Open `c:\ArcpyBook\Ch8\Crime_Ch8.mxd` in ArcMap.
2. Open the Python window.
3. Import the `arcpy` module:

   ```
   import arcpy
   ```

4. Set the workspace:

   ```
   arcpy.env.workspace = "c:/ArcpyBook/data/CityOfSanAntonio.gdb"
   ```

5. Start a `try` block:

   ```
   try:
   ```

6. Make an in-memory copy of the `Burglary` feature class using the **Make Feature Layer** tool. Make sure you indent this line of code:

   ```
   flayer = arcpy.MakeFeatureLayer_management("Burglary","Burglary_
   Layer")
   ```

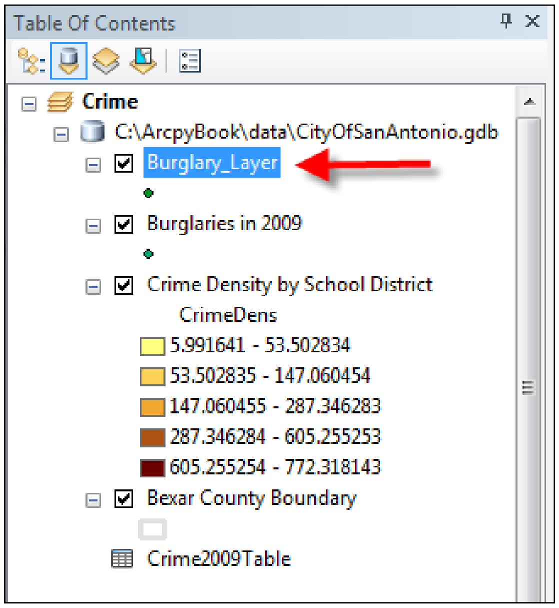
7. Add an `except` block and a line of code to print an error message in the event of a problem:

   ```
   except:
     print "An error occurred during creation"
   ```

8. The entire script should appear as follows:

   ```
   import arcpy
   arcpy.env.workspace = "c:/ArcpyBook/data/CityOfSanAntonio.gdb"
   try:
     flayer = arcpy.MakeFeatureLayer_management("Burglary","Burglary_
   Layer")
   except:
     print "An error occurred during creation"
   ```

9. Save the script to `c:\ArcpyBook\Ch8\CreateFeatureLayer.py`.
10. Run the script. The new `Burglary_Layer` file will be added to the ArcMap table of contents:



11. The **Make Table View** tool functionality is equivalent to the **Make Feature Layer** tool. The difference is that it works against standalone tables instead of feature classes.
12. Remove the following line of code:

    ```
    flayer = arcpy.MakeFeatureLayer_management("Burglary","Burglary_
    Layer")
    ```

13. Add the following line of code in its place:

    ```
    tView = arcpy.MakeTableView_management("Crime2009Table",
    "Crime2009TView")
    ```

14. Run the script to see the table view added to the ArcMap table of contents.

How it works...

The **Make Feature Layer** and **Make Table View** tools create in-memory representations of feature classes and tables respectively. Both the **Select by Attributes** and **Select by Location** tools require that these temporary, in-memory structures be passed in as parameters when called from a Python script. Both tools also require that you pass in a name for the temporary structures.

There's more...

You can also apply a query to either the **Make Feature Layer** or **Make Table View** tools to restrict the records returned in the feature layer or table view. This is done through the addition of a `where` clause when calling either of the tools from your script. This query is much the same as if you'd set a definition query on the layer through **Layer Properties** | **Definition Query**.

The syntax for adding a query is as follows:

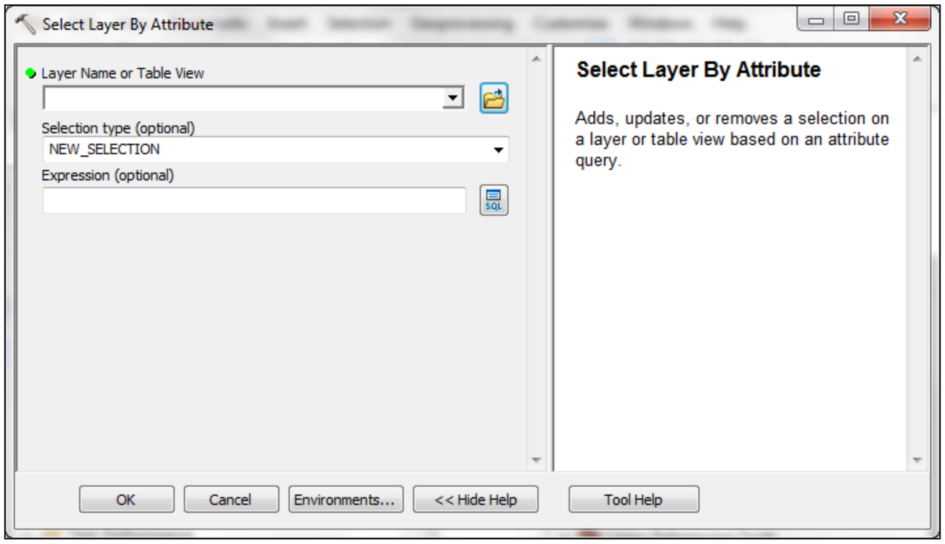
```
MakeFeatureLayer(in_features, out_layer, where_clause)
MakeTableView(in_table, out_view, where_clause)
```

Selecting features and rows with the Select Layer by Attribute tool

Attribute queries can be executed against a feature class or a table through the use of the **Select Layer by Attribute** tool. A `where` clause can be included to filter the selected records and various selection types can be included.

Getting ready

The **Select Layer by Attribute** tool shown in the following screenshot is used to select records from a feature class or table based on a query that you define. We covered the somewhat complex topic of queries in an earlier recipe in this chapter, so hopefully you now understand the basic concepts of creating a query. You have also learned how to create a temporary, in-memory representation of a feature class or table, which is a pre-requisite to using either the **Select by Attributes** or **Select by Location** tool.


Select Layer By Attribute

Layer Name or Table View

Selection type (optional)
NEW_SELECTION

Expression (optional)

Select Layer By Attribute

Adds, updates, or removes a selection on a layer or table view based on an attribute query.

The **Select by Attributes** tool uses a query along with either a feature layer or table view, and a selection method to select records. By default, the selection method will be a new selection set. Other selection methods include "add to selection", "remove from selection", "subset selection", "switch selection", and "clear selection". Each of the selection methods is summarized as follows:

- `NEW_SELECTION`: It is the default selection method and creates a new selection set
- `ADD_TO_SELECTION`: It adds a selection set to the currently selected records, based on a query

- `REMOVE_FROM_SELECTION`: It removes records from a selection set based on a query
- `SUBSET_SELECTION`: It combines selected records that are common to the existing selection set
- `SWITCH_SELECTION`: It selects records that are not selected currently and unselects the existing selection set
- `CLEAR_SELECTION`: It clears all records that are currently a part of the selected set

The syntax for calling the **Select by Attributes** tool is as follows:

```
arcpy.SelectLayerByAttribute_management(<input feature layer or
table view>, {selection method}, {where clause})
```

In this recipe, you'll learn how to use the **Select by Attributes** tool to select features from a feature class. You'll use the skills you learned in previous recipes to build a query, create a feature layer, and finally call the **Select by Attributes** tool.

How to do it…

Follow these steps to learn how to select records from a table or feature class using the **Select Layer by Attributes** tool:

1. Open IDLE and create a new script window.
2. Save the script to `c:\ArcpyBook\Ch8\SelectLayerAttribute.py`.
3. Import the `arcpy` module:
   ```
   import arcpy
   ```
4. Set the workspace to the City of San Antonio geodatabase.
   ```
   arcpy.env.workspace = "c:/ArcpyBook/data/CityOfSanAntonio.gdb"
   ```
5. Start a `try` block:
   ```
   try:
   ```
6. Create the query that we used in the first recipe in this chapter. This will serve as a `where` clause that will select all the records with a service area of `North`. This line of code and the next four should be indented:
   ```
   qry = '"SVCAREA" = \'North\''
   ```
7. Make an in-memory copy of the `Burglary` feature class:
   ```
   flayer = arcpy.MakeFeatureLayer_management("Burglary","Burglary_
   Layer")
   ```

8. Call the **Select Layer by Attribute** tool passing in a reference to the feature layer we just created. Define this as a new selection set and pass in a reference to the query:

   ```
   arcpy.SelectLayerByAttribute_management(flayer, "NEW_SELECTION",
   qry)
   ```

9. Print the number of selected records in the layer using the Get Count tool:

   ```
   cnt = arcpy.GetCount_management(flayer)
   print "The number of selected records is: " + str(cnt)
   ```

10. Add an `except` block and a line of code to print an error message in the event of a problem:

    ```
    except:
      print "An error occurred during selection"
    ```

11. The entire script should appear as shown in the following code snippet. Please remember to include indentation with the `try` and `except` blocks:

    ```
    import arcpy
    arcpy.env.workspace = "c:/ArcpyBook/data/CityOfSanAntonio.gdb"
    try:
      qry = '"SVCAREA" = \'North\''
      flayer =   arcpy.MakeFeatureLayer_
    management("Burglary","Burglary_Layer")
      arcpy.SelectLayerByAttribute_management(flayer, "NEW_SELECTION",
    qry)
      cnt = arcpy.GetCount_management(flayer)
      print "The number of selected records is: " + str(cnt)
    except:
      print "An error occurred during selection"
    ```

12. Save the script.
13. Run the script. If everything has been done correctly, you should see a message indicating that 7520 records have been selected:

    ```
    The total number of selected records is: 7520
    ```

How it works...

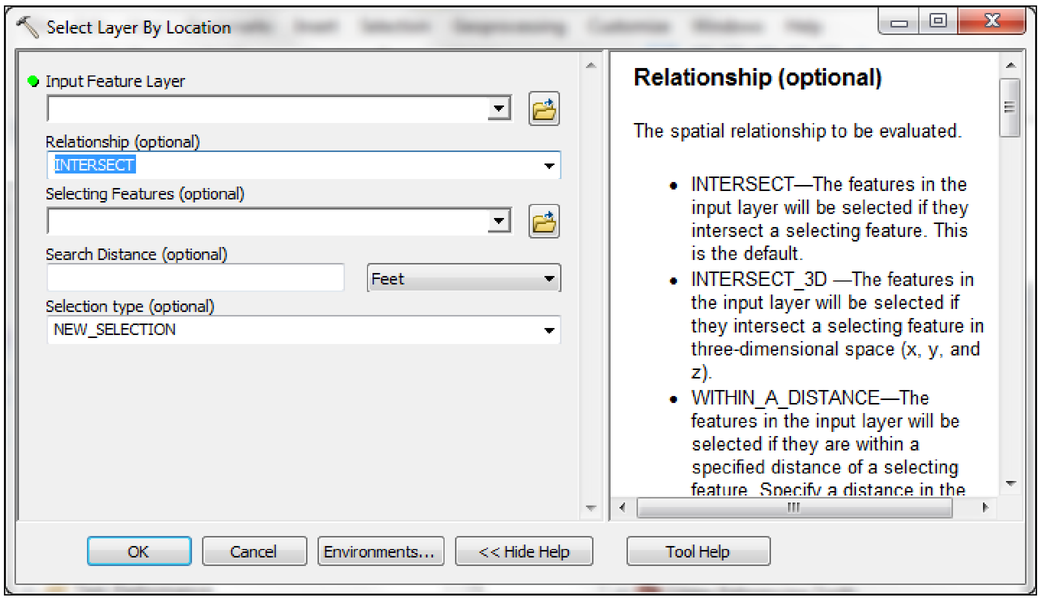
The **Select by Attributes** tool requires that either a feature layer or table view be passed in as the first parameter. In this recipe, we passed in a feature layer that was created by the Make Feature Layer tool in the line just above. We used **Make Feature Layer** to create a feature layer from the `Burglary` feature class. This feature layer was assigned to the variable `flayer`, which is then passed into the **Select by Attribute** tool as the first parameter. In this script, we also passed in a parameter indicating that we'd like to create a new selection set along with the `final` parameter, which is a `where` clause. The `where` clause is specified in the `qry` variable. This variable holds a query that will select all the features with a service area of `North`.

Selecting features with the Select by Location tool

The **Select Layer by Location** tool, as shown in the next screenshot, can be used to select features based on some type of spatial relationship. Since it deals with spatial relationships, this tool only applies to feature classes and their corresponding in-memory feature layers.

Getting ready

There are many different types of spatial relationships that you can apply while selecting features using the **Select by Location** tool, including intersect, contains, within, boundary touches, is identical, and many others. If not specified, the default intersect spatial relationship will be applied. The input feature layer is the only required parameter, but there are a number of optional parameters including the spatial relationship, search distance, a feature layer, or feature class to test against the input layer, and a selection type. In this recipe, you will learn how to use the **Select by Location** tool in a Python script to select features based on a spatial relationship. You'll use the tool to select burglaries that are within the boundaries of the Edgewood school district.


Select Layer By Location

Input Feature Layer

Relationship (optional)
INTERSECT

Selecting Features (optional)

Search Distance (optional)
Feet

Selection type (optional)
NEW_SELECTION

Relationship (optional)

The spatial relationship to be evaluated.

- INTERSECT—The features in the input layer will be selected if they intersect a selecting feature. This is the default.
- INTERSECT_3D —The features in the input layer will be selected if they intersect a selecting feature in three-dimensional space (x, y, and z).
- WITHIN_A_DISTANCE—The features in the input layer will be selected if they are within a specified distance of a selecting feature. Specify a distance in the

How to do it...

Follow these steps to learn how to perform a spatial query using the **Select by Location** tool:

1. Open IDLE and create a new script window.
2. Save the script to `c:\ArcpyBook\Ch8\SelectByLocation.py`.
3. Import the `arcpy` module:

   ```
   import arcpy
   ```

4. Set the workspace to the City of San Antonio geodatabase:

   ```
   arcpy.env.workspace = "c:/ArcpyBook/data/CityOfSanAntonio.gdb"
   ```

5. Start a `try` block:

   ```
   try:
   ```

6. Make an in-memory copy of the `Burglary` feature class:

   ```
   flayer = arcpy.MakeFeatureLayer_management("Burglary","Burglary_
   Layer")
   ```

7. Call the **Select Layer by Location** tool passing in a reference to the feature layer we just created. The spatial relationship test will be `COMPLETELY_WITHIN`, meaning that we want to find all burglaries that are completely within the boundaries of the comparison layer. Define `EdgewoodSD.shp` as the comparison layer:

   ```
   arcpy.SelectLayerByLocation_management (flayer, "COMPLETELY_
   WITHIN", "c:/ArcpyBook/Ch8/EdgewoodSD.shp")
   ```

8. Print the number of selected records in the layer using the **Get Count** tool:

   ```
   cnt = arcpy.GetCount_management(flayer)
   print "The number of selected records is: " + str(cnt)
   ```

9. Add an `except` block and a line of code to print an error message in the event of a problem:

   ```
   except:
     print "An error occurred during selection"
   ```

10. The entire script should appear as shown in the following code snippet. Please remember to include indentation with the `try` and `except` blocks:

    ```
    import arcpy
    arcpy.env.workspace = "c:/ArcpyBook/data/CityOfSanAntonio.gdb"
    try:
      flayer =   arcpy.MakeFeatureLayer_
    management("Burglary","Burglary_Layer")
    ```

```
  arcpy.SelectLayerByLocation_management (flayer, "COMPLETELY_
WITHIN", "c:/ArcpyBook/Ch8/EdgewoodSD.shp")
  cnt = arcpy.GetCount_management(flayer)
  print "The number of selected records is: " + str(cnt)
except:
  print "An error occurred during selection"
```

11. Save the script.
12. Run the script. If everything was done correctly, you should see a message indicating that 1470 records have been selected:

```
The total number of selected records is: 1470
```

In this case, we did not define the optional search distance and selection type parameters. By default, a new selection will be applied as the selection type. We didn't apply a distance parameter in this case, but we'll do that now to illustrate how it works.

1. Update the line of code that calls the **Select Layer** by Location tool:

```
arcpy.SelectLayerByLocation_management (flayer, "WITHIN_A_
DISTANCE", "c:/ArcpyBook/Ch8/EdgewoodSD.shp","1 MILES")
```

2. Save the script.
3. Run the script. If everything was done correctly, you should see a message indicating that 2976 records have been selected. This will select all burglaries within the boundaries of the Edgewood school district along with any burglaries within one mile of the boundary:

```
The total number of selected records is: 2976
```

The final thing you'll do in this section is use the **Copy Features** tool to write the temporary layer to a new feature class.

1. Comment out the two lines of code that get a count of the number of features, and print them to the screen:

```
## cnt = arcpy.GetCount_management(flayer)
## print "The number of selected records is: " + str(cnt)
```

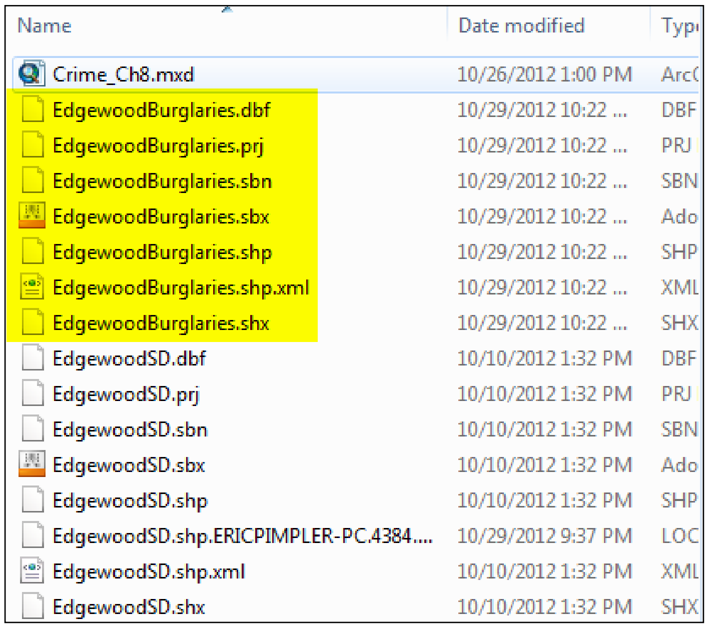
2. Add a line of code that calls the **Copy Features** tool. This line should be placed just below the line of code that calls the **Select Layer by Location** tool. The **Copy Features** tool accepts a feature layer as the first input parameter and an output feature class, which in this case will be a shapefile called `EdgewoodBurglaries.shp`:

```
arcpy.CopyFeatures_management(flayer, 'c:/ArcpyBook/Ch8/
EdgewoodBurglaries.shp')
```

3. The entire script should now appear as shown in the following code snippet. Please remember to include indentation with the `try` and `except` blocks:

```
import arcpy
arcpy.env.workspace = "c:/ArcpyBook/data/CityOfSanAntonio.gdb"
try:
  flayer = arcpy.MakeFeatureLayer_management("Burglary","Burglary_
Layer")
  arcpy.SelectLayerByLocation_management (flayer, "WITHIN_A_
DISTANCE", "c:/ArcpyBook/Ch8/EdgewoodSD.shp","1 MILES")
  arcpy.CopyFeatures_management(flayer, 'c:/ArcpyBook/Ch8/
EdgewoodBurglaries.shp')
        #cnt = arcpy.GetCount_management(flayer)
 #print "The total number of selected records is: " + str(cnt)
except:
  print "An error occurred during selection"
```

4. Save the script.
5. Run the script.
6. Examine your `c:\ArcpyBook\Ch8` folder to see the output shapefile:



Name	Date modified	Typ
Crime_Ch8.mxd	10/26/2012 1:00 PM	Arc(
EdgewoodBurglaries.dbf	10/29/2012 10:22 ...	DBF
EdgewoodBurglaries.prj	10/29/2012 10:22 ...	PRJ
EdgewoodBurglaries.sbn	10/29/2012 10:22 ...	SBN
EdgewoodBurglaries.sbx	10/29/2012 10:22 ...	Ado
EdgewoodBurglaries.shp	10/29/2012 10:22 ...	SHP
EdgewoodBurglaries.shp.xml	10/29/2012 10:22 ...	XML
EdgewoodBurglaries.shx	10/29/2012 10:22 ...	SHX
EdgewoodSD.dbf	10/10/2012 1:32 PM	DBF
EdgewoodSD.prj	10/10/2012 1:32 PM	PRJ
EdgewoodSD.sbn	10/10/2012 1:32 PM	SBN
EdgewoodSD.sbx	10/10/2012 1:32 PM	Ado
EdgewoodSD.shp	10/10/2012 1:32 PM	SHP
EdgewoodSD.shp.ERICPIMPLER-PC.4384....	10/29/2012 9:37 PM	LOC
EdgewoodSD.shp.xml	10/10/2012 1:32 PM	XML
EdgewoodSD.shx	10/10/2012 1:32 PM	SHX

How it works...

The **Select by Location** tool requires that a feature layer be passed in as the first parameter. In this recipe, we pass in a feature layer that was created by the **Make Feature Layer** tool in the line just above. We used **Make Feature Layer** to create a feature layer from the `Burglary` feature class. This feature layer was assigned to the variable `flayer`, which is then passed into the **Select by Location** tool as the first parameter. In this script, we've also passed in a parameter that indicates the spatial relationship that we'd like to apply. Finally, we've also defined a source layer to use for the spatial relationship comparison. Other optional parameters that can be applied include a search distance and a selection type.

Combining a spatial and attribute query with the Select by Location tool

There may be times when you want to select features using a combined attribute and spatial query. For example, you might want to select all burglaries within the Edgewood school district that occurred on a Monday. This can be accomplished by running the **Select by Location** and **Select by Attributes** tools sequentially and applying a `SUBSET SELECTION` selection type.

Getting ready

This recipe will require that you create a feature layer that will serve as a temporary layer, which will be used with the **Select by Location** and **Select Layer by Attributes** tools. The **Select by Location** tool will find all burglaries that are within the Edgewood School District and apply a selection set to those features. The **Select Layer by Attributes** tool uses the same temporary feature layer and applies a `where` clause that finds all burglaries that occurred on Monday. In addition, the tool also specifies that the selection should be a subset of the currently selected features found by the **Select by Location** tool. Finally, you'll print the total number of records that were selected by the combined spatial and attribute query.

How to do it...

1. Open IDLE and create a new script window.
2. Save the script as `c:\ArcpyBook\Ch8\SpatialAttributeQuery.py`.
3. Import the `arcpy` module:

   ```
   import arcpy
   ```

4. Set the workspace to the City of San Antonio geodatabase:

   ```
   arcpy.env.workspace = "c:/ArcpyBook/data/CityofSanAntonio.gdb"
   ```

5. Start a `try` block. You'll have to indent the next lines up to the `except` block:

   ```
   try:
   ```

6. Create a variable for the query and define the `where` clause:

   ```
   qry = '"DOW" = \'Mon\''
   ```

7. Create the feature layer:

   ```
   flayer = arcpy.MakeFeatureLayer_management("Burglary","Burglary_
   Layer")
   ```

8. Execute the **Select by Location** tool to find all burglaries within the Edgewood School District.

   ```
   arcpy.SelectLayerByLocation_management (flayer, "COMPLETELY_
   WITHIN", "c:/ArcpyBook/Ch8/EdgewoodSD.shp")
   ```

9. Execute the **Select Layer by Attributes** tool to find all burglaries that match the query we defined previously in the `qry` variable. This should be defined as a subset query:

   ```
   arcpy.SelectLayerByAttribute_management(flayer, "SUBSET_
   SELECTION", qry)
   ```

10. Print the number of records that were selected:

    ```
    cnt = arcpy.GetCount_management(flayer)
    ```

11. Add the `except` block:

    ```
    except:
      print 'Error in selection'
    ```

    The entire script should appear as follows:

    ```
    import arcpy
    arcpy.env.workspace = "c:/ArcpyBook/data/CityOfSanAntonio.gdb"
    try:
      qry = '"DOW" = \'Mon\''
      flayer = arcpy.MakeFeatureLayer_management("Burglary","Burglary_
    Layer")
      arcpy.SelectLayerByLocation_management (flayer, "COMPLETELY_
    WITHIN", "c:/ArcpyBook/Ch8/EdgewoodSD.shp")
      arcpy.SelectLayerByAttribute_management(flayer, "SUBSET_
    SELECTION", qry)
           cnt = arcpy.GetCount_management(flayer)
    ```

```
      print "The total number of selected records is: " + str(cnt)
    except:
      print 'Error in selection'
```

12. Save and run the script.

    If everything was done correctly, you should see a message indicating that 197 records have been selected. This will select all the burglaries within the boundaries of the Edgewood school district that occurred on a Monday.

    ```
    The total number of selected records is: 197
    ```

How it works...

A new feature layer is created with the **Make Feature Layer** tool, and assigned to the variable `flayer`. This temporary layer is then used as an input to the **Select by Location** tool along with a spatial operator `COMPLETELY_WITHIN`, to find all the burglaries within the Edgewood School District. This same feature layer, with a selection set already defined, is then used as an input parameter to the **Select Layer by Attributes** tool. In addition to passing a reference to the feature layer, the Select Layer by Attributes tool is also passed a parameter that defines the selection type and a `where` clause. The selection type is set to `SUBSET_SELECTION`. This selection type creates a new selection that is combined with the existing selection. Only the records that are common to both remain selected. The `where` clause passed in as the third parameter is an attribute query to find all the burglaries that occurred on Monday. The query uses the `DOW` field and looks for a value of `Mon`. Finally, the **Get Count** tool is used against the `flayer` variable to get a count of the number of selected records, and this is printed on the screen.

9
Using the ArcPy Data Access Module to Select, Insert, and Update Geographic Data and Tables

In this chapter, we will cover the following recipes:

- Overview of cursor objects
- Retrieving features from a feature class with SearchCursor
- Filtering records with a where clause
- Improving cursor performance with Geometry tokens
- Inserting rows with InsertCursor
- Updating rows with UpdateCursor
- Deleting rows with UpdateCursor
- Inserting and updating rows inside an edit session
- Reading geometry from a feature class

Introduction

We'll start this chapter with a basic question. What are cursors? **Cursors** are in-memory objects containing one or more rows of data from a table or feature class. Each row contains the attributes from each field in the data source, along with the geometry for each feature. Cursors allow you to search, add, insert, update, and delete data from tables and feature classes.

The ArcPy Data Access module or `arcpy.da` is new to ArcGIS 10.1 and contains methods that allow you to iterate through each row in a cursor. Various types of cursors can be created. For example, search cursors can be created to read values from rows. Update cursors can be created to update values in rows or delete rows, and insert cursors can be created to insert new rows.

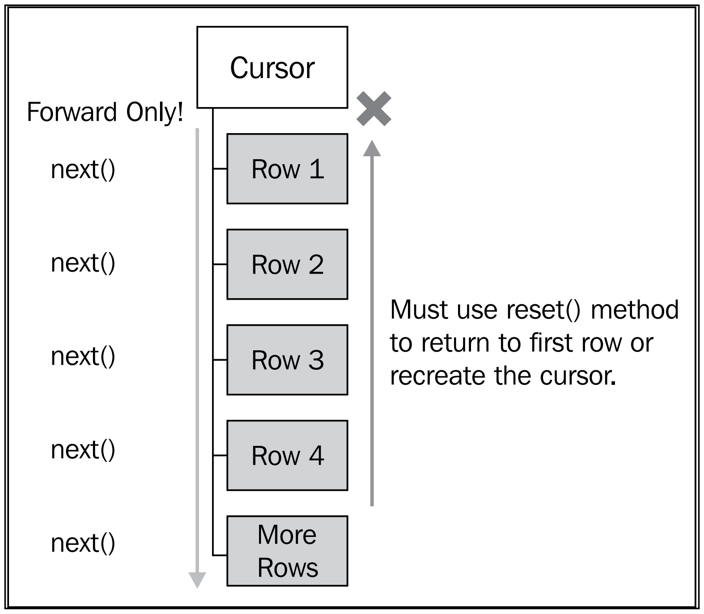
There are a number of cursor improvements that have been introduced with the ArcPy Data Access module. Prior to ArcGIS 10.1, cursor performance had been notoriously slow. Now, cursors are significantly faster. Esri has estimated that search cursors are up to 30 times faster while insert cursors are up to 12 times faster. In addition to these general performance improvements, the Data Access module also provides a number of new options that allow programmers to speed up processing. Rather than returning all the fields in a cursor, you can now specify that a subset of fields be returned. This increases the performance as less data needs to be returned. The same applies to geometry. Traditionally, when accessing the geometry of a feature, the entire geometric definition would be returned. You can now use geometry tokens to return a portion of the geometry rather than the full geometry for the feature. You can also use lists and tuples rather than using rows. Also new are edit sessions and the ability to work with versions, domains, and subtypes.

There are three cursor functions in `arcpy.da`. Each returns a cursor object having the same name as the function. `SearchCursor()` creates a read-only `SearchCursor` object containing rows from a table or feature class. `InsertCursor()` creates an `InsertCursor` object that can be used to insert new records into a table or feature class. `UpdateCursor()` returns a cursor object that can be used to edit or delete records from a table or feature class. Each of these cursor objects has methods for accessing rows in the cursor. You can see the relationship between the cursor functions, the objects they create, and how they are used as follows:

Function	**Object Created**	**Usage**
`SearchCursor()`	`SearchCursor`	Read-only view of data from a table or feature class
`InsertCursor()`	`InsertCursor`	Adds rows to a table or feature class
`UpdateCursor()`	`UpdateCursor`	Edit or delete rows in a table or feature class

The `SearchCursor()` function is used to return a `SearchCursor` object. This object can only be used to iterate through a set of rows returned for read-only purposes. No insertions, deletions, or updates can occur through this object. An optional `where` clause can be set to limit the rows returned.

Once you've obtained a cursor instance, it is common to iterate through the records, particularly with a `SearchCursor` or `UpdateCursor`. There are some peculiarities that you need to understand about navigating the records in a cursor. Cursor navigation is forward moving only. When a cursor is created, the pointer for the cursor sits just above the first row in the cursor. The first call to `next()` will move the pointer to the first row. Rather than calling the `next()` method, you can also use a `for` loop to process each of the records without the need to call the `next()` method. After performing whatever processing you need to do with this row, a subsequent call to `next()` will move the pointer to row 2. This process continues as long as you need to access additional rows. However, after a row has been visited, you can't go back a single record at a time. For instance, if the current row is row 3, you can't programmatically back up to row 2. You can only go forward. To revisit rows 1 and 2, you would need to either call the `reset()` method or recreate the cursor and move back through the object. As I mentioned, cursors are often navigated through the use of `for` loops as well. In fact, this is a more common way to iterate through a cursor and is a more efficient way to code your scripts. Cursor navigation is illustrated in the following diagram:



The `InsertCursor()` function is used to create an `InsertCursor` object that allows you to programmatically add new records to feature classes and tables. To insert rows, call the `insertRow()` method on this object. You can also retrieve a read-only tuple containing the field names in use by the cursor through the `fields` property. A lock is placed on the table or feature class being accessed through the cursor. It's important to always design your script in a way that releases the cursor when you are done.

The `UpdateCursor()` function can be used to create an `UpdateCursor` object that can update and delete rows in a table or feature class. As is the case with an `InsertCursor`, this function places a lock on the data while it's being edited or deleted. If the cursor is used inside a Python `with` statement, the lock will automatically be freed after the data has been processed. This hasn't always been the case. Prior to ArcGIS 10.1, cursors were required to be manually released using the Python `del` statement. Once an instance of `UpdateCursor` has been obtained, you can then call the `updateCursor()` method to update records in tables or feature classes and the `deleteRow()` method to delete a row.

The subject of data locks requires a little more explanation. Insert and update cursors must obtain a lock on the data source they reference. This means that no other application can concurrently access this data source. Locks are a way of preventing multiple users from changing data at the same time and thus corrupting the data. When the `InsertCursor()` and `UpdateCursor()` methods are called in your code, Python attempts to acquire a lock on the data. This lock must be specifically released after the cursor has finished processing, so that other users running applications such as `ArcMap` or `ArcCatalog` can access the data sources. Otherwise, no other application will be able to access the data. Prior to ArcGIS 10.1 and the `with` statement, cursors had to be specifically unlocked through Python's `del` statement. Similarly, `ArcMap` and `ArcCatalog` also acquire data locks when updating or deleting data. If a data source has been locked by either of these applications, your Python code will not be able to access the data. Therefore, best practice is to close `ArcMap` and `ArcCatalog` before running any standalone Python scripts that use insert or update cursors.

In this chapter, we're going to cover the use of cursors for accessing and editing tables and feature classes. However, many of the cursor concepts that existed before ArcGIS 10.1 still apply.

Retrieving features from a feature class with a SearchCursor

There are many occasions when you need to retrieve rows from a table or feature class for read-only purposes. For example, you might want to generate a list of all land parcels in a city with a value of greater than $100,000. In this case, you don't have any need to edit the data. Your needs are met simply by generating a list of rows that meet some sort of criteria.

Getting ready

The `SearchCursor()` function is used to return a `SearchCursor` object. This object can only be used to iterate through a set of rows returned for read-only purposes. No insertions, deletions, or updates can occur through this object. An optional `where` clause can be set to limit the rows returned. In this recipe, you will learn how to create a basic `SearchCursor` object on a feature class through the use of the `SearchCursor()` function.

The `SearchCursor` object contains a `fields` property along with `next()` and `reset()` methods. The `fields` property is a read-only structure in the form of a Python tuple, containing the fields requested from the feature class or table. You are going to hear the term tuple a lot in conjunction with cursors. If you haven't covered this topic before, tuples are a Python structure for storing a sequence of data similar to Python lists. But there are some important differences between Python tuples and lists. Tuples are defined as a sequence of values inside parentheses while lists are defined as a sequence of values inside brackets. Unlike lists, tuples can't grow and shrink, which can be a very good thing in some cases when you want data values to occupy a specific position each time. Such is the case with cursor objects that use tuples to store data from fields in tables and feature classes.

How to do it...

Follow these steps to learn how to retrieve rows from a table or feature class inside a `SearchCursor` object:

1. Open IDLE and create a new script window.
2. Save the script as `c:\ArcpyBook\Ch9\SearchCursor.py`.
3. Import the `arcpy.da` module:

   ```
   import arcpy.da
   ```

4. Set the workspace:

   ```
   arcpy.env.workspace = "c:/ArcpyBook/Ch9"
   ```

5. Use a Python `with` statement to create a cursor:

   ```
   with arcpy.da.SearchCursor("Schools.shp",("Facility","Name"))
   as cursor:
   ```

6. Loop through each row in the `SearchCursor` and print the name of the school. Make sure you indent the `for` loop inside the `with` block:

   ```
   for row in sorted(cursor):
     print("School name: " + row[1])
   ```

7. Save the script.
8. Run the script. You should see the following output:

```
School name: ALLAN
School name: ALLISON
School name: ANDREWS
School name: BARANOFF
School name: BARRINGTON
School name: BARTON CREEK
School name: BARTON HILLS
School name: BATY
School name: BECKER
School name: BEE CAVE
```

How it works...

The `with` statement, used with the `SearchCursor()` function, will create, open, and close the cursor. So you no longer have to be concerned with explicitly releasing the lock on the cursor as you did prior to ArcGIS 10.1. The first parameter passed into the `SearchCursor()` function is a feature class, represented by the `Schools.shp` file. The second parameter is a Python tuple containing a list of fields that we want returned in the cursor. For performance reasons, it is a best practice to limit the fields returned in the cursor to only those that you need to complete the task. Here, we've specified that only the `Facility` and `Name` fields should be returned. The `SearchCursor` object is stored in a variable called `cursor`.

Inside the `with` block, we are using a Python `for` loop to loop through each school returned. We're also using the Python `sorted()` function to sort the contents of the cursor. To access the values from a field on the row, simply use the index number of the field you want to return. In this case, we want to return the contents of the `Name` column, which will be index number `1`, since it is the second item in the tuple of field names that are returned.

Filtering records with a where clause

By default, `SearchCursor` will contain all rows in a table or feature class. However, in many cases, you will want to restrict the number of rows returned by some sort of criteria. Applying a filter through the use of a `where` clause limits the records returned.

Getting ready

By default, all rows from a table or feature class will be returned when you create a `SearchCursor` object. However, in many cases, you will want to restrict the records returned. You can do this by creating a query and passing it in as a `where` clause parameter when calling the `SearchCursor()` function. In this recipe, you'll build on the script you created in the previous recipe, by adding a `where` clause that restricts the records returned.

How to do it...

Follow these steps to apply a filter to a `SearchCursor` object that restricts the rows returned from a table or feature class:

1. Open IDLE and load the `SearchCursor.py` script that you created in the previous recipe.
2. Update the `SearchCursor()` function by adding a `where` clause that queries the `facility` field for records that have the text `High School`:

```
with arcpy.da.SearchCursor("Schools.shp",("Facility","Name"),
'"FACILITY" = \'HIGH SCHOOL\'') as cursor:
```

3. Save and run the script. The output will now be much smaller and restricted to only those schools that are high schools:

```
High school name: AKINS
High school name: ALTERNATIVE LEARNING CENTER
High school name: ANDERSON
High school name: AUSTIN
High school name: BOWIE
High school name: CROCKETT
High school name: DEL VALLE
High school name: ELGIN
High school name: GARZA
High school name: HENDRICKSON
High school name: JOHN B CONNALLY
High school name: JOHNSTON
High school name: LAGO VISTA
```

How it works...

We covered the creation of queries in *Chapter 8*, *Querying and Selecting Data*, so hopefully you now have a good grasp of how these are created along with all the rules you need to follow when coding these structures. The `where` clause parameter accepts any valid SQL query, and is used in this case to restrict the number of records that are returned.

Improving cursor performance with geometry tokens

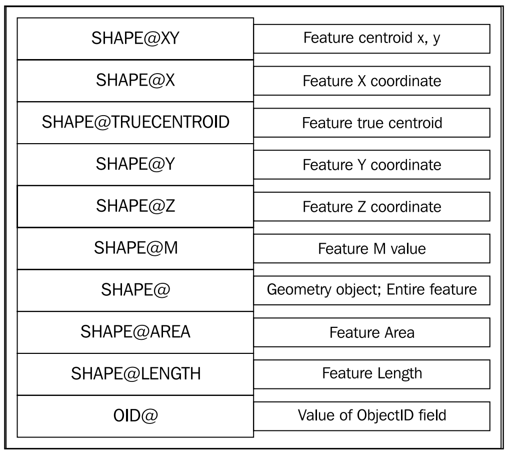
Geometry tokens were introduced in ArcGIS 10.1 as a performance improvement for cursors. Rather than returning the entire geometry of a feature inside the cursor, only a portion of the geometry is returned. Returning the entire geometry of a feature can result in decreased cursor performance due to the amount of data that has to be returned. It's significantly faster to return only the geometry that is needed.

Getting ready

A token is provided as one of the fields in the `field` list passed into the constructor for a cursor and is in the format `SHAPE@<Part of Feature to be Returned>`. The only exception to this format is the `OID@` token, which returns the object ID of the feature. The following code example retrieves the x and y coordinates of a feature:

```
with arcpy.da.SearchCursor(fc, ("SHAPE@XY","Facility","Name"))
as cursor:
```

The following table lists the available geometry tokens. Not all cursors support the full list of tokens. Please check the ArcGIS help files for information on the tokens supported by each cursor type. The `SHAPE@` token returns the entire geometry of the feature. Use this carefully though, because it is an expensive operation to return the entire geometry of a feature and can dramatically affect performance. If you don't need the entire geometry, then do not include this token!



SHAPE@XY	Feature centroid x, y
SHAPE@X	Feature X coordinate
SHAPE@TRUECENTROID	Feature true centroid
SHAPE@Y	Feature Y coordinate
SHAPE@Z	Feature Z coordinate
SHAPE@M	Feature M value
SHAPE@	Geometry object; Entire feature
SHAPE@AREA	Feature Area
SHAPE@LENGTH	Feature Length
OID@	Value of ObjectID field

In this recipe, you will use a geometry token to increase the performance of a cursor. You'll retrieve the x and y coordinates of each land parcel from a `parcels` feature class, along with some attribute information about the parcel.

How to do it...

Follow these steps to add a geometry token to a cursor, which should improve the performance of this object:

1. Open IDLE and create a new script window.
2. Save the script as `c:\ArcpyBook\Ch9\GeometryToken.py`.
3. Import the `arcpy.da` module:
   ```
   import arcpy.da, time
   ```
4. Set the workspace:
   ```
   arcpy.env.workspace = "c:/ArcpyBook/Ch9"
   ```
5. We're going to measure how long it takes to execute the code using a geometry token. Add a start time for the script:
   ```
   start = time.clock()
   ```
6. Use a Python `with` statement to create a cursor that includes the centroid of each feature as well as the ownership information stored in the `PY_FULL_OW` field:
   ```
   with arcpy.da.SearchCursor("coa_parcels.shp",("PY_FULL_OW","SHAPE@
   XY")) as cursor:
   ```

7. Loop through each row in `SearchCursor` and print the name of the school. Make sure you indent the `for` loop inside the `with` block:

   ```
   for row in cursor:
     print("Parcel owner: {0} has a location of: {1}".format(row[0],
   row[1]))
   ```

8. Measure the elapsed time:

   ```
   elapsed = (time.clock() - start)
   ```

9. Print the execution time:

   ```
   print "Execution time: " + str(elapsed)
   ```

10. Save the script.
11. Run the script. You should see something similar to the following output. Note the execution time; your time will vary:

    ```
    Parcel owner: CITY OF AUSTIN ATTN REAL ESTATE DIVISION has a
    location of: (3110480.5197341456, 10070911.174956793)
    Parcel owner: CITY OF AUSTIN ATTN REAL ESTATE DIVISION has a
    location of: (3110670.413783513, 10070800.960865)
    Parcel owner: CITY OF AUSTIN has a location of:
    (3143925.0013213265, 10029388.97419636)
    Parcel owner: CITY OF AUSTIN % DOROTHY NELL ANDERSON ATTN
    BARRY LEE ANDERSON has a location of: (3134432.983822767,
    10072192.047894118)
    Execution time: 9.08046185109
    ```

Now, we're going to measure the execution time if the entire geometry is returned instead of just the portion of the geometry that we need:

1. Save a new copy of the script as `c:\ArcpyBook\Ch9\GeometryTokenEntireGeometry.py`.
2. Change the `SearchCursor()` function to return the entire geometry using `SHAPE@` instead of `SHAPE@XY`:

   ```
   with arcpy.da.SearchCursor("coa_parcels.shp",("PY_FULL_
   OW","SHAPE@")) as cursor:
   ```

3. Save and run the script. You should see the following output. Your time will vary from mine, but notice that the execution time is slower. In this case, it's only a little over a second slower but we're only returning 2600 features. If the feature class were significantly large, as many are, this would be amplified:

   ```
   Parcel owner: CITY OF AUSTIN ATTN REAL ESTATE DIVISION has a
   location of: <geoprocessing describe geometry object object at
   0x06B9BE00>
   ```

```
Parcel owner: CITY OF AUSTIN ATTN REAL ESTATE DIVISION has a
location of: <geoprocessing describe geometry object object at
0x2400A700>
Parcel owner: CITY OF AUSTIN has a location of: <geoprocessing
describe geometry object object at 0x06B9BE00>
Parcel owner: CITY OF AUSTIN % DOROTHY NELL ANDERSON ATTN BARRY
LEE ANDERSON has a location of: <geoprocessing describe geometry
object object at 0x2400A700>
Execution time: 10.1211390896
```

How it works...

A geometry token can be supplied as one of the field names supplied in the constructor for the cursor. These tokens are used to increase the performance of a cursor by returning only a portion of the geometry instead of the entire geometry. This can dramatically increase the performance of a cursor, particularly when you are working with large polyline or polygon datasets. If you only need specific properties of the geometry in your cursor, you should use these tokens.

Inserting rows with InsertCursor

You can insert a row into a table or feature class using an `InsertCursor` object. If you want to insert attribute values along with the new row, you'll need to supply the values in the order found in the attribute table.

Getting ready

The `InsertCursor()` function is used to create an `InsertCursor` object that allows you to programmatically add new records to feature classes and tables. The `insertRow()` method on the `InsertCursor` object adds the row. A row, in the form of a list or tuple, is passed into the `insertRow()` method. The values in the list must correspond to the field values defined when the `InsertCursor` object was created. Just as with the other types of cursors, you can also limit the field names returned using the second parameter of the method. This function supports geometry tokens as well.

The following code example illustrates how you can use `InsertCursor` to insert new rows into a feature class. Here, we are inserting two new wildfire points into the `California` feature class. The row values to be inserted are defined in a `list` variable. Then, an `InsertCursor` object is created, passing in the feature class and fields. Finally, the new rows are inserted into the feature class using the `insertRow()` method:

```
rowValues = [{'Bastrop','N',3000,(-105.345,32.234)),('Ft Davis','N',
456, (-109.456,33.468))]
fc = "c:/data/wildfires.gdb/California"
```

```
fields["FIRE_NAME", "FIRE_CONTAINED", "ACRES", "SHAPE@XY"]
with arcpy.da.InsertCursor(fc, fields) as cursor:
  for row in rowValues:
    cursor.insertRow(row)
```

In this recipe, you will use `InsertCursor` to add wildfires retrieved from a `text` file into a point feature class. When inserting rows into a feature class, you will need to know how to add the geometric representation of a feature into the feature class. This can be accomplished using `InsertCursor` along with two miscellaneous objects: `Array` and `Point`. In this exercise, we will add point features in the form of wildfire incidents to an empty point feature class. In addition, you will use Python file manipulation techniques to read the coordinate data from a text file.

How to do it...

We will be importing North American wildland fire incident data from a single day in October, 2007. This data is contained in a comma-delimited text file containing one line for each fire incident on that particular day. Each fire incident has a latitude/longitude coordinate pair separated by commas along with a confidence value. This data was derived by automated methods that use remote sensing data to derive the presence or absence of a wildfire. Confidence values can range from 0 to 100. Higher numbers represent a greater confidence that this is indeed a wildfire:

1. Open the file at `c:\ArcpyBook\Ch9\Wildfire Data\NorthAmericaWildfire_2007275.txt` and examine the contents.

   You will notice that this is a simple comma-delimited text file containing longitude and latitude values for each fire along with a confidence value. We will use Python to read the contents of this file line-by-line and insert new point features into the `FireIncidents` feature class located in the `c:\ArcpyBook\Ch9 \WildfireData\WildlandFires.mdb` personal geodatabase.

2. Close the file.
3. Open `ArcCatalog`.
4. Navigate to `c:\ArcpyBook\Ch9\WildfireData`.

   You should see a personal geodatabase called `WildlandFires`. Open this geodatabase and you will see a point feature class called `FireIncidents`. Right now this is an empty feature class. We will add features by reading the text file you examined earlier and inserting points.

5. Right-click on `FireIncidents` and select **Properties**.
6. Click on the **Fields** tab.

The latitude/longitude values found in the file we examined earlier will be imported into the `SHAPE` field and the confidence values will be written to the `CONFIDENCEVALUE` field.

7. Open IDLE and create a new script.
8. Save the script to `c:\ArcpyBook\Ch9\InsertWildfires.py`.
9. Import the `arcpy` and `os` modules:

   ```
   import arcpy,os
   ```

10. Set the workspace:

    ```
    arcpy.env.workspace = "C:/ArcpyBook/Ch9/WildfireData/
    WildlandFires.mdb"
    ```

11. Open the text file and read all the lines into a list:

    ```
    f = open("C:/ArcpyBook/Ch9/WildfireData/
    NorthAmericaWildfires_2007275.txt","r")
    lstFires = f.readlines()
    ```

12. Start a `try` block:

    ```
    try:
    ```

13. Create an `InsertCursor` object using a `with` block. Make sure you indent inside the `try` statement. The cursor will be created on the `FireIncidents` feature class:

    ```
    with da.InsertCursor("FireIncidents",("SHAPE@
    XY","CONFIDENCEVALUE")) as cur:
    ```

14. Create a counter variable that will be used to print the progress of the script:

    ```
    cntr = 1
    ```

15. Loop through the text file line by line using a `for` loop. Since the text file is comma-delimited, we'll use the Python `split()` function to separate each value into a list variable called `vals`. We'll then pull out the individual latitude, longitude, and confidence value items and assign them to variables. Finally, we'll place these values into a list variable called `rowValue`, which is then passed into the `insertRow()` function for the `InsertCursor` object, and then we'll print a message:

    ```
    for fire in lstFires:
          if 'Latitude' in fire:
            continue
          vals = fire.split(",")
          latitude = float(vals[0])
          longitude = float(vals[1])
          confid = int(vals[2])
    ```

```
      rowValue = [(latitude,longitude),confid]
      cur.insertRow(rowValue)
      print "Record number " + str(cntr) + " written to feature
class"
      #arcpy.AddMessage("Record number" + str(cntr) + " written
to feature class")
      cntr = cntr + 1
```

16. Add the `except` block to print any errors that may occur:

```
except Exception as e:
  print e.message
```

17. Add a `finally` block to close the text file:

```
finally:
  f.close()
```

18. The entire script should appear as follows:

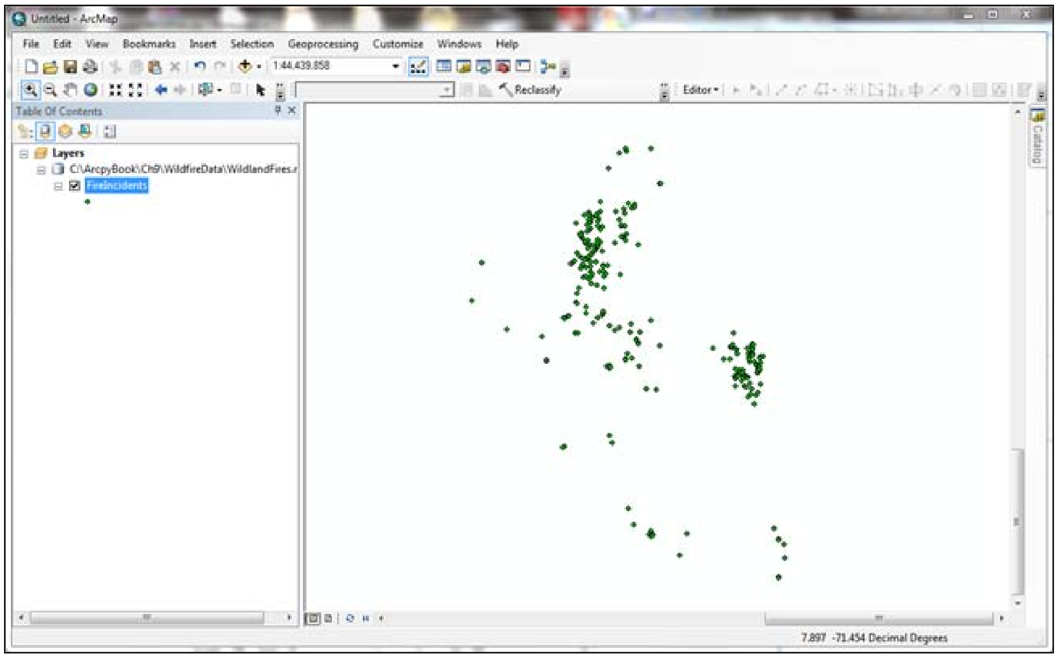
```
import arcpy, os

arcpy.env.workspace = "C:/ArcpyBook/Ch9/WildfireData/
WildlandFires.mdb"
f = open("C:/ArcpyBook/Ch9/WildfireData/
NorthAmericaWildfires_2007275.txt","r")
lstFires = f.readlines()
try:
  with da.InsertCursor("FireIncidents",("SHAPE@
XY","CONFIDENCEVALUE")) as cur:
    cntr = 1
    for fire in lstFires:
      if 'Latitude' in fire:
        continue
      vals = fire.split(",")
      latitude = float(vals[0])
      longitude = float(vals[1])
      confid = int(vals[2])
      rowValue = [(latitude,longitude),confid]
      cur.insertRow(rowValue)
      print "Record number " + str(cntr) + " written to feature
class"
      #arcpy.AddMessage("Record number" + str(cntr) + " written
to feature class")
```

```
      cntr = cntr + 1
except Exception as e:
  print e.message
finally:
  f.close()
```

19. Save and run the script. You should see messages being written to the output window as the script runs:

```
Record number: 406 written to feature class
Record number: 407 written to feature class
Record number: 408 written to feature class
Record number: 409 written to feature class
Record number: 410 written to feature class
Record number: 411 written to feature class
```

20. Open `ArcMap` and add the `FireIncidents` feature class to the table of contents. The points should be visible, as shown in the following screenshot:


Untitled - ArcMap
File Edit View Bookmarks Insert Selection Geoprocessing Customize Windows Help
1:44,439,858
Reclassify
Editor
Table Of Contents
Layers
C:\ArcpyBook\Ch9\WildfireData\WildlandFires.
FireIncidents
Catalog
7.897 -71.454 Decimal Degrees

How it works...

Some additional explanation may be needed here. The `lstFires` variable contains a list of all the wildfires that were contained in the comma-delimited text file. The `for` statement will loop through each of these records one by one, inserting each individual record into the `fire` variable. We also include an `if` statement that is used to skip the first record in the file, which serves as the header. As I explained earlier, we then pull out the individual latitude, longitude, and confidence value items from the `vals` variable, which is just a Python list object and assign them to variables called `latitude`, `longitude`, and `confid`. We then place these values into a new list variable called `rowValue` in the order that we defined when we created `InsertCursor`. That is, the latitude and longitude pair should be placed first, followed by the confidence value. Finally, we call the `insertRow()` function on the `InsertCursor` object assigned to the variable `cur`, passing in the new `rowValue` variable. We close by printing a message that indicates the progress of the script and also creating our `except` and `finally` blocks to handle errors and close the text file. Placing the `file.close()` method in the `finally` block ensures that it will execute, and close the file, even if there is an error in the previous try statement.

Updating rows with an UpdateCursor

If you need to edit or delete rows from a table or feature class, you can use `UpdateCursor`. As is the case with `InsertCursor`, the contents of `UpdateCursor` can be limited through the use of a `where` clause.

Getting ready

The `UpdateCursor()` function can be used to either update or delete rows in a table or feature class. The returned cursor places a lock on the data, which will automatically be released if used inside a Python `with` statement. An `UpdateCursor` object is returned from a call to this method.

The `UpdateCursor` object places a lock on the data while it's being edited or deleted. If the cursor is used inside a Python `with` statement, the lock will automatically be freed after the data has been processed. This hasn't always been the case. Previous versions of cursors were required to be manually released using the Python `del` statement. Once an instance of `UpdateCursor` has been obtained, you can then call the `updateCursor()` method to update records in tables or feature classes and the `deleteRow()` method to delete a row.

In this recipe, you're going to write a script that updates each feature in the `FireIncidents` feature class by assigning a value of `poor`, `fair`, `good`, or `excellent` to a new field that is more descriptive of the confidence values using an `UpdateCursor`. Prior to updating the records, your script will add a new field to the `FireIncidents` feature class.

How to do it...

Follow these steps to create an `UpdateCursor` object that will be used to edit rows in a feature class:

1. Open IDLE and create a new script.
2. Save the script to `c:\ArcpyBook\Ch9\UpdateWildfires.py`.
3. Import the `arcpy` and `os` modules:

   ```
   import arcpy,os
   ```

4. Set the workspace:

   ```
   arcpy.env.workspace = "C:/ArcpyBook/Ch9/WildfireData/
   WildlandFires.mdb"
   ```

5. Start a `try` block:

   ```
   try:
   ```

6. Add a new field called `CONFID_RATING` to the `FireIncidents` feature class. Make sure to indent inside the `try` statement:

   ```
   arcpy.AddField_management("FireIncidents","CONFID_
   RATING","TEXT","10")
   print "CONFID_RATING field added to FireIncidents"
   ```

7. Create a new instance of `UpdateCursor` inside a `with` block:

   ```
   with arcpy.da.UpdateCursor("FireIncidents",("CONFIDENCEVALUE","CON
   FID_RATING")) as cursor:
   ```

8. Create a counter variable that will be used to print the progress of the script. Make sure you indent this line of code and all the lines of code to follow inside the `with` block:

   ```
   cntr = 1
   ```

9. Loop through each of the rows in the `FireIncidents` fire class. Update the `CONFID_RATING` field according to the following guidelines:
   - Confidence value 0 to 40 = `POOR`
   - Confidence value 41 to 60 = `FAIR`
   - Confidence value 61 to 85 = `GOOD`
   - Confidence value 86 to 100 = `EXCELLENT`

     ```
     for row in cursor:
           # update the confid_rating field
           if row[0] <= 40:
     ```

```
                  row[1] = 'POOR'
                  cursor.updateRow(row)
                elif row[0] > 40 and row[0] <= 60:
                  row[1] = 'FAIR'
                  cursor.updateRow(row)
                elif row[0] > 60 and row[0] <= 85:
                  row[1] = 'GOOD'
                  cursor.updateRow(row)
                else:
                  row[1] = 'EXCELLENT'
                  cursor.updateRow(row)
          print "Record number " + str(cntr) + " updated"
          cntr = cntr + 1
```

10. Add the `except` block to print any errors that may occur:

```
except Exception as e:
  print e.message
```

11. The entire script should appear as follows:

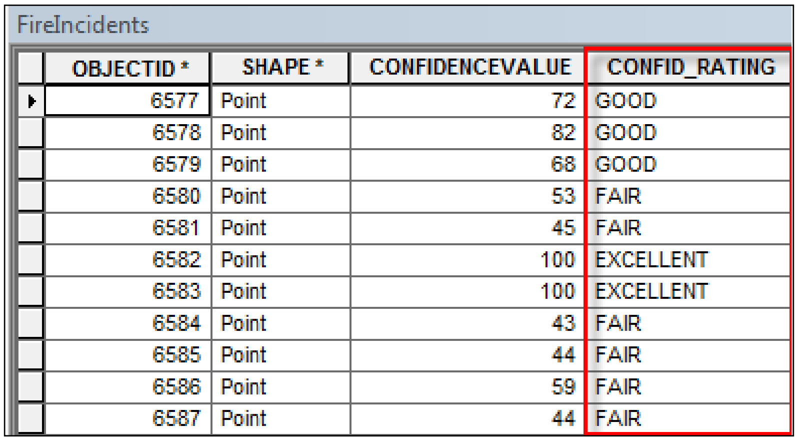
```
import arcpy, os

arcpy.env.workspace = "C:/ArcpyBook/Ch9/WildfireData/
WildlandFires.mdb"
try:
  #create a new field to hold the values
  arcpy.AddField_management("FireIncidents","CONFID_
RATING","TEXT","10")
  print "CONFID_RATING field added to FireIncidents"
  with arcpy.da.UpdateCursor("FireIncidents",("CONFIDENCEVALUE","C
ONFID_RATING")) as cursor:
    cntr = 1
    for row in cursor:
      # update the confid_rating field
      if row[0] <= 40:
        row[1] = 'POOR'
      elif row[0] > 40 and row[0] <= 60:
        row[1] = 'FAIR'
      elif row[0] > 60 and row[0] <= 85:
        row[1] = 'GOOD'
      else:
        row[1] = 'EXCELLENT'
```

```
            cursor.updateRow(row)
print "Record number " + str(cntr) + " updated"
            cntr = cntr + 1
except Exception as e:
    print e.message
```

12. Save and run the script. You should see messages being written to the output window as the script runs:

```
Record number 406 updated
Record number 407 updated
Record number 408 updated
Record number 409 updated
Record number 410 updated
```

13. Open `ArcMap` and add the `FireIncidents` feature class. Open the attribute table and you should see that a new `CONFID_RATING` field has been added and populated by `UpdateCursor`:



FireIncidents

OBJECTID *	SHAPE *	CONFIDENCEVALUE	CONFID_RATING
6577	Point	72	GOOD
6578	Point	82	GOOD
6579	Point	68	GOOD
6580	Point	53	FAIR
6581	Point	45	FAIR
6582	Point	100	EXCELLENT
6583	Point	100	EXCELLENT
6584	Point	43	FAIR
6585	Point	44	FAIR
6586	Point	59	FAIR
6587	Point	44	FAIR

When you insert, update, or delete data in cursors, the changes are permanent and can't be undone if you're working outside an edit session. However, with the new edit session functionality provided by ArcGIS 10.1, you can now make these changes inside an edit session to avoid these problems. We'll cover edit sessions soon.

How it works...

In this case, we've used `UpdateCursor` to update each of the features in a feature class. We first used the `Add Field` tool to add a new field called `CONFID_RATING`, which will hold new values that we assign based on values found in another field. The groups are poor, fair, good, and excellent, and are based on numeric values found in the `CONFIDENCEVALUE` field. We then created a new instance of `UpdateCursor` based on the `FireIncidents` feature class, and returned the two fields mentioned previously. The script then loops through each of the features and assigns a value of poor, fair, good, or excellent to the `CONFID_RATING` field (`row[1]`), based on the numeric value found in `CONFIDENCEVALUE`. A Python `if/elif/else` structure is used to control the flow of the script based on the numeric value. The value for `CONFID_RATING` is then committed to the feature class by passing in the row variable into the `updateRow()` method.

Deleting rows with an UpdateCursor

In addition to being used to edit rows in a table or feature class, an `UpdateCursor` can also be used to delete rows. Please keep in mind that when rows are deleted outside an edit session, the changes are permanent.

Getting ready

In addition to updating records, `UpdateCursor` can also delete records from a table or feature class. The `UpdateCursor` object is created in the same way in either case, but instead of calling `updateRow()`, you call `deleteRow()` to delete a record. You can also apply a `where` clause to `UpdateCursor`, to limit the records returned. In this recipe, we'll use an `UpdateCursor` object that has been filtered using a `where` clause to delete records from our `FireIncidents` feature class.

How to do it...

Follow these steps to create an `UpdateCursor` object that will be used delete rows from a feature class:

1. Open IDLE and create a new script.
2. Save the script to `c:\ArcpyBook\Ch9\DeleteWildfires.py`.
3. Import the `arcpy` and `os` modules:

   ```
   import arcpy,os
   ```

4. Set the workspace:

```
arcpy.env.workspace = "C:/ArcpyBook/Ch9/WildfireData/
WildlandFires.mdb"
```

5. Start a `try` block:

```
try:
```

6. Create a new instance of `UpdateCursor` inside a `with` block:

```
with arcpy.da.UpdateCursor("FireIncidents",("CONFID_
RATING"),'[CONFID_RATING] = \'POOR\'') as cursor:
```

7. Create a counter variable that will be used to print the progress of the script. Make sure you indent this line of code and all the lines of code to follow inside the `with` block:

```
cntr = 1
```

8. Delete the returned rows by calling the `deleteRow()` method. This is done by looping through the returned cursor and deleting the rows one at a time:

```
for row in cursor:
  cursor.deleteRow()
     print "Record number " + str(cntr) + " deleted"
     cntr = cntr + 1
```

9. Add the `except` block to print any errors that may occur:

```
except Exception as e:
  print e.message
```

10. The entire script should appear as follows:

```
import arcpy, os

arcpy.env.workspace = "C:/ArcpyBook/Ch9/WildfireData/
WildlandFires.mdb"
try:
  with arcpy.da.UpdateCursor("FireIncidents",("CONFID_
RATING"),'[CONFID_RATING] = \'POOR\'') as cursor:
    cntr = 1
    for row in cursor:
      cursor.deleteRow()
      print "Record number " + str(cntr) + " deleted"
      cntr = cntr + 1
except Exception as e:
  print e.message
```

11. Save and run the script. You should see messages being written to the output window as the script runs. 37 records should be deleted from the `FireIncidents` feature class:

    ```
    Record number 1 deleted
    Record number 2 deleted
    Record number 3 deleted
    Record number 4 deleted
    Record number 5 deleted
    ```

How it works...

Rows from feature classes and tables can be deleted using the `deleteRow()` method on `UpdateCursor`. In this recipe, we used a `where` clause in the constructor of `UpdateCursor` to limit the records returned to only those features with a `CONFID_RATING` of `POOR`. We then looped through the features returned in the cursor and called the `deleteRow()` method to delete the row from the feature class.

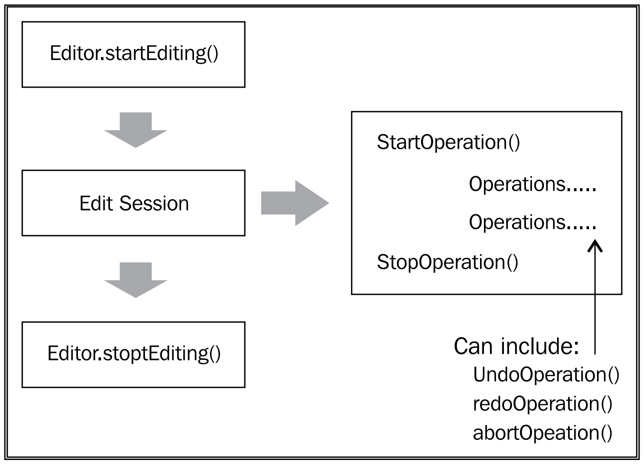
Inserting and updating rows inside an edit session

As I've mentioned throughout the chapter, inserts, updates, or deletes to a table or feature class done outside an edit session are permanent. They can't be undone. Edit sessions give you much more flexibility for rolling back any unwanted changes.

Getting ready

Up until now, we've used insert and update cursors to add, edit, and delete data from feature classes and tables. These changes have been permanent as soon as the script was executed and can't be undone. The new `Editor` class in the Data Access module supports the ability to create edit sessions and operations. With edit sessions, changes applied to feature classes or tables are temporary until permanently applied with a specific method call. This is the same functionality provided by the `Edit` toolbar in ArcGIS Desktop.

Edit sessions begin with a call to `Editor.startEditing()`, which initiates the session. Inside the session, you then start an operation with the `Editor.startOperation()` method. From within this operation, you then perform various operations that perform edits on your data. These edits can also be subject to undo, redo, and abort operations for rolling back, rolling forward, and aborting your editing operations. After the operations have been completed, you then call the `Editor.stopOperation()` method followed by `Editor.stopEditing()`. Sessions can be ended without saving changes. In this event, changes are not permanently applied. An overview of this process is provided in the following screenshot:



Edit sessions can also be ended without saving changes. In this event, changes are not permanently applied. Edit sessions also allow for operations to be applied inside the session and then either applied permanently to the database or rolled back. In addition, the `Editor` class also supports undo and redo operations.

The following code example shows the full edit session stack including the creation of the `Editor` object, the beginning of an edit session and an operation, edits to the data (an insert in this case), stopping the operation, and finally the end of the edit session by saving the data:

```
edit = arcpy.da.Editor('Database Connections/Portland.sde')
edit.startEditing(False)
edit.startOperation()
with arcpy.da.InsertCursor("Portland.jgp.schools",("SHAPE","Name"))
as cursor:
  cursor.insertRow([7642471.100, 686465.725), 'New School'])
edit.stopOperation()
edit.stopEditing(True)
```

The `Editor` class can be used with personal, file, and ArcSDE geodatabases. In addition, sessions can also be started and stopped on versioned databases. You are limited to editing only a single workspace at a time, and this workspace is specified in the constructor for the `Editor` object simply by passing in a string that references the workspace. Once created, this `Editor` object then has access to all the methods for starting, stopping, and aborting operations, and performing undo and redo operations.

How to do it...

Follow these steps to wrap `UpdateCursor` inside an edit session:

1. Open IDLE.
2. Open the `c:\ArcpyBook\Ch9\UpdateWildfires.py` script and save it to a new script called `c:\ArcpyBook\Ch9\EditSessionUpdateWildfires.py`.
3. We're going to make several alterations to this existing script that updates values in the `CONFID_RATING` field.
4. Remove the following lines of code:

   ```
   arcpy.AddField_management("FireIncidents","CONFID_
   RATING","TEXT","10")
   print "CONFID_RATING field added to FireIncidents"
   ```

5. Create an instance of the `Editor` class and start an edit session. These lines of code should be placed just inside the `try` block:

   ```
   edit = arcpy.da.Editor('C:\ArcpyBook\data\CityOfSanAntonio.gdb')
   edit.startEditing(True)
   ```

6. Alter the `if` statement so that it appears as follows:

   ```
   if row[0] > 40 and row[0] <= 60:
     row[1] = 'GOOD'
   elif row[0] > 60 and row[0] <= 85:
     row[1] = 'BETTER'
   else:
     row[1] = 'BEST'
   ```

7. End the edit session and save edits. Place this line of code just below the counter increment:

   ```
   edit.stopEditing(True)
   ```

8. The entire script should appear as follows:

   ```
   import arcpy, os

   arcpy.env.workspace = "C:/ArcpyBook/Ch9/WildfireData/
   WildlandFires.mdb"
   ```

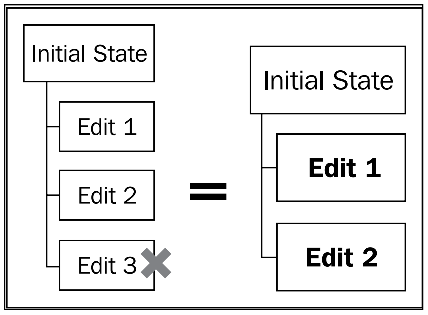
```
try:
  edit = arcpy.da.Editor('C:\ArcpyBook\data\CityOfSanAntonio.gdb')
  edit.startEditing(True)
  with arcpy.da.UpdateCursor("FireIncidents",("CONFIDENCEVALUE","C
ONFID_RATING")) as cursor:
    cntr = 1
    for row in cursor:
      # update the confid_rating field
      if row[0] > 40 and row[0] <= 60:
        row[1] = 'GOOD'
      elif row[0] > 60 and row[0] <= 85:
        row[1] = 'BETTER'
      else:
        row[1] = 'BEST'
      cursor.updateRow(row)
      print "Record number " + str(cntr) + " updated"
      cntr = cntr + 1
  edit.stopEditing(True)
except Exception as e:
  print e.message
```

9. Save and run the script.

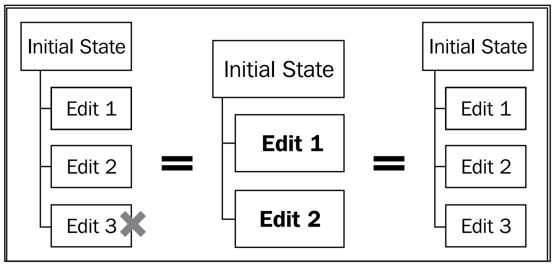
How it works...

Edit operations should take place inside an edit session, which can be initiated with the `Editor.startEditing()` method. The `startEditing()` method takes two optional parameters including `with_undo` and `multiuser_mode`. The `with_undo` parameter accepts a Boolean value of `true` or `false`, with a default of `true`. This creates an undo/redo stack when set to `true`. The `multiuser_mode` parameter defaults to `true`. When `false`, you have full control of editing a non-versioned or versioned dataset. If your dataset is non-versioned and you use `stopEditing(False)`, your edits will not be committed. Otherwise, if set to `true`, your edits will be committed. The `Editor.stopEditing()` method takes a single Boolean value of `true` or `false`, indicating whether changes should be saved or not. This defaults to `true`.

The `Editor` class supports undo and redo operations. We'll first look at undo operations. During an edit session, various edit operations can be applied. In the event that you need to undo a previous operation, a call to `Editor.undoOperation()` will remove the most previous edit operation in the stack. This is illustrated as follows:



Redo operations, initiated by the `Editor.redoOperation()` method, will redo an operation that was previously undone. This is illustrated as follows:



Reading geometry from a feature class

There may be times when you need to retrieve the geometric definition of features in a feature class. ArcPy provides the ability to read this information through various objects.

Getting ready

In ArcPy, feature classes have associated geometry objects including `Polygon`, `Polyline`, `PointGeometry`, and `MultiPoint` that you can access from your cursors. These objects refer to the `shape` field in the attribute table for a feature class. You can read the geometries of each feature in a feature class through these objects.

Polyline and polygon feature classes are composed of features containing multiple parts. You can use the `partCount` property to return the number of parts per feature and then use `getPart()` for each part in the feature to loop through each of the points and pull out the coordinate information. Point feature classes are composed of one `PointGeometry` object per feature that contains the coordinate information for each point.

In this recipe, you will use a `SearchCursor` and `Polygon` object to read the geometry of a polygon feature class.

How to do it...

Follow these steps to learn how to read the geometric information from each feature in a feature class:

1. Open IDLE and create a new script.
2. Save the script to `c:\ArcpyBook\Ch9\ReadGeometry.py`.
3. Import the `arcpy` module:
   ```
   import arcpy
   ```
4. Set the input feature class to the `SchoolDistricts` polygon feature class:
   ```
   infc = "c:/ArcpyBook/data/CityOfSanAntonio/SchoolDistricts"
   ```
5. Create a `SearchCursor` object with the input feature class, and return the `ObjectID` and `Shape` fields. The `Shape` field contains the geometry for each feature. The cursor will be created inside a `for` loop that we'll use to iterate through all the features in the feature class:
   ```
   for row in arcpy.da.SearchCursor(infc, ["OID@", "SHAPE@"]):
   Print the object id of each feature.
   # Print the current ID
     print("Feature {0}:".format(row[0]))
     partnum = 0
   ```
6. Use a `for` loop to loop through each part of the feature:
   ```
     # Step through each part of the feature
     for part in row[1]:
       # Print the part number
       print("Part {0}:".format(partnum))
   ```
7. Use a `for` loop to loop through each vertex in each part and print the x and y coordinates:
   ```
       # Step through each vertex in the feature
       #
       for pnt in part:
   ```

```
            if pnt:
              # Print x,y coordinates of current point
              #
              print("{0}, {1}".format(pnt.X, pnt.Y))
            else:
              # If pnt is None, this represents an interior ring
              #
              print("Interior Ring:")
          partnum += 1
```

8. Save and run the script. You should see the following output as the script writes the information for each feature, each part of the feature, and the x and y coordinates that define each part:

```
Feature 1:
Part 0:
-98.492224986, 29.380866971
-98.489300049, 29.379610054
-98.486967023, 29.378995028
-98.48503096, 29.376808947
-98.481447988, 29.375624018
-98.478799041, 29.374304981
```

How it works...

We initially created a `SearchCursor` object to hold the contents of our feature class. After this, we looped through each row in the cursor using a `for` loop. For each row, we looped through all the parts of the geometry. Remember that polyline and polygon features are composed of two or more parts. For each part, we also returned the points associated with each part and we printed the `x` and `y` coordinates of each point.

10
Listing and Describing GIS Data

In this chapter, we will cover the following recipes:

- Getting a list of feature classes in a workspace
- Restricting the list of objects returned with a wildcard
- Restricting the list of objects returned with a feature type
- Getting a list of fields in a feature class or table
- Using the Describe() function to return descriptive information about a feature class
- Using the Describe() function to return descriptive information about an image
- Returning workspace information with the Describe() function

Introduction

Python provides you the ability to batch process data through scripting. This helps you to automate workflows and increases the efficiency of your data processing. For example, you may need to iterate through all the datasets on a disk and perform a specific action on each dataset. Your first step is often to perform an initial gathering of data before proceeding to the main body of the geoprocessing task. This initial data gathering is often accomplished through the use of one or more **List** methods found in ArcPy. These lists are returned as true Python list objects. These list objects can then be iterated over for further processing. ArcPy provides a number of functions that can be used to generate lists of data. These methods work on many different types of GIS data. In this chapter, we will examine the many functions provided by ArcPy for creating lists of data. In *Chapter 3*, *Managing Map Documents and Layers*, we also covered a number of list functions. However, these functions were related to working with the `arcpy.mapping` module, and specifically for working with map documents and layers. The list functions we cover in this chapter reside directly in `arcpy` and are more generic in nature.

We will also cover the `Describe()` function for returning a dynamic object that will contain property groups. These dynamically generated `Describe` objects will contain property groups that are dependent upon the type of data that has been described. For instance, when the `Describe()` function is run against a feature class, the properties specific to a feature class will be returned. In addition, all data, regardless of the datatype, acquires a set of generic properties that we'll discuss.

Getting a list of feature classes in a workspace

As with all the list functions that we'll examine in this chapter, getting a list of feature classes in a workspace is often the first step in a multi-step process that your script will execute. For example, you might want to add a new field to all the feature classes in a file geodatabase. To do this, you'd first need to get a list of all the feature classes in the workspace.

Getting ready

ArcPy provides functions for getting lists of fields, indexes, datasets, feature classes, files, rasters, tables, and more. The `ListFeatureClasses()` function can be used to generate a list of all feature classes in a workspace. `ListFeatureClasses()` has three optional arguments that can be passed into the function and which will serve to limit the returned list. The first optional argument is a **wildcard** that can be used to limit the feature classes returned based on name. The second optional argument can be used to limit the feature classes returned based on the datatype (point, line, polygon, and so on). The third optional parameter limits the returned feature classes by a feature dataset. In this recipe, we will return all feature classes from a workspace.

How to do it...

Follow these steps to learn how to use the `ListFeatureClasses()` function to retrieve a list of the feature classes in a workspace:

1. Open IDLE and create a new script window.
2. Save the script as `c:\ArcpyBook\Ch10\ListFeatureClasses.py`.
3. Import the `arcpy` module:

   ```
   import arcpy
   ```

4. Set the workspace:

   ```
   arcpy.env.workspace = "C:/ArcpyBook/data/CityOfSanAntonio.gdb"
   ```

   You must always remember to set the workspace using the environment settings before calling any list function. Otherwise, the list function will not know which dataset the list should be pulled from.

5. Call the `ListFeatureClasses()` function and assign the results to a variable called `fcList`:

   ```
   fcList = arcpy.ListFeatureClasses()
   ```

6. Loop through each of the feature classes in `fcList` and print them to the screen:

   ```
   for fc in fcList:
     print fc
   ```

7. Save and run the script. You should see the following output:

   ```
   Crimes2009
   CityBoundaries
   CrimesBySchoolDistrict
   SchoolDistricts
   BexarCountyBoundaries
   Texas_Counties_LowRes
   Burglary
   ```

How it works...

Before calling any list function, you need to set the workspace environment setting, which sets the current workspace from which you will generate the list. The `ListFeatureClasses()` function can accept three optional parameters that will limit the feature classes that are returned. Most of the other list functions work in the same way. However, in this case, we've called the `ListFeatureClasses()` function without passing in any parameters. What this will do is return all the feature classes in the current workspace within a Python list object, which will then be iterated with a `for` loop. Each feature class returned in the list is represented as a string containing the name of the feature class.

There's more...

Instead of returning a list of feature classes in a workspace, you may need to get a list of tables. The `ListTables()` function returns a list of standalone tables in a workspace. This list can be filtered by name or table type. Table types can include `dBase`, `INFO`, and `ALL`. All the values in the list are of the `string` datatype and contain the table names.

Restricting the list of feature classes returned with a wildcard

By default, the `ListFeatureClasses()` function will return all of the feature classes in a workspace. You will often want to restrict this list in some way. Three optional parameters can be passed into `ListFeatureClasses()` to restrict the feature classes returned. All are optional. The first parameter is a wildcard used to restrict the returned list based on some combination of characters. Other parameters that can be used to restrict the list to include a datatype and a feature dataset.

Getting ready

The list of feature classes returned by the `ListFeatureClasses()` function can be restricted through the use of a wildcard passed in as the first parameter. The wildcard is used to restrict the contents of your list based on a name. For example, you may want to return only a list of feature classes that start with the letter `B`. To accomplish this, you use a combination of an asterisk with any number of characters. The following code example shows the use of a wildcard to restrict the contents of a list:

```
fcs = arcpy.ListFeatureClasses("B*")
```

In this recipe, you will learn how to restrict the list of feature classes returned through the use of a wildcard.

How to do it...

Follow these steps to learn how to restrict the list of feature classes returned by the `ListFeatureClasses()` function through the use of a wildcard that is passed in as the first parameter:

1. Open IDLE and the `c:\ArcpyBook\Ch10\ListFeatureClasses.py` script.
2. Add a wildcard that restricts the list of feature classes returned to only those feature classes that start with the letter `C`:

    ```
    fcs = arcpy.ListFeatureClasses("C*")
    ```

3. Save and run the script to see the following output:

    ```
    Crimes2009
    CityBoundaries
    CrimesBySchoolDistrict
    ```

How it works...

The `ListFeatureClasses()` function can accept three optional parameters including a wildcard that will restrict the list of feature classes by a name. In this case, we used the wildcard character (`*`) to restrict the list of feature classes returned to only those that begin with the letter `C`.

Restricting the list of feature classes returned with a feature type.

In addition to using a wildcard to restrict the feature classes returned by `ListFeatureClasses()`, you can also filter by feature type.

Getting ready

In addition to using a wildcard to restrict the list returned by the `ListFeatureClasses()` function, a type restriction can also be applied in conjunction with the wildcard or by itself. For example, the following code sample shows the two being used together to restrict the list returned to only `polygon` feature classes beginning with the letter `B`. In this recipe, you will restrict the feature classes returned through the use of a feature type parameter along with a wildcard:

```
fcs = arcpy.ListFeatureClasses("B*", "Polygon")
```

How to do it...

Follow these steps to learn how to restrict the list of feature classes returned by the `ListFeatureClasses()` function by feature type:

1. Open IDLE and the `c:\ArcpyBook\Ch10\ListFeatureClasses.py` script.
2. Add a second parameter to the `ListFeatureClasses()` function that restricts the feature classes returned to only those that start with the letter `C` and are of type `polygon`:

   ```
   fcs = arcpy.ListFeatureClasses("C*","Polygon")
   ```

3. Save and run the script to see the following output:

   ```
   CityBoundaries
   CrimesBySchoolDistrict
   ```

How it works...

The second, optional parameter that can be passed into the `ListFeatureClasses()` function can be used to limit the results by feature type. In this recipe, we have limited the feature classes to only polygon features. Other valid feature types include point and polyline, region.

There's more...

The third, optional parameter that can be passed into the `ListFeatureClasses()` function is a feature dataset name. This will filter the list to return only the featured classes within a particular feature dataset. When this optional parameter is not included in the call to `ListFeatureClasses()`, only standalone feature classes from the current workspace will be returned.

Getting a list of fields in a feature class or table

Feature classes and tables contain one or more columns of attribute information. You can get a list of the fields in a feature class through the `ListFields()` function.

Getting ready

The `ListFields()` function returns a list containing individual `Field` objects for each field in a feature class or table. Some functions, such as `ListFields()` and `ListIndexes()`, require an input dataset to operate on. You can use a wildcard or field type to constrain the list that is returned. Each `Field` object contains various read-only properties, including `Name`, `AliasName`, `Type`, `Length`, and so on.

How to do it...

Follow these steps to learn how to return a list of fields in a feature class:

1. Open IDLE and create a new script window.
2. Save the script as `c:\ArcpyBook\Ch10\ListOfFields.py`.
3. Import the `arcpy` module.
   ```
   import arcpy
   ```
4. Set the workspace:
   ```
   arcpy.env.workspace = "C:/ArcpyBook/data/CityOfSanAntonio.gdb"
   ```
5. Call the `ListFields()` method on the `Burglary` feature class inside a `try` block:
   ```
   try:
     fieldList = arcpy.ListFields("Burglary")
   ```
6. Loop through each of the fields in the list of fields and print the name, type, and length. Make sure you indent as needed:
   ```
     for fld in fieldList:
       print "%s is a type of %s with a length of %i" % (fld.name, 
   fld.type, fld.length)
   ```
7. Add the `except` block:
   ```
   except Exception e:
     print e.message();
   ```
8. The entire script should appear as follows:
   ```
   import arcpy

   arcpy.env.workspace = "C:/ArcpyBook/data/CityOfSanAntonio.gdb"
   try:
    fieldList = arcpy.ListFields("Burglary")
     for fld in fieldList:
   ```

```
    print "%s is a type of %s with a length of %i" % (fld.name,
fld.type, fld.length)
except Exception e:
  print e.message()
```

9. Save and run the script. You should see the following output:

```
OBJECTID is a type of OID with a length of 4
Shape is a type of Geometry with a length of 0
CASE is a type of String with a length of 11
LOCATION is a type of String with a length of 40
DIST is a type of String with a length of 6
SVCAREA is a type of String with a length of 7
SPLITDT is a type of Date with a length of 8
SPLITTM is a type of Date with a length of 8
HR is a type of String with a length of 3
DOW is a type of String with a length of 3
SHIFT is a type of String with a length of 1
OFFCODE is a type of String with a length of 10
OFFDESC is a type of String with a length of 50
ARCCODE is a type of String with a length of 10
ARCCODE2 is a type of String with a length of 10
ARCTYPE is a type of String with a length of 10
XNAD83 is a type of Double with a length of 8
YNAD83 is a type of Double with a length of 8
```

How it works...

The `ListFields()` function returns a list of fields from a feature class or a table. This function accepts one required parameter, which is a reference to the feature class or table the function should be executed against. You can limit the fields returned by using a wild card or a field type. In this recipe, we only specified a feature class, which indicates that all fields will be returned. For each field returned, we printed the name, field type, and field length. As I mentioned earlier when discussing the `ListFeatureClasses()` function, `ListFields()` and all the other list functions are often called as the first step in a multi-step process within a script. For example, you might want to update the population statistics contained within a population field for a census tracts feature class. To do this, you could get a list of all the fields within a feature class, loop through this list looking for a specific field name that contains the population information, and then update the population information for each row. Alternatively, the `ListFields()` function accepts a wildcard as one of its parameters. So, if you know the name of the population field ahead of time, pass this in as the wildcard and only a single field will be returned.

Using the Describe() function to return descriptive information about a feature class

All datasets contain information that is descriptive in nature. For example, a feature class has a name, shape type, spatial reference, and so on. This information can be valuable to your scripts when you seek specific information before continuing with further processing in the script. For example, you might want to perform a buffer only on polyline feature classes instead of points or polygons. Using the `Describe()` function you can obtain basic descriptive information about any dataset. You can think of this information as metadata.

Getting ready

The `Describe()` function provides you with the ability to get basic information about datasets. These datasets could include feature classes, tables, ArcInfo coverage, layer files, workspaces, rasters, and others. A `Describe` object is returned, which contains specific properties based on the datatype being described. Properties on the `Describe` object are organized into property groups, and all datasets fall into at least one property group. For example, performing a `Describe()` against a geodatabase would return the **GDB** `FeatureClass`, `Table`, and `Dataset` property groups. Each of these property groups contains specific properties that can be examined.

The `Describe()` function accepts a string parameter, which is a pointer to a data source. In the following code example, we pass in a feature class contained within a file geodatabase. The function returns a `Describe` object that contains a set of dynamic properties called property groups. We can then access these various properties, as we have done in this case, by simply printing the properties using the `print` function:

```
arcpy.env.workspace = "c:/ArcpyBook/Ch10/CityOfSanAntonio.gdb"
desc = arcpy.Describe("Schools")
print "The feature type is: " + desc.featureType
The feature type is: Simple
print "The shape type is: " + desc.shapeType
The shape type is: Polygon
print "The name is: " + desc.name
The name is: Schools
print "The path to the data is: " + desc.path
The path to the data is: c:/ArcpyBook/Ch10/CityOfSanAntonio.gdb
```

All datasets, irrespective of their type, contain a default set of properties located on the `Describe` object. These are read-only properties. Some of the more commonly used properties include `dataType`, `catalogPath`, `name`, `path`, and `file`.

In this recipe, you will write a script that obtains descriptive information about a feature class using the `Describe()` function.

How to do it...

Follow these steps to learn how to obtain descriptive information about a feature class:

1. Open IDLE and create a new script window.
2. Save the script as `c:\ArcpyBook\Ch10\DescribeFeatureClass.py`.
3. Import the `arcpy` module.
4. import arcpy Set the workspace:

   ```
   arcpy.env.workspace = "C:/ArcpyBook/data/CityOfSanAntonio.gdb"
   ```

5. Start a `try` block:

   ```
   try:
   ```

6. Call the `Describe()` function on the `Burglary` feature class and print the shape type:

   ```
   descFC = arcpy.Describe("Burglary")
   print "The shape type is: " + descFC.ShapeType
   ```

7. Get a list of fields in the feature class and print the name, type, and length of each:

   ```
   flds = descFC.fields
   for fld in flds:
     print "Field: " + fld.name
     print "Type: " + fld.type
     print "Length: " + str(fld.length)
   ```

8. Get the geographic extent of the feature class and print the coordinates that define the extent:

   ```
   ext = descFC.extent
   print "XMin: %f" % (ext.XMin)
   print "YMin: %f" % (ext.YMin)
   print "XMax: %f" % (ext.XMax)
   print "YMax: %f" % (ext.YMax)
   ```

9. Add the `except` block:

```
except Exception e:
  print e.message()
```

10. The entire script should appear as follows:

```
import arcpy
arcpy.env.workspace = "c:/ArcpyBook/data/CityOfSanAntonio.gdb"
try:
  descFC = arcpy.Describe("Burglary")
  print "The shape type is: " + descFC.ShapeType
  flds = descFC.fields
  for fld in flds:
    print "Field: " + fld.name
    print "Type: " + fld.type
    print "Length: " + str(fld.length)
  ext = descFC.extent
  print "XMin: %f" % (ext.XMin)
  print "YMin: %f" % (ext.YMin)
  print "XMax: %f" % (ext.XMax)
  print "YMax: %f" % (ext.YMax)
except:
  print arcpy.GetMessages()
```

11. Save and run the script. You should see the following output:

```
The shape type is: Point
Field: OBJECTID
Type: OID
Length: 4
Field: Shape
Type: Geometry
Length: 0
Field: CASE
Type: String
Length: 11
Field: LOCATION
Type: String
Length: 40
.....
.....
XMin: -103.518030
YMin: -6.145758
XMax: -98.243208
YMax: 29.676404
```

How it works...

Executing a `Describe()` function against a feature class, which we have done in this script, returns a `FeatureClass` property group along with access to the `Table` and `Dataset` property groups. In addition to returning a `FeatureClass` property group, you also have access to a `Table` properties group.

The `Table` property group is important primarily because it gives you access to the fields in a standalone table or feature class. You can also access any indexes on the table or feature class through this property group. The `Fields` property on table properties returns a Python list containing one `Field` object for each field in the feature class. Each field has a number of read-only properties including the `name`, `alias`, `length`, `type`, `scale`, `precision`, and so on. The most obviously useful properties are `name` and `type`. In this script, we printed the field name, type, and length. Note the use of a Python `for` loop to process each field in the Python list.

Finally, we printed out the geographic extent of the layer through the use of the `Extent` object, returned by the `extent` property on the `Dataset` property group. The `Dataset` property group contains a number of useful properties. Perhaps, the most used properties include `extent` and `spatialReference`, as many geoprocessing tools and scripts require this information at some point during execution. You can also obtain the `datasetType` and versioning information along with several other properties.

Using the Describe() function to return descriptive information about an image

Raster files also contain descriptive information, which can be returned by the `Describe()` function.

Getting ready

A raster dataset can also be described through the use of the `Describe()` function.
In this recipe, you will describe a raster dataset by returning its extent and spatial reference. The `Describe()` function contains a reference to the general purpose `Dataset` properties group as well, which contains a reference to the `SpatialReference` object for the dataset. The `SpatialReference` object can then be used to get detailed spatial reference information for the dataset.

How to do it...

Follow these steps to learn how to obtain descriptive information about a raster image file:

1. Open IDLE and create a new script window.
2. Save the script as `c:\ArcpyBook\Ch10\DescribeRaster.py`.
3. Import the `arcpy` module:

   ```
   import arcpy
   ```

4. Set the workspace:

   ```
   arcpy.env.workspace = "C:/ArcpyBook/data"
   ```

5. Start a `try` block:

   ```
   try:
   ```

6. Call the `Describe()` function on a raster dataset making sure you indent the next few lines of code inside the `try` statement:

   ```
   descRaster = arcpy.Describe("AUSTIN_EAST_NW.sid")
   ```

7. Get the extent of the raster dataset and print it:

   ```
   ext = descRaster.extent
   print "XMin: %f" % (ext.XMin)
   print "YMin: %f" % (ext.YMin)
   print "XMax: %f" % (ext.XMax)
   print "YMax: %f" % (ext.YMax)
   ```

8. Get a reference to the `SpatialReference` object and print it:

   ```
   sr = descRaster.SpatialReference
   print sr.name
   print sr.type
   ```

9. Add the `except` block:

   ```
   except Exception e:
     print e.message()
   ```

10. The entire script should appear as follows:

    ```
    import arcpy
    arcpy.env.workspace = "c:/ArcpyBook/data"
    try:
      descRaster = arcpy.Describe("AUSTIN_EAST_NW.sid")
      ext = descRaster.extent
    ```

```
        print "XMin: %f" % (ext.XMin)
        print "YMin: %f" % (ext.YMin)
        print "XMax: %f" % (ext.XMax)
        print "YMax: %f" % (ext.YMax)

        sr = descRaster.SpatialReference
        print sr.name
        print sr.type
      except:
        print arcpy.GetMessages()
```

11. Save raster and run the script. You should see the following output:

```
XMin: 3111134.862457
YMin: 10086853.262238
XMax: 3131385.723907
YMax: 10110047.019228
NAD83_Texas_Central
Projected
```

How it works...

This recipe is very similar to the previous one. The difference is that we're using the `Describe()` function against a raster dataset instead of against a vector feature class. In both cases, we've returned the geographic extent of the datasets using the `extent` object. However, in the script we've also obtained the `SpatialReference` object for the raster dataset and printed the information about this object including the name and type.

Returning workspace information with the Describe() function

There are several different types of geodatabases that can be used with ArcGIS, including personal, file, and enterprise. As we saw in *Chapter 8*, *Querying and Selecting Data*, the construction of queries will vary depending upon the type of geodatabase a dataset resides within. Your scripts may or may not know the geodatabase type ahead of time. To make your scripts more robust in the case of queries, you could use the `Describe()` function against a workspace to capture this information and construct your query accordingly.

Getting ready

The `Workspace` property group provides information about a workspace (such as a folder, personal or file geodatabase, or enterprise geodatabase). These properties are particularly helpful when obtaining information about an ArcSDE connection. Information that can be obtained through this property group includes the connection information when the workspace is an ArcSDE workspace, domains associated with the geodatabase, and the workspace type, which can be `FileSystem`, `LocalDatabase`, or `RemoteDatabase`. `LocalDatabase` refers to personal or file geodatabases, while `RemoteDatabase` refers to an `ArcSDE` geodatabase. In this recipe, you'll use the `Workspace` property group to obtain information about a file geodatabase.

How to do it...

Follow these steps to learn how to obtain descriptive information about a workspace:

1. Open IDLE and create a new script window.
2. Save the script as `c:\ArcpyBook\Ch10\DescribeWorkspace.py`.
3. Import the `arcpy` module:

   ```
   import arcpy
   ```

4. Start a `try` block:

   ```
   try:
   ```

5. Call the `Describe()` function on the `CityOfSanAntonio` geodatabase and make sure to indent this statement inside the try statement. The two print statements below should also be indented.

   ```
   descRaster = arcpy.Describe("c:/ArcpyBook/data/CityOfSanAntonio.
   gdb")
   ```

6. Print the workspace type:

   ```
   print descWorkspace.workspaceType
   ```

7. Print the detailed workspace information:

   ```
   print descWorkspace.workspaceFactoryProgID
   ```

8. Add the `except` block:

   ```
   except Exception e:
     print e.message()
   ```

9. Save and run the script. You should see the following output:

   ```
   LocalDatabase
   esriDataSourcesGDB.FileGDBWorkspaceFactory.1
   ```

How it works...

The `workspaceType` property returns one of three values: `FileSystem`, `LocalDatabase`, or `RemoteDatabase`. The `localDatabase` value indicates that you're working with either a personal or file geodatabase. However, it isn't any more specific. To get the specific geodatabase, you can retrieve the `workspaceFactoryProgID` property, which will indicate the type of geodatabase. In this case, it's a file geodatabase.

11
Customizing the ArcGIS Interface with Add-Ins

In this chapter, we will cover the following recipes:

- Downloading and installing the Python Add-In wizard
- Creating a button add-in
- Installing and testing an add-in
- Creating a tool add-in

Introduction

In this chapter, we're going to cover the creation, testing, editing, and sharing of add-ins created with Python. **Add-ins** provide a way of adding user interface items to ArcGIS Desktop through a modular code base designed to perform specific actions. Interface components can include buttons, tools, toolbars, menus, combo boxes, tool palettes, and application extensions. The add-in concept was first introduced in ArcGIS Desktop 10.0 and could be created with .NET or Java. However, with the release of ArcGIS 10.1, add-ins can now be created with Python. Add-ins are created using Python scripts and an XML file that defines how the user interface should appear.

Add-ins provide an easy way to distribute user interface customizations to end users. No installation programs are necessary. A single compressed file with a file extension of `.esriaddin` is copied to a well-known folder, and ArcGIS Desktop handles the rest. To simplify the development even further, a Python Add-In wizard has been provided by Esri. You can download the wizard from the Esri website. We'll do that in our first recipe in this chapter.

There are a number of add-in types that can be created. Buttons and tools are the simplest type of add-ins that you can create. Buttons simply execute business logic when clicked. Tools are similar to buttons but require interaction with the map before the business logic is executed. Combo boxes provide a list of choices for the user to select from.

There are also a number of container objects including menus, toolbars, tool palettes, and application extensions. Menus act as a container for buttons or other menus. Toolbars are a container for buttons, tools, combo boxes, tool palettes, and menus. They are the most versatile of the add-in container types. Tool palettes also act as a container for tools, and need to be added to a toolbar before the tools will be exposed. Finally, application extensions are the most complex add-in type. This type of add-in coordinates activities between other components and is responsible for listening for and responding to various events, such as the addition or removal of a layer from a data frame.

Downloading and installing the Python Add-In wizard

Esri provides a tool that you can use to make the development of add-ins easier. The Python Add-In wizard can be downloaded from the Esri website and is a great resource for creating your add-ins.

Getting ready

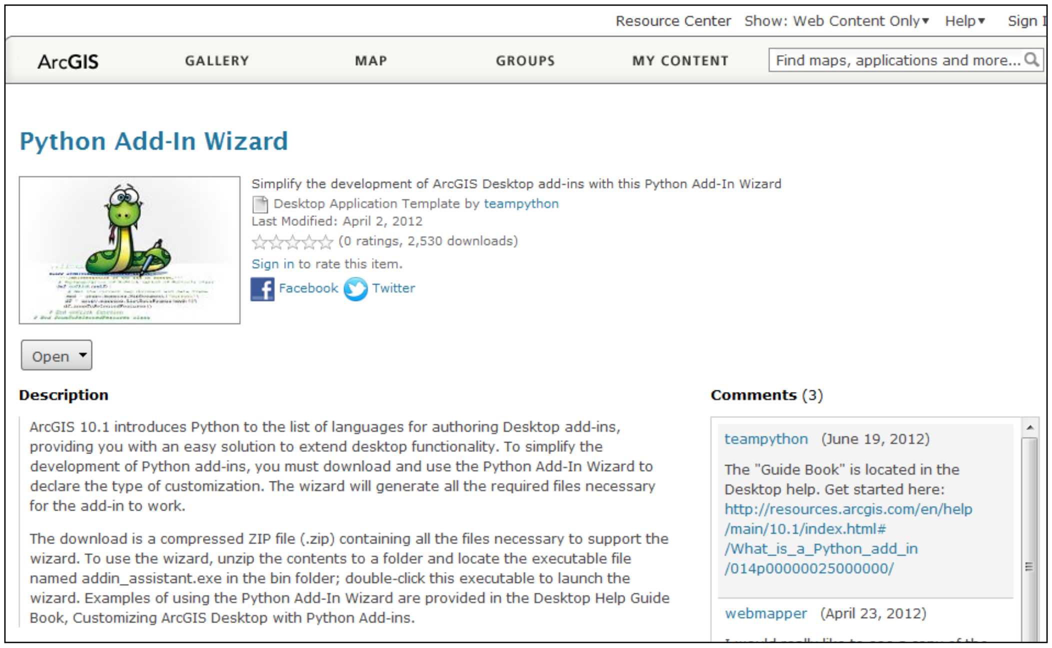
The Python Add-In Wizard, is a great resource for creating the necessary files for an add-in. It generates the required files for the add-ins from a visual interface. In this recipe, you will download and install the Python Add-In wizard.

How to do it...

Follow these steps to learn how to download and install the Python Add-in wizard:

1. Open a web browser and navigate to: `http://www.arcgis.com/home/item.html?id=5f3aefe77f6b4f61ad3e4c62f30bff3b`.

You should see a web page similar to the following screenshot:


Resource Center Show: Web Content Only▾ Help▾ Sign
ArcGIS
GALLERY
MAP
GROUPS
MY CONTENT
Find maps, applications and more...
Open

Python Add-In Wizard

Simplify the development of ArcGIS Desktop add-ins with this Python Add-In Wizard
Desktop Application Template by teampython
Last Modified: April 2, 2012
(0 ratings, 2,530 downloads)
Sign in to rate this item.
Facebook Twitter

Description

ArcGIS 10.1 introduces Python to the list of languages for authoring Desktop add-ins, providing you with an easy solution to extend desktop functionality. To simplify the development of Python add-ins, you must download and use the Python Add-In Wizard to declare the type of customization. The wizard will generate all the required files necessary for the add-in to work.

The download is a compressed ZIP file (.zip) containing all the files necessary to support the wizard. To use the wizard, unzip the contents to a folder and locate the executable file named addin_assistant.exe in the bin folder; double-click this executable to launch the wizard. Examples of using the Python Add-In Wizard are provided in the Desktop Help Guide Book, Customizing ArcGIS Desktop with Python Add-ins.

Comments (3)

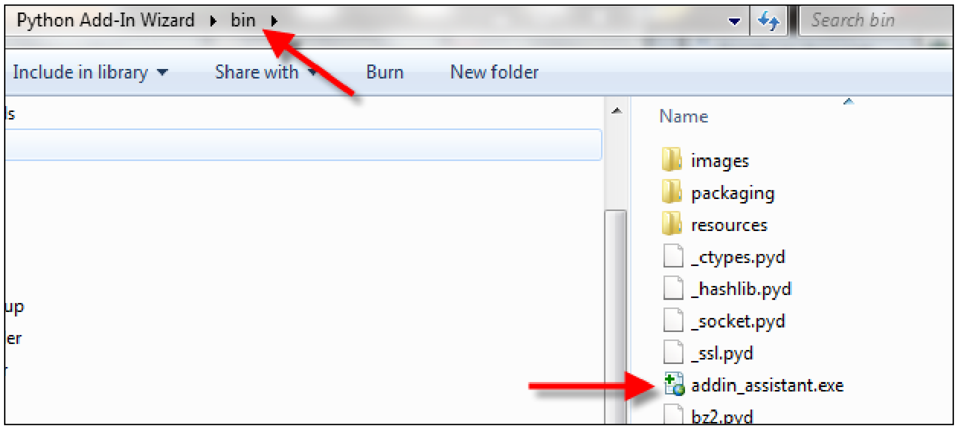
teampython (June 19, 2012)

The "Guide Book" is located in the Desktop help. Get started here: http://resources.arcgis.com/en/help/main/10.1/index.html#/What_is_a_Python_add_in/014p00000025000000/

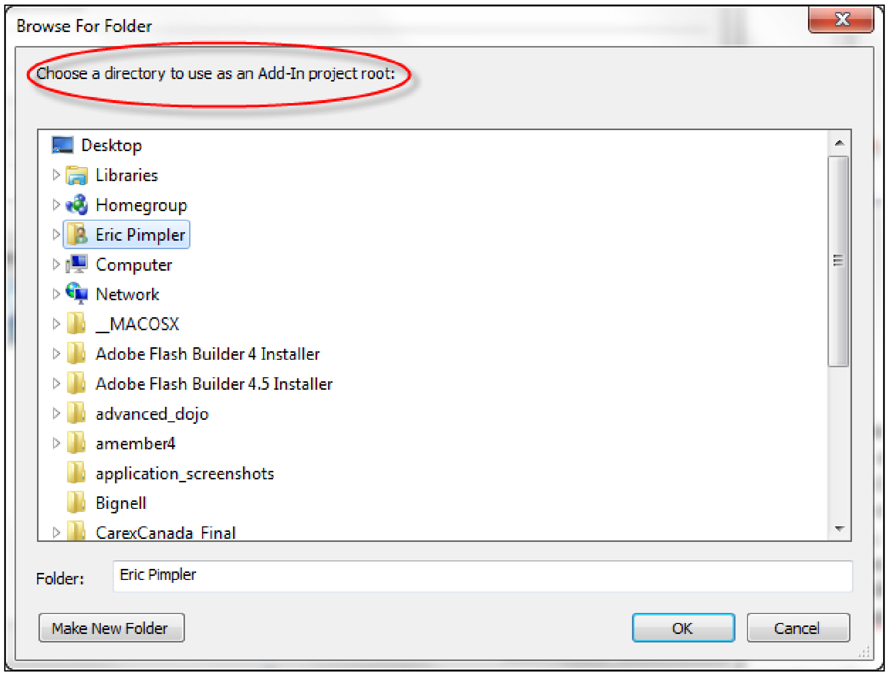
webmapper (April 23, 2012)

2. Click on the **Open** button to download the installer file.
3. Using Windows Explorer, create a new folder called `Python Add-In Wizard` somewhere on your computer. The name of the folder is irrelevant, but to keep things simple and easy to remember you should go with `Python Add-In Wizard` or something similar.
4. Unzip the file to this new folder There are many utilities that can be used to unzip a file. Each will differ slightly in how they are used but with WinZip, you should be able to right-click on the file and select **Extract**.

5. Open the `bin` folder that was unzipped and double-click on `addin_assistant.exe` to run the wizard. In the following screenshot, I have created a new folder called `Python Add-In Wizard` and unzipped the downloaded file. The `bin` folder was created and inside this folder is a file called `addin_assistant.exe`:



6. Double-clicking on `addin_assistant.exe` will prompt you to choose a directory to use as the add-in project root:



How it works...

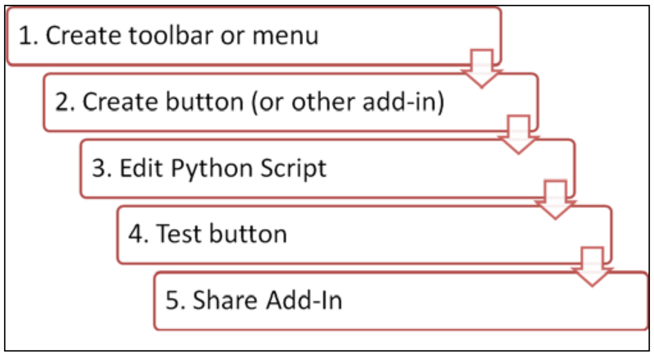
The Python Add-In wizard is a visual interface tool that you can use to create add-ins for ArcGIS Desktop. It greatly simplifies the process through a point-and-click tool. In the next recipe, you'll create basic ArcGIS Desktop add-in using the wizard.

Creating a button add-in

Button add-ins are the simplest types of add-ins and are also the most commonly used. With button add-ins, the functionality that you code in your script is executed each time the button is clicked.

Getting ready

Creating an add-in project is the first step in the creation of a new add-in. To create a project using the Python Add-In wizard, you select a working directory, enter various project settings, and click on the **Save** button. Creation of the add-in then follows a well-defined process, as illustrated in the following screenshot:

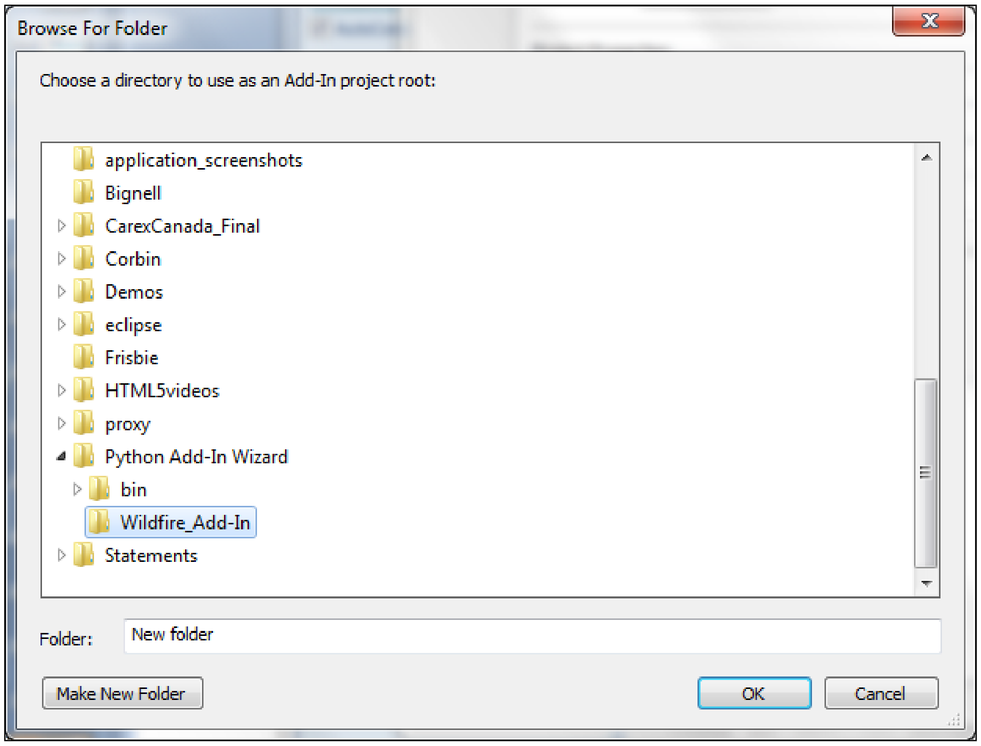


You must first create a container for the add-in and this can be either a toolbar or a menu. Next, create the button, tool, or another add-in that you want to add to the container. In this recipe, we'll just assume it's a button. Next, you need to edit the Python script associated with the button. You'll also want to test the button to make sure it works as expected. Finally, you can share the add-in with others. In this recipe, you'll learn how to use the Add-In wizard to create a button add-in for ArcGIS Desktop. The button add-in will run a custom script tool that you created in an earlier recipe, which loads wildfire data from a text file to a point feature class.

How to do it...

Follow these steps to learn how to create a button add-in:

1. Open the ArcGIS Python Add-In wizard by double-clicking on the `addin_assistant.exe` file located in the `bin` folder, where you extracted the wizard.
2. Create a new project folder called `Wildfire_Addin` and click **OK**:

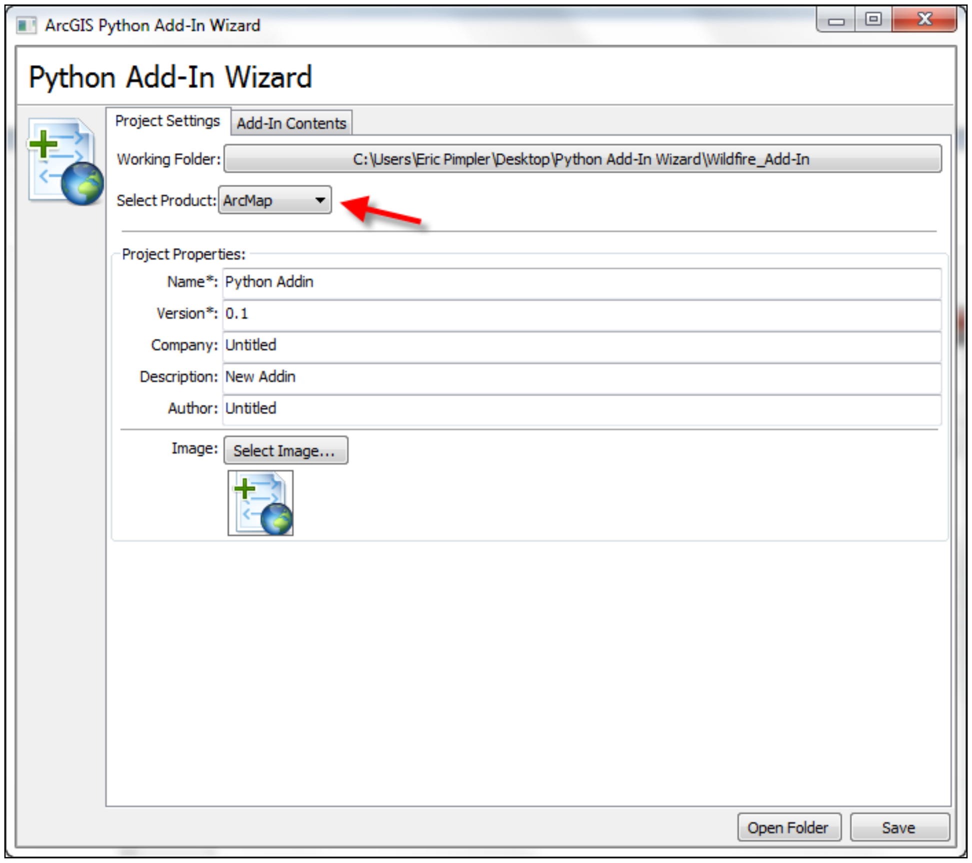


Browse For Folder

Choose a directory to use as an Add-In project root:

Folder: New folder

3. The **Project Settings** tab should be active initially and display the working directory you just created. By default, ArcMap should be the selected product, but you should verify that this is the case:



ArcGIS Python Add-In Wizard

Python Add-In Wizard

Working Folder: C:\Users\Eric Pimpler\Desktop\Python Add-In Wizard\Wildfire_Add-In

Select Product: ArcMap

Project Properties:

Name*: Python Addin

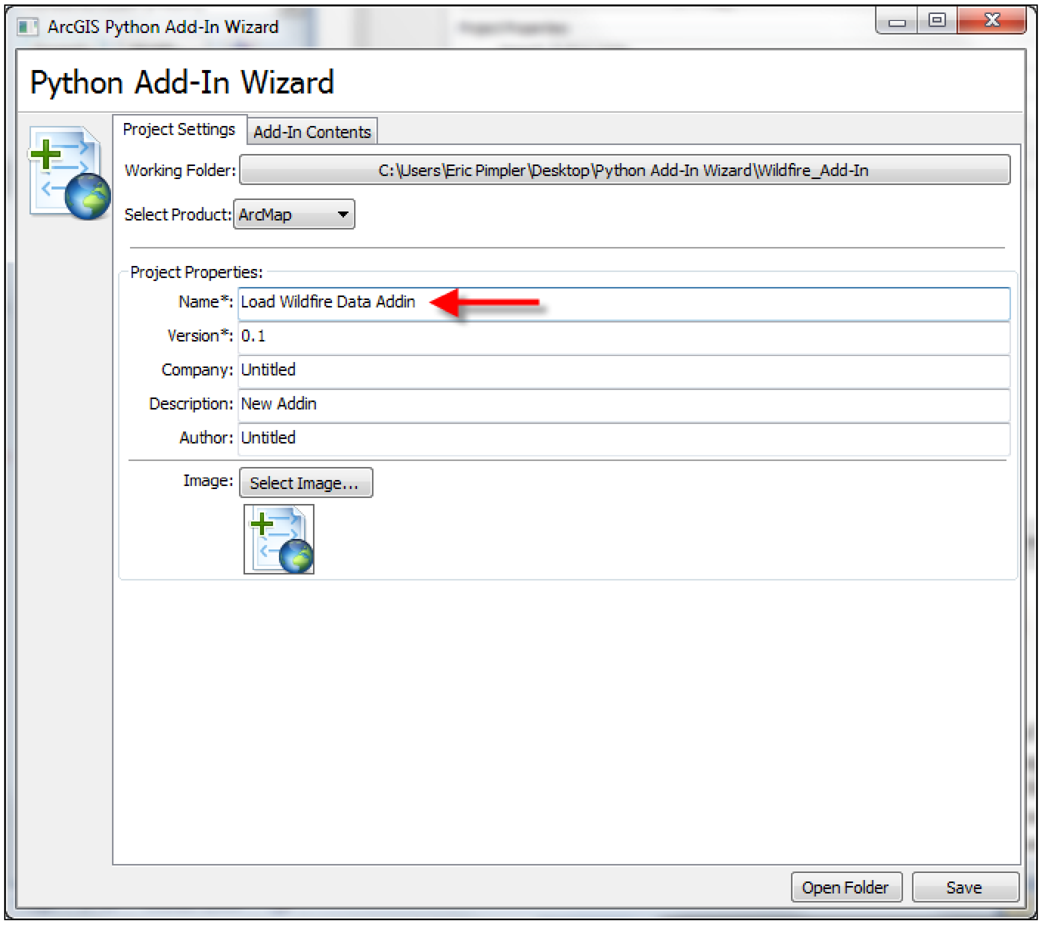
Version*: 0.1

Company: Untitled

Description: New Addin

Author: Untitled

4. Give your project a name. We'll call it `Load Wildfire Data Addin`:



Working Folder: C:\Users\Eric Pimpler\Desktop\Python Add-In Wizard\Wildfire_Add-In

Select Product: ArcMap

Project Properties:

Name*: Load Wildfire Data Addin

Version*: 0.1

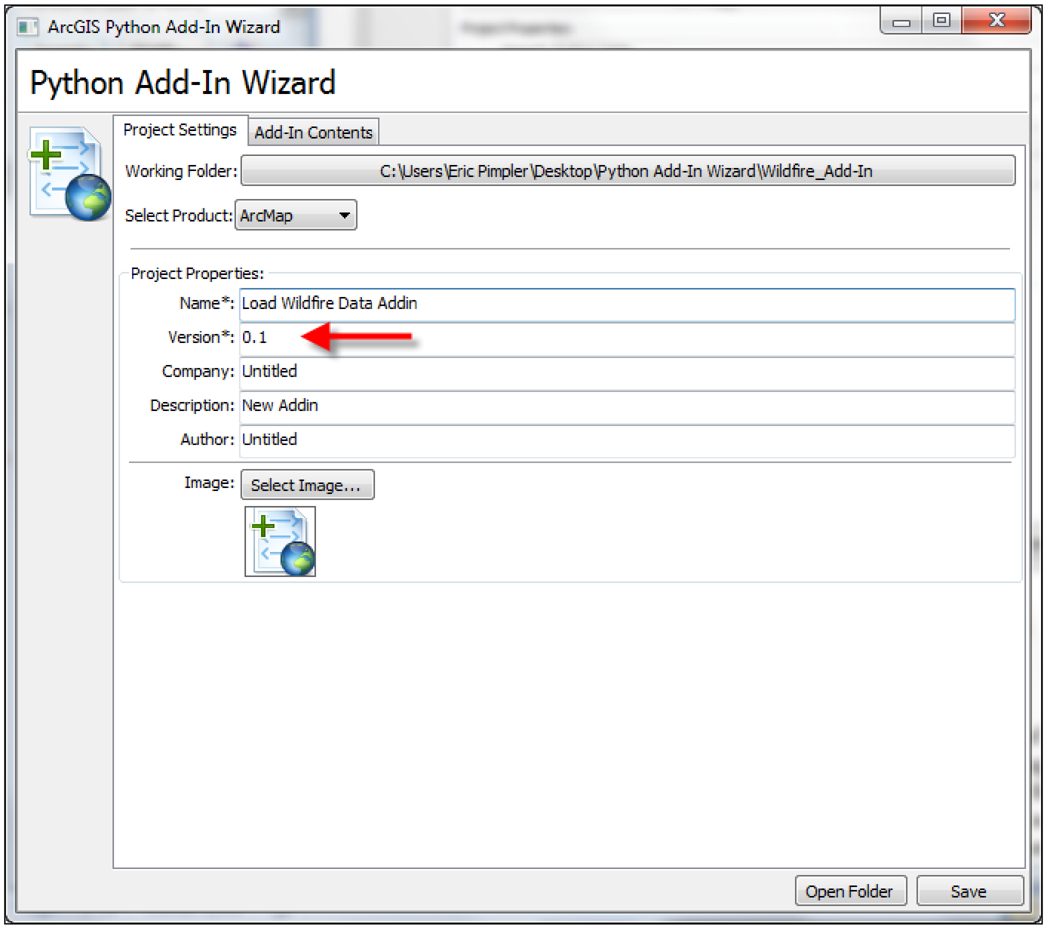
Company: Untitled

Description: New Addin

Author: Untitled

Image:

5. By default, the Version is 0.1. You can change this if you'd like. Version numbers should change as you update or make additions to your tool. This helps with tracking and sharing of your add-ins:


ArcGIS Python Add-In Wizard

Python Add-In Wizard

Working Folder: C:\Users\Eric Pimpler\Desktop\Python Add-In Wizard\Wildfire_Add-In

Select Product: ArcMap

Project Properties:

Name*: Load Wildfire Data Addin

Version*: 0.1

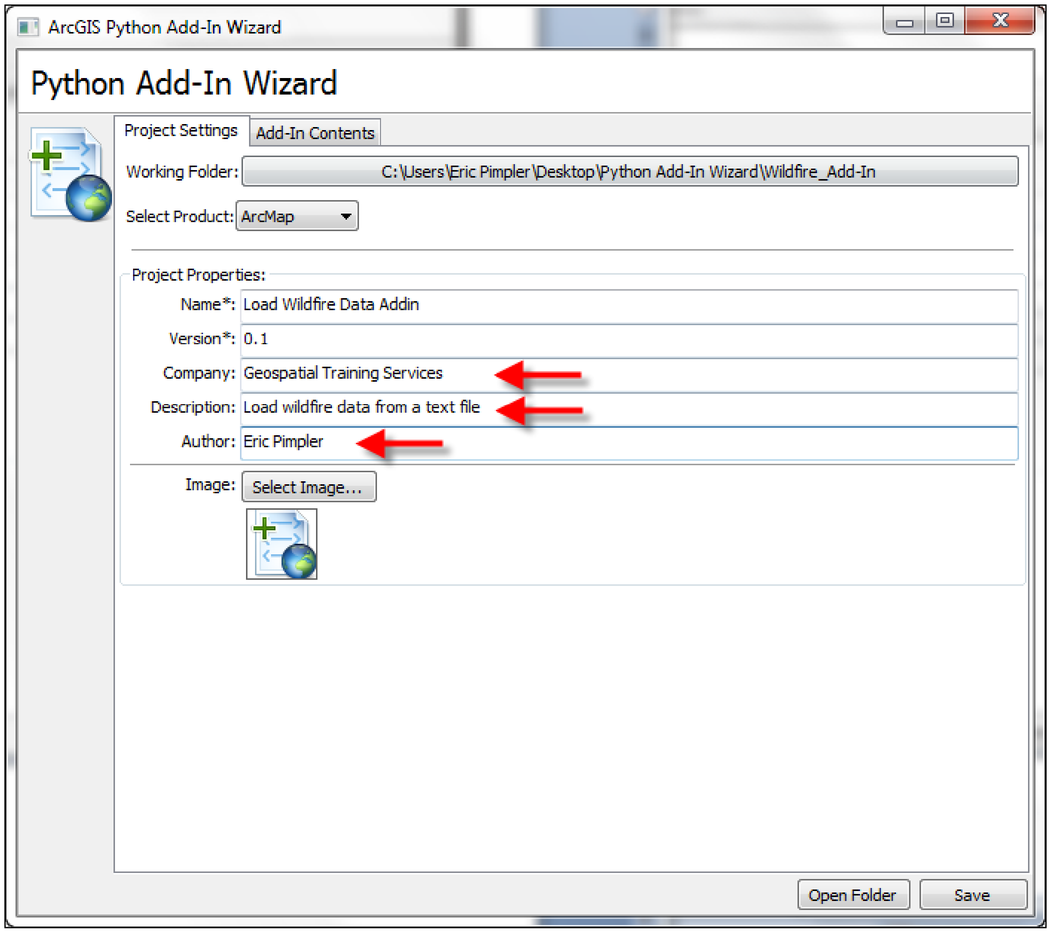
Company: Untitled

Description: New Addin

Author: Untitled

Image: Select Image...

6. The name and version properties are the only two required properties. It's a good practice to go ahead and add the company, description, and author information as shown in the following screenshot. Add your own information:



Working Folder: C:\Users\Eric Pimpler\Desktop\Python Add-In Wizard\Wildfire_Add-In

Select Product: ArcMap

Project Properties:

Name*: Load Wildfire Data Addin

Version*: 0.1

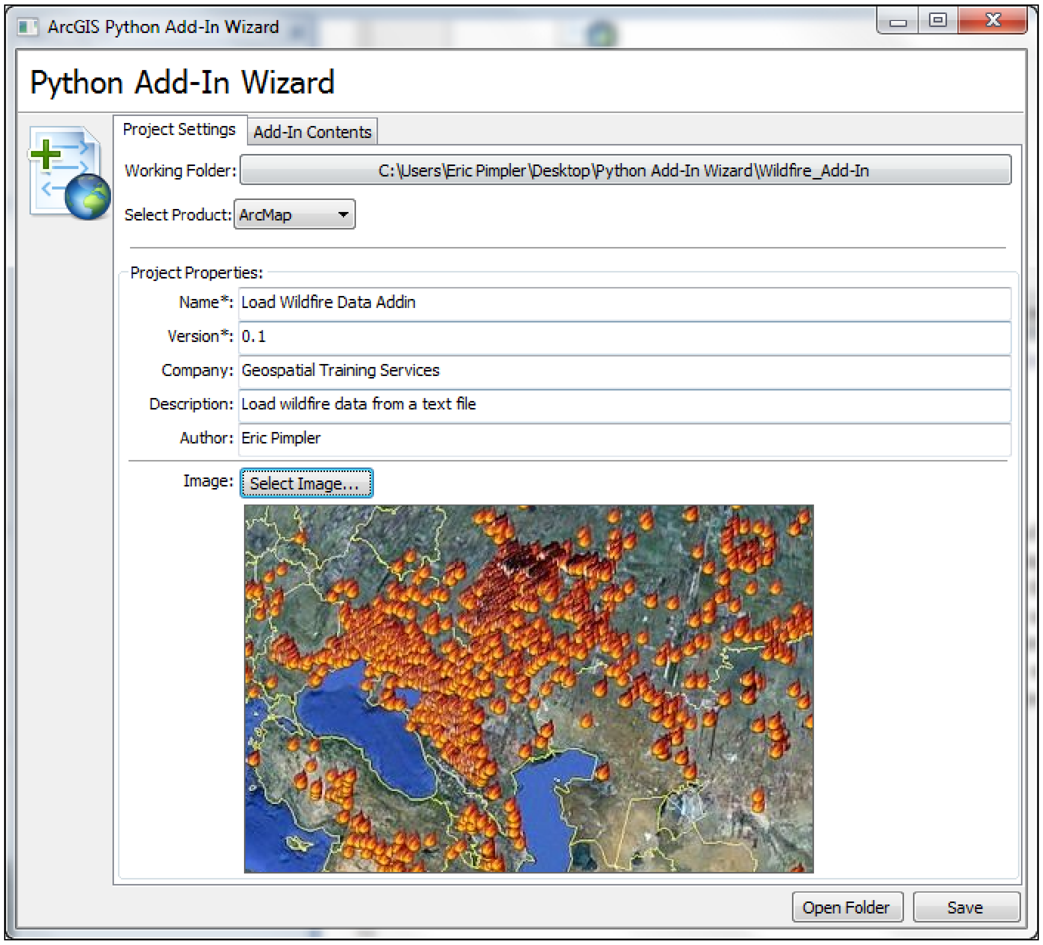
Company: Geospatial Training Services

Description: Load wildfire data from a text file

Author: Eric Pimpler

Image:

7. You may also wish to add an image for the add-in. A file called `wildfire.png` has been provided for this purpose in the `C:\ArcpyBook\Ch11` folder:



Working Folder: C:\Users\Eric Pimpler\Desktop\Python Add-In Wizard\Wildfire_Add-In

Select Product: ArcMap

Project Properties:

Name*: Load Wildfire Data Addin

Version*: 0.1

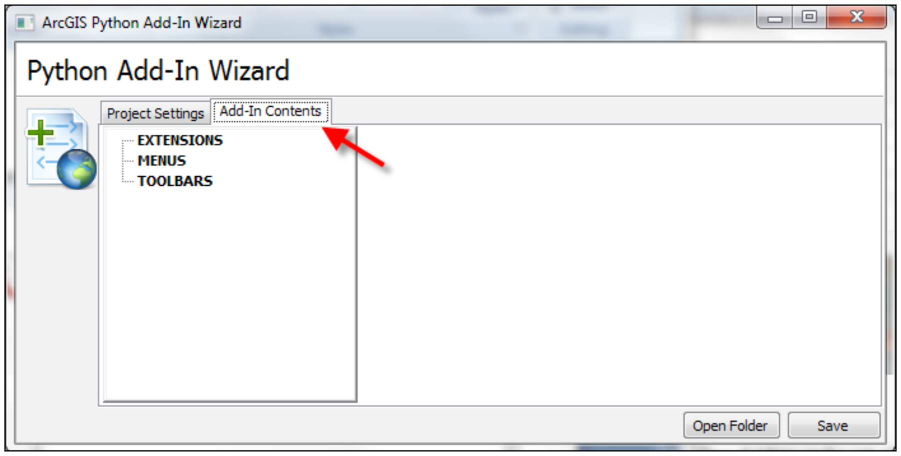
Company: Geospatial Training Services

Description: Load wildfire data from a text file

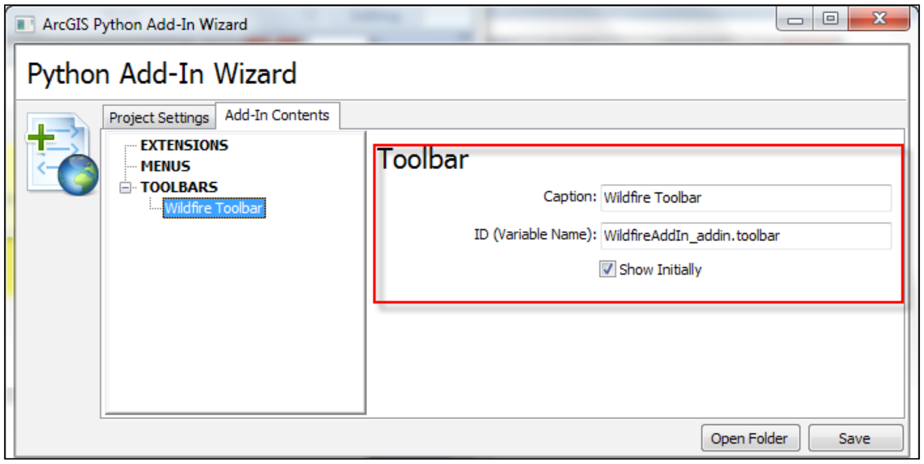
Author: Eric Pimpler

Image:

8. The **Add-In Contents** tab is used to define the various add-ins that can be created. In this step, we're going to create a toolbar to hold a single button add-in that runs a wildfire script, which imports `fires` from a text file into a feature class. Click on the **Add-In Contents** tab:



9. In the **Add-In Contents** tab, right-click on **Toolbars** and select **New Toolbar**. Give the toolbar a caption, accept the default ID, and make sure the **Show Initially** checkbox is selected:



Toolbar

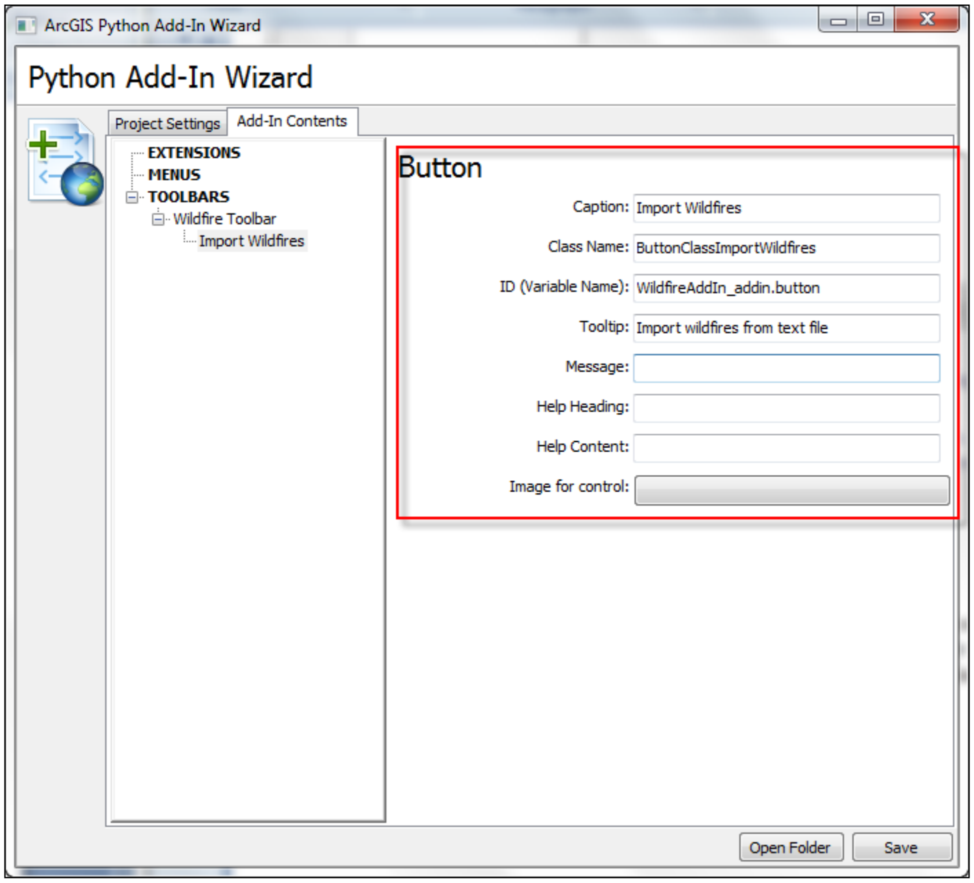
Caption: Wildfire Toolbar

ID (Variable Name): WildfireAddIn_addin.toolbar

☑ Show Initially

The **Toolbar** add-in, while it doesn't do a whole lot functionally, is very important because it acts as a container for other add-ins such as buttons, tools, combo boxes, tool palettes, and menus. Toolbars can be floating or docked. Creating a toolbar add-in is easy using the Python Add-In wizard.

10. Click on the **Save** button.
11. Now, we'll add a button by right-clicking on the new **Wildfire Toolbar** option and selecting **New Button**.
12. Fill in the button details including a caption, class name, ID, tooltip, and others. You can also include an image for the control. I haven't done so in this case, but you may want to do so. This information is saved to the configuration file for the add-in:



Caption: Import Wildfires
Class Name: ButtonClassImportWildfires
ID (Variable Name): WildfireAddIn_addin.button
Tooltip: Import wildfires from text file
Message:
Help Heading:
Help Content:
Image for control:

13. Click on the **Save** button.

    Add-ins have a Python script that they are attached to. This file, by default, will be named `AddIns_addin.py` and can be found in the `install` directory of your working project folder.

14. We've already created a custom ArcToolbox Python script tool that loads a comma-delimited text file from a disk containing wildfire data into a feature class. We will be using this script in our add-in. In Windows Explorer, go to the `addin` directory you created earlier. It should be called `Wildfire_Addin`. Go to the `Install` folder and you should find a file called `WildfireAddin_addin.py`. Load this file into your Python editor.
15. Find the `onClick(self)` method, shown in the following code snippet. This method is triggered when the button is clicked:

    ```
    import arcpy
    import pythonaddins

    class ButtonClassImportWildfires(object):
      """Implementation for WildfireAddIn_addin.button (Button)"""
      def __init__(self):
        self.enabled = True
        self.checked = False
      def onClick(self):
        pass
    ```

16. Remove the `pass` statement from the `onClick` event.
17. In Chapter 7, you created a custom script tool that loads wildfire data from a text file to a feature class. Inside the `onClick` event, call this Load Wildfires from Text custom script tool so that it displays the user interface for selecting the text file, template, and feature class to write to.

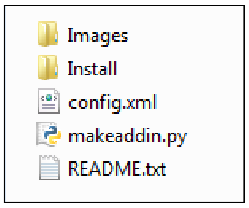
    ```
    def onClick(self):
        LoadWildfires_wildfire()
    ```

18. Save the file.

In the next recipe, you will learn how to install your new add-in.

How it works...

As you've seen in this recipe, the Python Add-In wizard handles the creation of the add-in through a visual interface. However, behind the scenes, the wizard creates a set of folders and files for the add-in. The add-in file structure is really quite simple. Two folders and a set of files comprise the add-in structure. You can see this structure in the following screenshot:



The `Images` folder contains any icons or other image files used by your add-in. In this recipe, we used the `wildfire.png` image. So, this file should now be in the `Images` folder. The `Install` folder contains the Python script that handles the business logic of the add-in. This is the file you work with extensively to code the add-in. It performs whatever business logic needs to be performed by the buttons, tools, menu items, and so on. The `config.xml` file in the main folder of the add-in defines the user interface and any static properties such as the name, author, version, and so on. The `makeaddin.py` file can be double-clicked to create the `.esriaddin` file, which wraps everything into a compressed file with an `.esriaddin` extension. This `.esriaddin` file is what will be distributed to end users, so that the add-in can be installed.

Installing and testing an add-in

You'll want to test add-ins before distributing them to your end users. For testing, you first need to install the add-in.

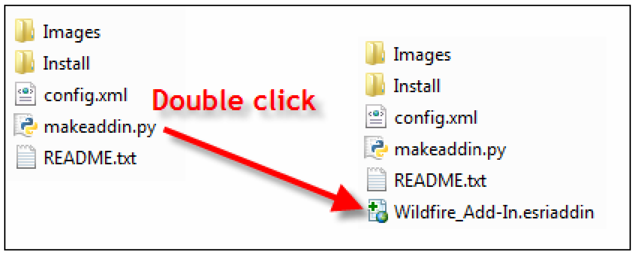
Getting ready

In the working folder for your add-in, the `makeaddin.py` script can be used to copy all files and folders to a compressed add-in folder in a working directory with the file format `<working folder name>.esriaddin`. Double-click on this `.esriaddin` file to launch the ESRI ArcGIS add-in installation utility, which will install your add-in. You can then go into ArcGIS Desktop and test the add-in. The custom toolbar or menu may already be visible and ready to test. If it is not visible, go to the **Customize** menu and click on **Add-in Manager**. The **Add-In Manager** dialog box lists the installed add-ins that target the current application. Add-in information, such as name, description, and image, which are entered as project settings should be displayed.

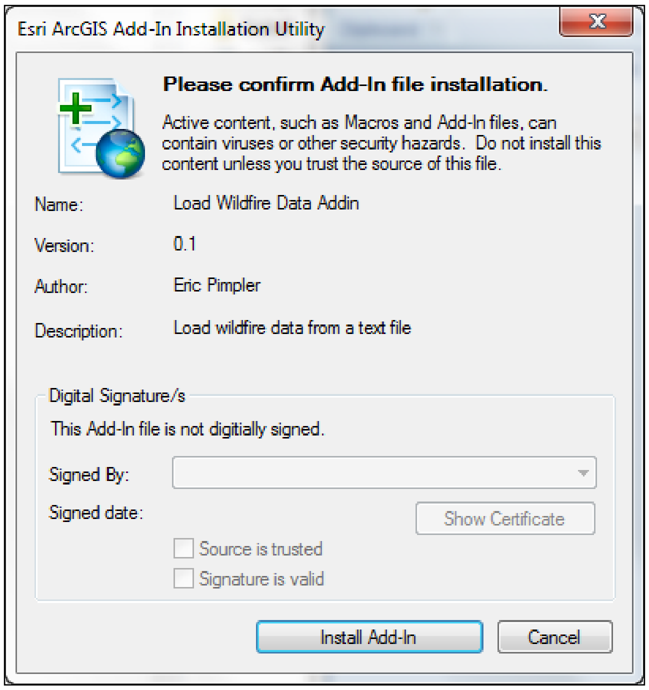
How to do it...

To learn how to install and test a Python add-in, please follow these steps:

1. Inside the main folder for your add-in will be a Python script file called `makeaddin.py`. This script creates the `.esriaddin` file. Double-click on the script to execute and create the `.esriaddin` file. This process is illustrated in the following screenshot:



2. To install the add-in for ArcGIS Desktop, double-click on the `Widlfire_Add-In.esriaddin` file, which will launch the **Esri ArcGIS Add-In Installation Utility** window, as shown in the following screenshot:


Esri ArcGIS Add-In Installation Utility

Please confirm Add-In file installation.

Active content, such as Macros and Add-In files, can contain viruses or other security hazards. Do not install this content unless you trust the source of this file.

Name: Load Wildfire Data Addin

Version: 0.1

Author: Eric Pimpler

Description: Load wildfire data from a text file

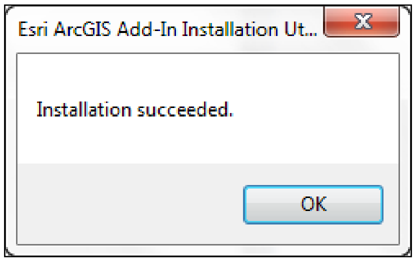
Digital Signature/s

This Add-In file is not digitially signed.

Signed By:

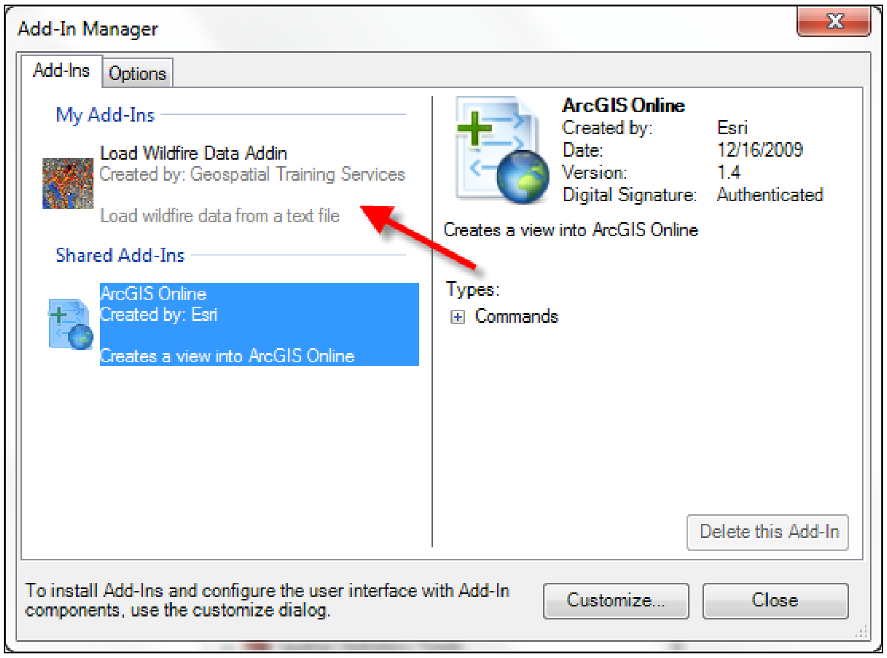
Signed date:

3. Click on **Install Add-In**. If everything was successful, you should see the following message:


Esri ArcGIS Add-In Installation Ut...

Installation succeeded.

4. To test the add-in, open ArcMap. Your add-in may already be active. If not, select **Customize** | **Add-In Manager**. This will display the **Add-In Manager** dialog box , as shown in the following screenshot. You should be able to see the add-in that you created:


Add-In Manager

My Add-Ins

Load Wildfire Data Addin
Created by: Geospatial Training Services

Load wildfire data from a text file

Shared Add-Ins

ArcGIS Online
Created by: Esri

Creates a view into ArcGIS Online

ArcGIS Online

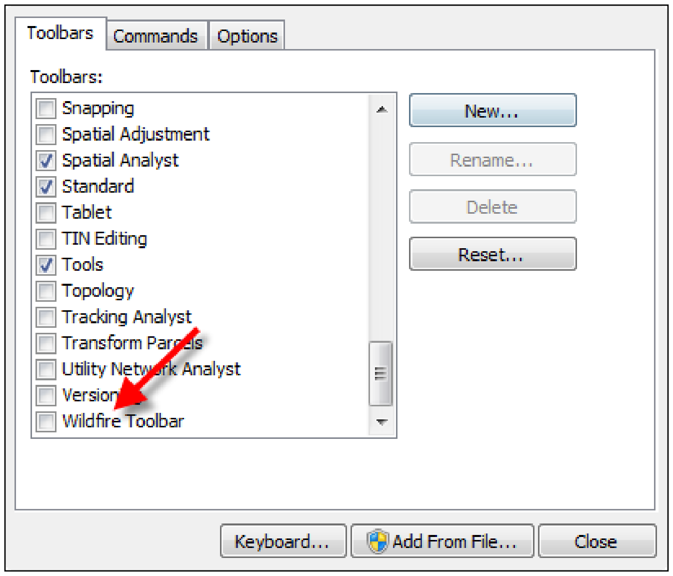
Created by:	Esri
Date:	12/16/2009
Version:	1.4
Digital Signature:	Authenticated

Creates a view into ArcGIS Online

Types:
⊞ Commands

To install Add-Ins and configure the user interface with Add-In components, use the customize dialog.

5. If needed, select the **Customize** button. To add the toolbar to the application, click on the **Toolbars** tab and choose the toolbar you created:

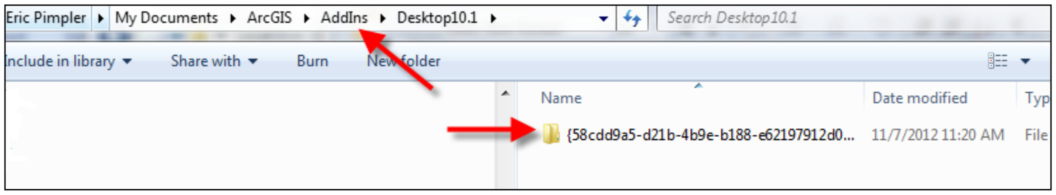


How it works...

The utility will place the add-in into a well-known folder discoverable by ArcGIS Desktop. The well-known folder locations are as follows:

- `Vista/7: C:\Users\<username>\Documents\ArcGIS\AddIns\Desktop10.1`
- `XP: C:\Documents and Settings\<username>\My Documents\ArcGIS\AddIns\Desktop10.1`

A folder with a globally unique identifier or GUID name will be created inside the well-known folder. The add-in will reside inside this unique folder name. This is illustrated in the following screenshots. When ArcGIS Desktop starts, it will search these directories and load the add-ins:



The add-in will look similar to the following:

wildfire_add-in.esriaddin 11/7/2012 11:11 AM

> The default add-in folder is located in the ArcGIS folder within your user account. For example, if your ArcGIS installation is version 10.1, the add-in is copied to the following location on a Vista or Windows 7 operating system: `c:\users\<username>\Documents\ArcGIS\AddIns\Desktop10.1`.

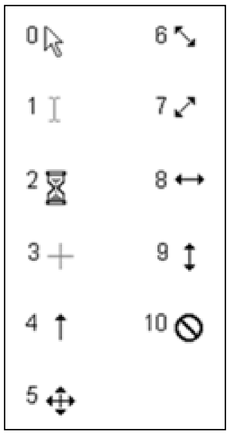
You can also use a private network drive to distribute add-ins to end users. The Add-In Manager in ArcGIS Desktop adds and maintains lists of folders that can be searched for add-ins. Select the **Options** tab and then **Add Folder** to add a network drive to the list.

Creating a tool add-in

Tool add-ins are similar to buttons, with the exception that tools require some type of interaction with the map. The Zoom In tool, for example, is a type of tool. Tools should be placed inside a toolbar or tool palette. The properties are much the same as you'd find with a button. You'll also need to edit the Python script.

Getting ready

The `Tool` class has a number of properties including `cursor`, `enabled`, and `shape`. The `cursor` property sets the cursor for the tool when it is clicked, and is defined as an integer value corresponding to the cursor types, as follows:



By default, tools are enabled. This can be changed, though, by setting the `enabled` property to `false`. Finally, the `shape` property specifies the type of shape to be drawn and can be a line, rectangle, or circle. These properties are typically set inside the constructor for the tool which is defined by the `__init__` method, as shown in the following code example. `self` refers to the current object (a tool in this case) and is a variable that refers to this current object:

```
def __init__(self):
  self.enabled = True
  self.cursor = 3
  self.shape = 'Rectangle'
```

There are a number of functions associated with the `Tool` class. All classes will have a constructor, which is used to define the properties for the class. You saw an example of this `__init__` function earlier. Other important functions of the tool class include `onRectangle()`, `onCircle()`, and `onLine()`. These functions correspond to the shape that will be drawn on the map with the tool. The geometry of the drawn shape is passed into the function. There are also a number of mouse and key functions that can be used. Finally, the `deactivate()` function can be called when you want to deactivate the tool.

We've already seen the constructor for the `Tool` class in action. This function, called `__init__`, is used to set various properties for the tool when it is created. Here, we've also shown the `onRectangle()` function for the `Tool` class. This function is called when a rectangle is drawn on the map. The geometry of the rectangle is passed into the function along with a reference to the tool itself:

```
def onRectangle(self, rectangle_geometry):
```

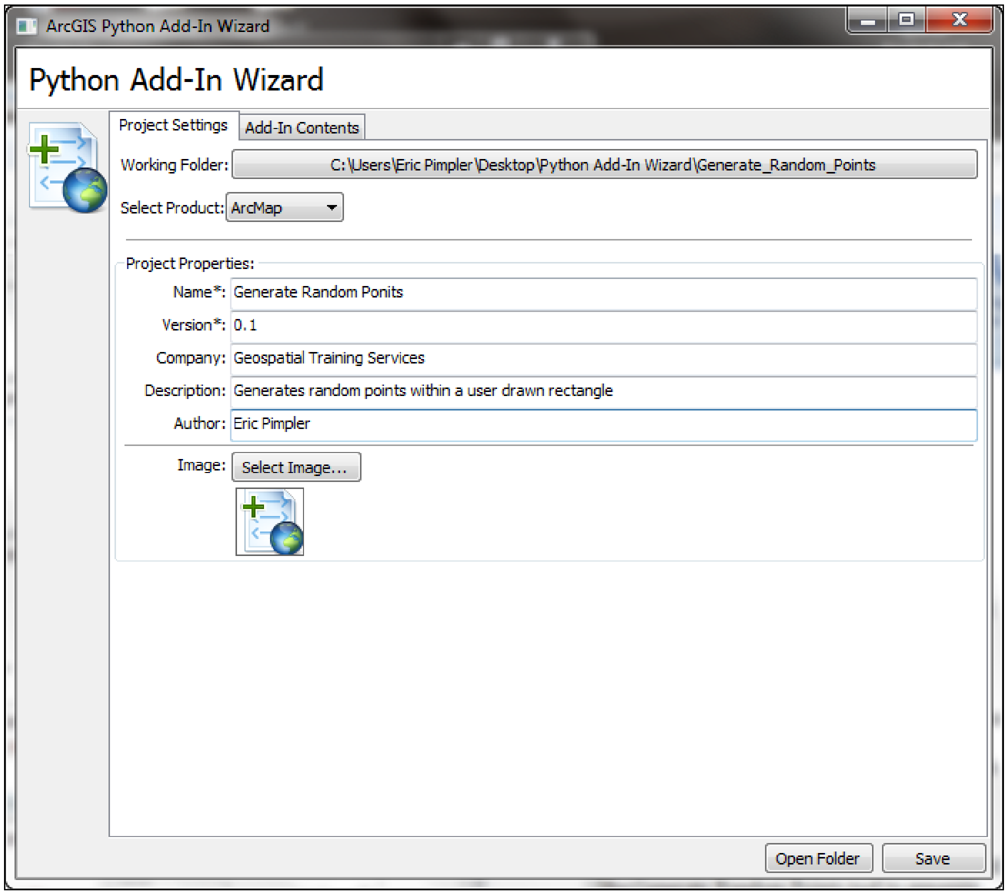
In this recipe, you will learn how to create a tool add-in that responds to the user dragging a rectangle on the map. The tool will use the **Generate Random Points** tool to generate points within the rectangle.

How to do it...

Follow these steps to create a tool add-in with the ArcGIS Python Add-In wizard:

1. Open the ArcGIS Python Add-In wizard by double-clicking on the `addin_assistant.exe` file located in the `bin` folder, where you extracted the wizard.
2. Create a new project folder called `Generate_Random_Points` and click **OK**.

3. In the **Project Settings** tab, enter properties including **Name**, **Version**, **Company**, **Description**, and **Author**:



Working Folder: C:\Users\Eric Pimpler\Desktop\Python Add-In Wizard\Generate_Random_Points

Select Product: ArcMap

Project Properties:

Name*: Generate Random Ponits

Version*: 0.1

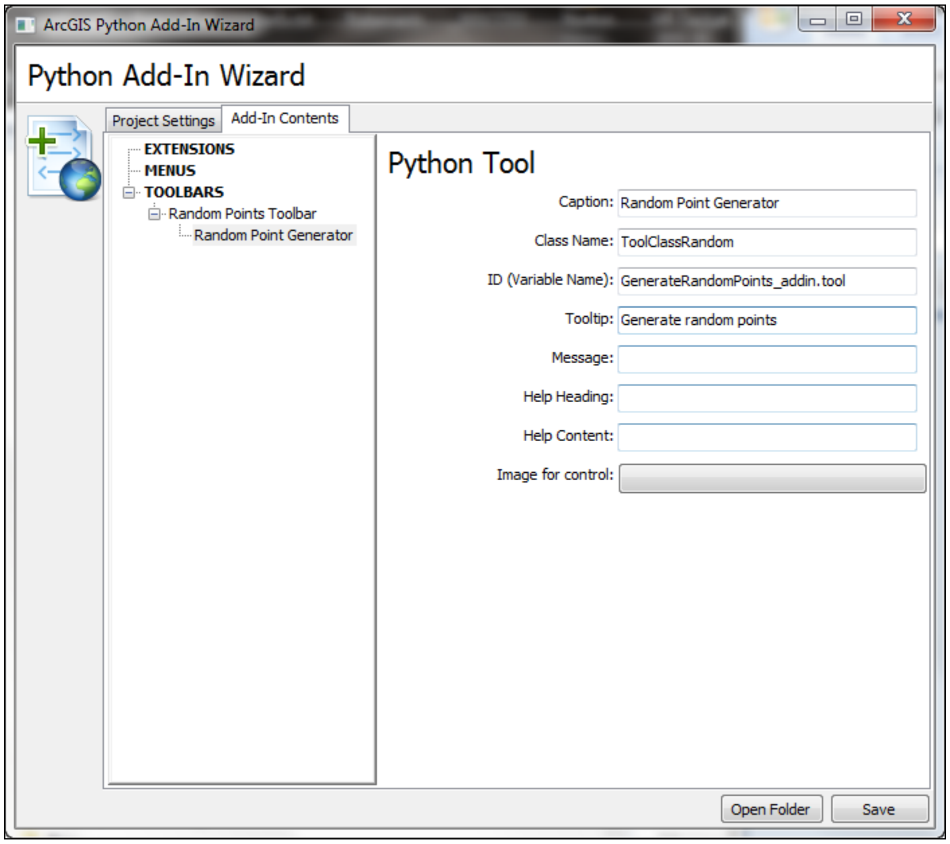
Company: Geospatial Training Services

Description: Generates random points within a user drawn rectangle

Author: Eric Pimpler

4. Click on the **Add-In Contents** tab.
5. Right-click on **Toolbars** and select **New Toolbar**.
6. Set the caption for the toolbar to `Random Points Toolbar`.
7. Right-click on the newly created `Random Points Toolbar` and select **New Tool**.

8. Enter items for the tool as shown in the following screenshot:



Python Tool

Caption: Random Point Generator
Class Name: ToolClassRandom
ID (Variable Name): GenerateRandomPoints_addin.tool
Tooltip: Generate random points
Message:
Help Heading:
Help Content:
Image for control:

9. Click on **Save**. This will generate the folder and file structure for the add-in.
10. Go to the `Install` folder for the new add-in and open `GenerateRandomPoints_addin.py` in IDLE.
11. Add the following code to the `__init__` function, which is the constructor for the tool:

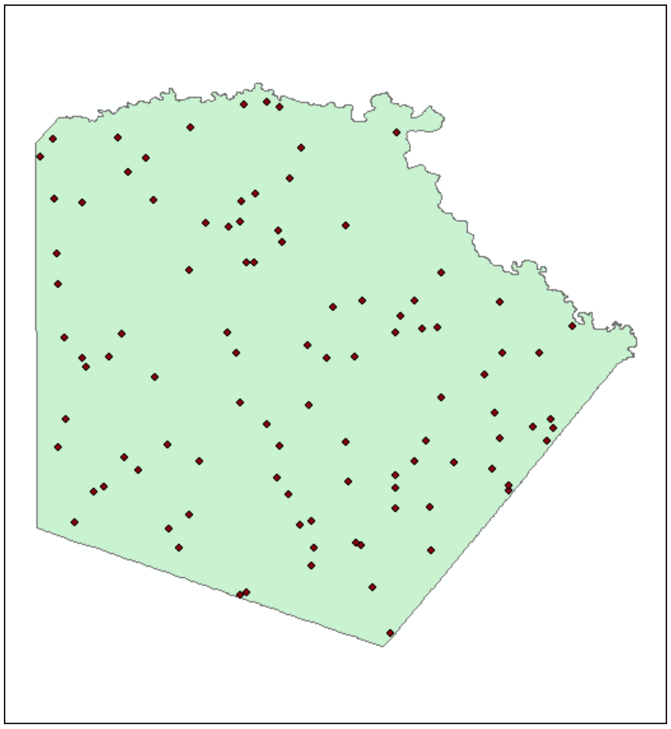
```
def __init__(self):
  self.enabled = True
  self.cursor = 3
  self.shape = 'Rectangle'
```

12. In the `onRectangle()` function, write code to generate a set of random points within the rectangle drawn on the screen:

```
def onRectangle(self, rectangle_geometry):
  extent = rectangle_geometry
```

```
        arcpy.env.workspace = r'c:\ArcpyBook\Ch11'
         if arcpy.Exists('randompts.shp'):
           arcpy.Delete_management('randompts.shp')
           randompts = arcpy.CreateRandomPoints_management(arcpy.env.
workspace,'randompts.shp',"",rectangle_geometry)
           arcpy.RefreshActiveView()
         return randompts
```

13. Save the file.
14. Generate the `.esriaddin` file by double-clicking on the `makeaddin.py` file in the main folder for the add-in.
15. Install the add-in by double-clicking on `Generate_Random_Points.esriaddin`.
16. Open ArcMap with a new map document file and add the Generate Random Points toolbar if necessary.
17. Add the `BexarCountyBoundaries` feature class from `C:\ArcpyBook\data\CityOfSanAntonio.gdb`.
18. Test the add-in by dragging a rectangle on the map. The output should appear similar to the following screenshot. Your map will vary though because the points are generated randomly:



How it works...

Tool add-ins are very similar to button add-ins, the difference being that tool add-ins require some sort of interaction with the map before the functionality is triggered. An interaction with the map can be a number of things including clicking on the map, drawing a polygon or rectangle, or performing various mouse or key events. Python code is written to respond to one or more of these events. In this recipe, you learned how to write code that responds to the `onRectangle()` event. You also set various properties inside the constructor for the add-in including `cursor` and `shape`, which will be drawn on the map.

There's more...

There are a number of additional add-ins that you can create. The `ComboBox` add-in provides a drop-down list of values that the user can select from, or alternatively they can type a new value into an editable field. As with the other add-ins, you'll first want to create a new project with the Python Add-In wizard, add a new toolbar, and then create a combo box to add to the toolbar.

The Tool Palette provides a way of grouping related tools. It does need to be added to an existing toolbar. By default, tools will be added to the palette in a grid-like pattern.

The `Menu` add-in acts as a container for buttons and other menus. Menus, in addition to being displayed by the ArcGIS Desktop Add-in Manager, will also be displayed in the **Customize** dialog box for ArcGIS Desktop.

Application extensions are used to add specific sets of related functionality to ArcGIS Desktop. Several examples include Spatial Analyst, 3D Analyst, and Business Analyst. Typically, application extensions are responsible for listening for events and handling them. For example, you could create an application extension that saves the map document file each time a user adds a layer to the map. Application extensions also coordinate activities between components.

12
Error Handling and Troubleshooting

In this chapter, we will cover the following recipes:

- Exploring the default Python error message
- Adding Python exception handling structures (try/except/finally)
- Retrieving tool messages with GetMessages()
- Filtering tool messages by severity level
- Returning individual messages with GetMessage()
- Testing for and responding to specific error messages

Introduction

Various messages are returned during the execution of ArcGIS geoprocessing tools and functions. These messages can be informational in nature or indicate warning or error conditions that can result in the tool not creating the expected output or in outright failure of the tool to execute. These messages do not appear as message boxes. Instead, you will need to retrieve them using various ArcPy functions. To this point in the book, we have ignored the existence of these messages, warnings, and errors. This is mainly due to the fact that I wanted you to concentrate on learning some basic concepts, without adding the extra layer of code complexity that is necessary for creating robust geoprocessing scripts that can handle error situations gracefully. That being said, it's now time that you learn how to create the geoprocessing and Python exception handling structures that will enable you to create versatile geoprocessing scripts. These scripts can handle messages that indicate warnings, errors, and general information, which are generated while your script is running. These code details will help make your scripts more flexible and less error prone. You've already used the basic `try` and `except` blocks to perform some basic error handling. But, in this chapter, we'll go into more detail about why and how these structures are used.

Exploring the default Python error message

By default, Python will generate an error message anytime it encounters a problem in your script. These error messages will not always be very informative to the end user running the script. However, it is valuable to take a look at these raw messages. In later recipes, we'll use Python error handling structures to get a cleaner look at the errors and respond as necessary.

Getting ready

In this recipe, we will create and run a script that intentionally contains error conditions. We will not include any geoprocessing or Python exception handling techniques in the script. We're intentionally doing this, because I want you to see the error information returned by Python.

How to do it...

Follow the steps below to see a raw Python error message that is generated when an error occurs while a tool is being executed in a script:

1. Open IDLE and create a new script.
2. Save the script to `c:\ArcpyBook\Ch12\ErrorHandling.py`.
3. Import the `arcpy` module:
   ```
   import arcpy
   ```
4. Set the workspace:
   ```
   arcpy.env.workspace = "c:/ArcpyBook/data"
   ```
5. Call the `Buffer` tool. The `Buffer` tool requires a buffer distance be entered as one of its parameters. In this code block, we have intentionally left out the distance parameter:
   ```
   arcpy.Buffer_analysis("Streams.shp","Streams_Buff.shp")
   ```
6. Run the script. You should see the following error message:
   ```
    Runtime error Traceback (most recent call last): File "<string>",
   line 1, in <module> File "c:\program files (x86)\arcgis\
   desktop10.1\arcpy\arcpy\analysis.py", line 687, in Buffer  raise e
   ExecuteError: Failed to execute. Parameters are not valid. ERROR
   000735: Distance [value or field]: Value is required Failed to
   execute (Buffer).
   ```

How it works...

What you see in the output error message isn't terribly informative. If you are a fairly experienced programmer, you'll generally be able to make out what the problem is. In this case we did not include a buffer distance. However, in many cases, the returned error message will not give you much information that you can use to resolve the problem. Errors in your code are simply a fact of life in programming. However, how your code responds to these errors, also called exceptions, is very important. You should plan to handle errors gracefully through the use of Python error handling structures, which examine `arcpy` generated exceptions and act accordingly. Without these structures in place, your scripts will fail immediately, frustrating your users in the process.

Adding Python exception handling structures (try/except/finally)

Python has built-in exception handling structures that allow you to capture error messages that are generated. Using this error information, you can then display a more appropriate message to the end user and respond to the situation as needed.

Getting ready

Exceptions are unusual or error conditions that occur in your code. Exception statements in Python enable you to trap and handle errors in your code, allowing you to gracefully recover from error conditions. In addition to error handling, exceptions can be used for a variety of other things including event notification and special-case handling.

Python exceptions occur in two ways. Exceptions in Python can either be intercepted or triggered. When an error condition occurs in your code, Python automatically triggers an exception, which may or may not be handled by your code. It is up to you as a programmer to catch an automatically triggered exception. Exceptions can also be triggered manually by your code. In this case, you would also provide an exception handling routine to catch these manually triggered exceptions. You can manually trigger an exception by using the `raise` statement.

The `try/except` statement is a complete, compound Python statement used to handle exceptions. This variety of `try` statement starts with a `try` header line followed by a block of indented statements, then one or more optional `except` clauses that name exceptions to be caught, and an optional `else` clause at the end.

The `try/except/else` statement works as follows. Once inside a `try` statement, Python marks the fact that you are in a `try` block and knows that any exception condition that occurs within this block will be forwarded to the various `except` statements for handling.

Each statement inside the `try` block is executed. Assuming that no exception conditions occur, the code pointer will then jump to the `else` statement and execute the code block contained within the `else` statement before moving to the next line of code below the `try` block. If an exception occurs inside the `try` block, Python searches for a matching exception code. If a matching exception is found, the code block inside the `except` block is executed. The code then picks up below the full `try` statement. The `else` statements are not executed in this case. If a matching exception header is not found, Python will propagate the exception to a `try` statement above this code block. In the event that no matching `except` header is found, the exception comes out of the top level of the process. This results in an unhandled exception and you wind up with the type of error message that we saw in our first recipe in this chapter.

In this recipe, we're going to add in some basic Python exception handling structures. There are several variations of the `try/except/else/finally` exception handling structure. In this recipe, we'll start with a very simple `try/except` structure.

How to do it…

Follow the steps below to add Python error handling structures to a script:

1. If necessary, open the `c:\ArcpyBook\Ch12\ErrorHandling.py` file in IDLE.
2. Alter your script to include a `try/except` block:

```
import arcpy
try:
  arcpy.env.workspace = "c:/ArcpyBook/data"
  arcpy.Buffer_analysis("Streams.shp","Streams_Buff.shp")
except:
  print "Error"
```

3. Save and run the script. You should see the simple message `Error`. That's not any more helpful than the output we received in our first recipe. In fact, it's even less useful. However, the point of this recipe is simply to introduce you to the `try/except` error handling structure.

How it works…

This is an extremely simple structure. The `try` block indicates that everything indented under the `try` statement will be subject to exception handling. If an exception of any type is found, control of the code processing jumps to the `except` section and prints the error message(s), which in this case is simply `Error`. Now, as I mentioned, this is hardly informative to your users, but hopefully it gives you a basic idea of how `try/except` blocks work, and as a programmer you will better understand any errors reported by your users. In the next recipe, you'll learn how to add tool-generated messages to this structure.

There's more...

The other type of `try` statement is the `try/finally` statement, which allows for finalization actions. When a `finally` clause is used in a `try` statement, its block of statements always run at the very end, whether an error condition occurs or not. The `try/finally` statement works as follows. If an exception occurs, Python runs the `try` block, then the `finally` block, and then execution continues past the entire `try` statement. If an exception does not occur during execution, Python runs the `try` block, then the `finally` block, and then execution is passed back to a higher level `try` statement. This is useful when you want to make sure an action takes place after a code block runs regardless of whether or not an error condition occurs.

Retrieving tool messages with GetMessages()

ArcPy includes a `GetMessages()` function that you can use to retrieve messages generated when an ArcGIS tool is executing. Messages can include informational messages, such as the start and ends times of a tool execution as well as warnings and errors, which can result in something less than the desired result or complete failure of the tool to execute to completion.

Getting ready

During the execution of a tool, various messages are generated. These messages include informational messages, such as the start and end times of a tool execution, parameter values passed to the tool, and progress information. In addition, warnings and errors can also be generated by the tool. These messages can be read by your Python script, and your code can be designed to appropriately handle any warnings or errors that have been generated.

ArcPy stores the messages from the last tool that was executed and you can retrieve these messages using the `GetMessages()` function, which returns a single string containing all messages from the tool that was last executed. You can filter this string by severity to return only certain types of messages, such as warnings or errors. The first message will always include the name of the tool executed, and the last message is the start and end time.

In this recipe, you will add a line of code to the `except` statement, which will print more descriptive information about the current tool run.

How to do it...

Follow these steps to learn how to add a `GetMessages()` function to your script that generates a list of messages from the tool last executed.

1. If necessary, open the `c:\ArcpyBook\Ch12\ErrorHandling.py` file in IDLE.

2. Alter your script to include the `GetMessages()` function:

```
import arcpy
try:
  arcpy.env.workspace = "c:/ArcpyBook/data"
  arcpy.Buffer_analysis("Streams.shp","Streams_Buff.shp")
except:
  print arcpy.GetMessages()
```

3. Save and run the script. This time, the error message should be much more informative. Also notice that there are other types of messages that are generated including the start and end times of the script's execution:

```
Executing: Buffer c:/ArcpyBook/data\Streams.shp c:/ArcpyBook/data\
Streams_Buff.shp # FULL ROUND NONE #
Start Time: Tue Nov 13 22:23:04 2012
Failed to execute. Parameters are not valid.
ERROR 000735: Distance [value or field]: Value is required
Failed to execute (Buffer).
Failed at Tue Nov 13 22:23:04 2012 (Elapsed Time: 0.00 seconds)
```

How it works...

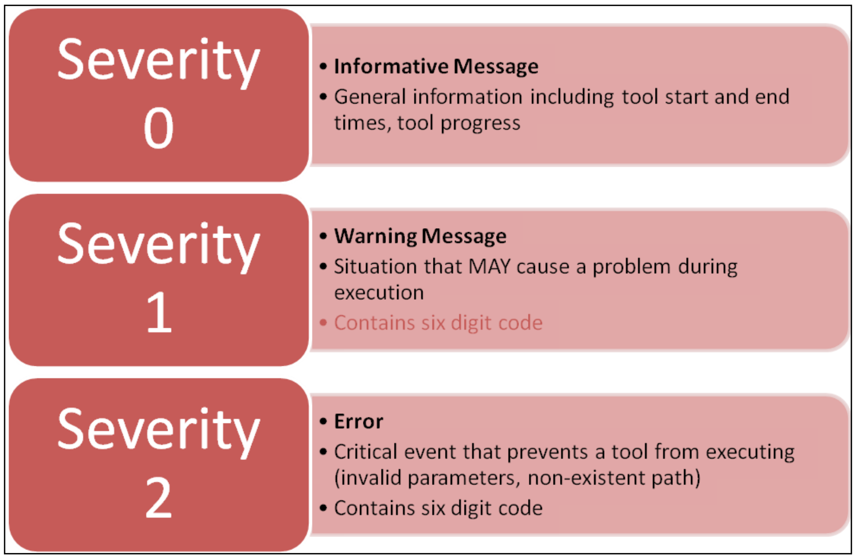
The `GetMessages()` function returns all the messages generated by the last tool that was run. I want to emphasize that it only returns messages from the last tool that was run. Keep this in mind if you have a script with multiple tools that are being run. Historical tool run messages are not accessible through this function. However, there is a `Result` object that you can use if you need to retrieve historical tool run messages.

Filtering tool messages by severity level

As I mentioned in the last recipe, all tools generate a number of messages that can be classified as information, warning, or error messages. The `GetMessages()` method accepts a parameter that allows you to filter the messages that are returned. For example, you may not be interested in the informative or warning messages in your script. However, you are certainly interested in error messages as they indicate a fatal error that will not allow a tool to successfully execute. Using `GetMessages()`, you can filter the returned message to only error messages.

Getting ready

Messages are classified into one of the three types, which are indicated by a severity level. **Informational messages** provide descriptive information concerning things such as a tools progress, start and end times of the tool, output data characteristics, and much more. The severity level of an informational message is indicated by a value of `0`. **Warning messages** are generated when a problem has occurred during execution that may affect the output. Warnings are indicated with a severity level of `1` and don't normally stop a tool from running. The last type of message is an **error message**, which is indicated with a numeric value of `2`. These indicate fatal events that prevent a tool from running. Multiple messages may be generated during the execution of a tool, and these are stored in a list. More information about message severity levels is provided in the following image. In this recipe, you will learn how to filter the messages generated by the `GetMessages()` function.


Severity 0
• Informative Message
• General information including tool start and end times, tool progress
Severity 1
• Warning Message
• Situation that MAY cause a problem during execution
• Contains six digit code
Severity 2
• Error
• Critical event that prevents a tool from executing (invalid parameters, non-existent path)
• Contains six digit code

How to do it...

Filtering the messages returned by a tool is really quite simple. You simply provide the severity level you'd like to return as a parameter to the `GetMessages()` function.

1. If necessary, open the `c:\ArcpyBook\Ch12\ErrorHandling.py` file in IDLE.

2. Alter the `GetMessages()` function so that you pass in a value of `2` as the only parameter:

```
import arcpy
try:
  arcpy.env.workspace = "c:/ArcpyBook/data"
  arcpy.Buffer_analysis("Streams.shp","Streams_Buff.shp")
except:
  print arcpy.GetMessages(2)
```

3. Save and run the script to see the output:

```
Failed to execute. Parameters are not valid.
ERROR 000735: Distance [value or field]: Value is required
Failed to execute (Buffer).
```

How it works...

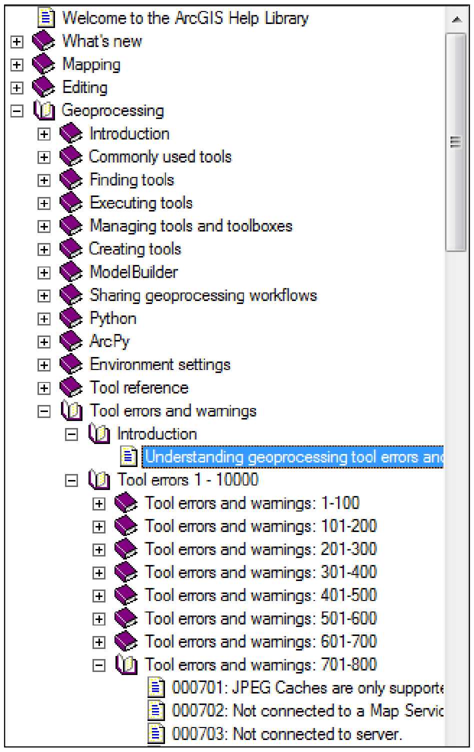
As I mentioned, the `GetMessages()` method can accept an integer argument of `0`, `1`, or `2`. Passing a value of `0` indicates that all messages should be returned, while passing a value of `1` indicates that you wish to see warnings. In our case, we have passed a value of `2`, which indicates that we only want to see error messages. Therefore, you won't see any of the other information messages, such as the start and end times of the script.

Testing for and responding to specific error messages

All errors and warnings generate a specific error code. It is possible to check for specific error codes in your scripts and perform some type of action based on these errors. This can make your scripts even more versatile.

Getting ready...

All errors and warnings generated by a geoprocessing tool contain both a six digit code and a description.Your script can test for specific error codes and respond accordingly.You can get a listing of all the available error messages and codes in the ArcGIS Desktop help system by going to **Geoprocessing** | **Tool errors and warnings**.This is illustrated in the following screenshot.All errors will have a unique page that briefly describes the error by code number:



- Welcome to the ArcGIS Help Library
- What's new
- Mapping
- Editing
- Geoprocessing
  - Introduction
  - Commonly used tools
  - Finding tools
  - Executing tools
  - Managing tools and toolboxes
  - Creating tools
  - ModelBuilder
  - Sharing geoprocessing workflows
  - Python
  - ArcPy
  - Environment settings
  - Tool reference
  - Tool errors and warnings
    - Introduction
      - Understanding geoprocessing tool errors anc
    - Tool errors 1 - 10000
      - Tool errors and warnings: 1-100
      - Tool errors and warnings: 101-200
      - Tool errors and warnings: 201-300
      - Tool errors and warnings: 301-400
      - Tool errors and warnings: 401-500
      - Tool errors and warnings: 501-600
      - Tool errors and warnings: 601-700
      - Tool errors and warnings: 701-800
        - 000701: JPEG Caches are only supporte
        - 000702: Not connected to a Map Servic
        - 000703: Not connected to server.

How to do it...

Follow these steps to learn how to write a code that responds to specific error code generated by the execution of a geoprocessing tool:

1. Open the ArcGIS Desktop help system by going to **Start** | **Programs** | **ArcGIS** | **ArcGIS for Desktop Help**.
2. Go to **Geoprocessing** | **Tool errors and warnings** | **Tool errors 1-10000** |**Tool errors and warnings: 701-800**.
3. Select **000735: <value>: Value is required**.This error indicates that a parameter required by the tool has not been provided.You'll recall from running this script that we have not provided the buffer distance and that the error message generated, as a result, contains the error code that we are viewing in the help system.In the following code, you will find the full text of the error message.Notice the error code.

   ```
   ERROR000735:Distance[valueorfield]:Valueisrequired
   ```

4. If necessary, open the `c:\ArcpyBook\Ch12\ErrorHandling.py` file in IDLE.
5. In your script, alter the `except` statement, so that it appears as follows:

```
except:
 print "Error found in Buffer tool \n"
errCode = arcpy.GetReturnCode(3)
 if str(errCode) in "735":
 print "Distance value not provided \n"
 print "Running the buffer again with a default value \n"
defaultDistance = "100 Feet"
arcpy.Buffer_analysis("Streams.shp", "Streams_Buff",
defaultDistance)
 print "Buffer complete"
```

6. Save and run the script.You should see various messages printed, as follows:

```
Error found in Buffer tool
Distance value not provided for buffer
Running the buffer again with a default distance value
Buffer complete
```

How it works...

What you've done in this code block is use the `arcpy.GetReturnCode()` function to return the error code generated by the tool.Then, an `if` statement is used to test if the error code contains the value `735`, which is the code that indicates that a required parameter has not been provided to the tool.You then provided a default value for the buffer distance and called the `Buffer` tool again; providing the default buffer valuethis time.

Returning individual messages with GetMessage()

While `GetMessages()` returns a list containing all messages from the last tool run, you can also get individual messages from the string using `GetMessage()`.

Getting ready

Up to this point, we have been returning all messages generated by the tool. However, you can return individual messages to your user through the `GetMessage()` method, which takes an integer as a parameter indicating the particular message you'd like to retrieve. Each message generated by the tool is placed into a message list or array. We discussed list objects earlier in the book, so you'll remember that this is just a collection of some type of object.

Just as a reminder: lists are zero-based, meaning that the first item in the list is located at position `0`. For example, `GetMessage(0)` would return the first message in the list, while `GetMessage(1)` would return the second message in the list. The first message will always be the tool being executed along with any parameters. The second message returns the start time of the script, while the last message returns the end time of the script.

How to do it...

1. If necessary, open the `c:\ArcpyBook\Ch12\ErrorHandling.py` file in IDLE.
2. Alter the `except` block as follows:

```
import arcpy
try:
  arcpy.env.workspace = "c:/ArcpyBook/data"
  arcpy.Buffer_analysis("Streams.shp","Streams_Buff.shp")
except:
  print arcpy.GetMessage(1)
  print arcpy.GetMessage(arcpy.GetMessageCount() - 1)
```

3. Save and run the script to see the output:

```
Start Time: Wed Nov 14 09:07:35 2012
Failed at Wed Nov 14 09:07:35 2012 (Elapsed Time: 0.00 seconds)
```

How it works...

We haven't covered the `GetMessageCount()` function yet. This function returns the number of messages returned by the tool. Remember that our list of messages is zero-based, so we have to subtract one from the `GetMessageCount()` function to arrive at the last message in the list. Otherwise, we'd be attempting to access a message that does not exist. In this script, we have accessed the start and end times of the script. The second message is always the start time for the script, while the last message will always be the end time of the script. This concept is illustrated as follows:

```
Message 0 - Executing: Buffer c:/ArcpyBook/data\Streams.shp c:/ArcpyBook/
data\Streams_Buff.shp # FULL ROUND NONE #
Message 1 - Start Time: Tue Nov 13 22:23:04 2012
Message 2 - Failed to execute. Parameters are not valid.
Message 3 - ERROR 000735: Distance [value or field]: Value is required
Message 4 - Failed to execute (Buffer).
Message 5 - Failed at Tue Nov 13 22:23:04 2012 (Elapsed Time: 0.00
seconds)
```

Total message count is `6`, but the last message is number `5`. This is because the count starts with `0`. This is why you need to subtract `1` as stated previously. In this case, the start and end times are the same, because the script contains an error. However, it does illustrate how to access the individual messages generated by the tool.

A
Automating Python Scripts

In this chapter, we will cover the following recipes:

- Running Python scripts from the command line
- Using sys.argv[] to capture command-line input
- Adding Python scripts to batch files
- Scheduling batch files to run at prescribed times

Introduction

Python geoprocessing scripts can be executed either outside ArcGIS as a standalone script or inside ArcGIS as a script tool. Both methods have their advantages and disadvantages. Up to this point in the book, all our scripts have been run either inside ArcGIS as a script tool, or from a Python development environment such as IDLE, or the Python Window in ArcGIS. However, Python scripts can also be executed from the Windows operating system command line. The command line is a window that you can use to type in commands rather than the usual point-and-click approach provided by Windows. This method of running Python scripts is useful for scheduling the execution of a script. There are a number of reasons why you might want to schedule your scripts. Many geoprocessing scripts take a long time to fully execute and need to be scheduled to run during non-working hours on a regular basis. Additionally, some scripts need to be executed on a routine basis (every day, week, month, and so on), and should be scheduled for efficiency. In this chapter, you will learn how to execute scripts from the command line, place scripts inside batch files, and schedule the execution of scripts at prescribed times. Please keep in mind that any scripts run from the command line will still need access to an ArcGIS Desktop license in order to use the `arcpy` module.

Running Python scripts from the command line

Up to this point in the book, all your Python scripts have been run as either script tools in ArcGIS or from a Python development environment. The Windows command prompt provides yet another way of executing your Python scripts. The command prompt is used primarily to execute scripts that will be run as a part of a batch file and/or as scheduled tasks.

Getting ready

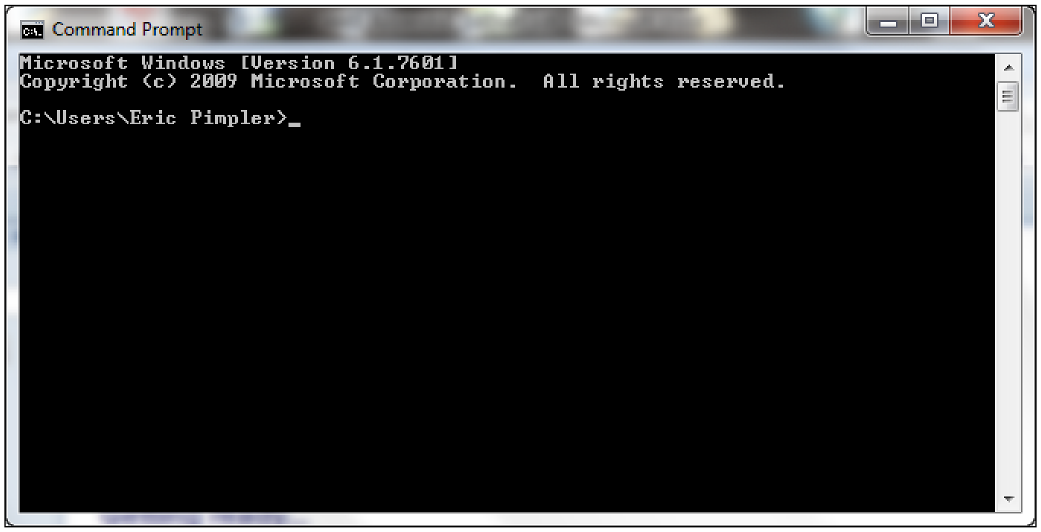
There are a couple of advantages to running Python geoprocessing scripts from the command prompt. These scripts can be scheduled to batch process your data during off hours for more efficient processing, and they are easier to debug due to the built-in Python error handling and debugging capabilities.

In this recipe, you will learn how to use the Windows command prompt to execute a Python script. You will need administrative rights to complete this recipe, so you may need to contact your information technology support group to make this change.

How to do it...

Follow these steps to learn how to run a script from the Windows command prompt:

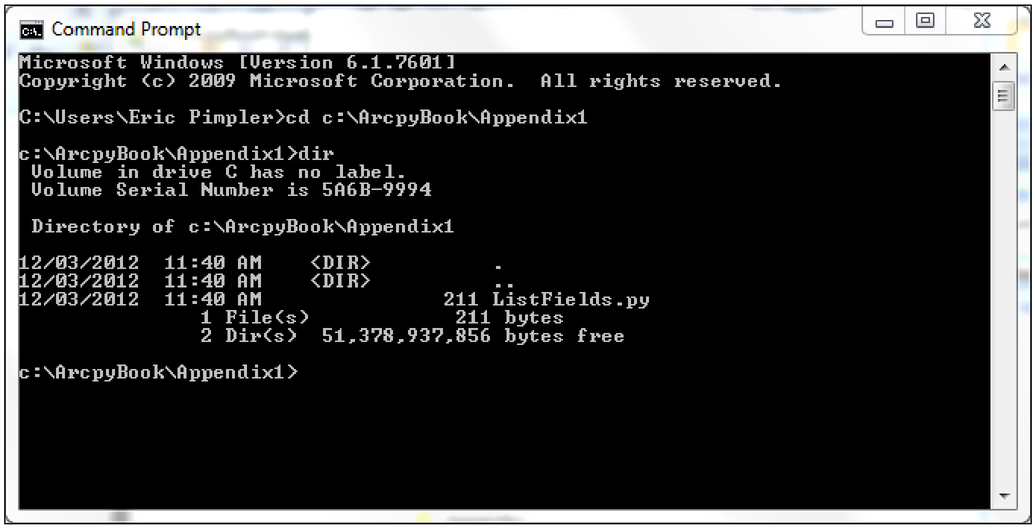
1. In Windows, go to **Start** | **All Programs** | **Accessories** | **Command Prompt** to display a window similar to the following screenshot:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Eric Pimpler>_
```

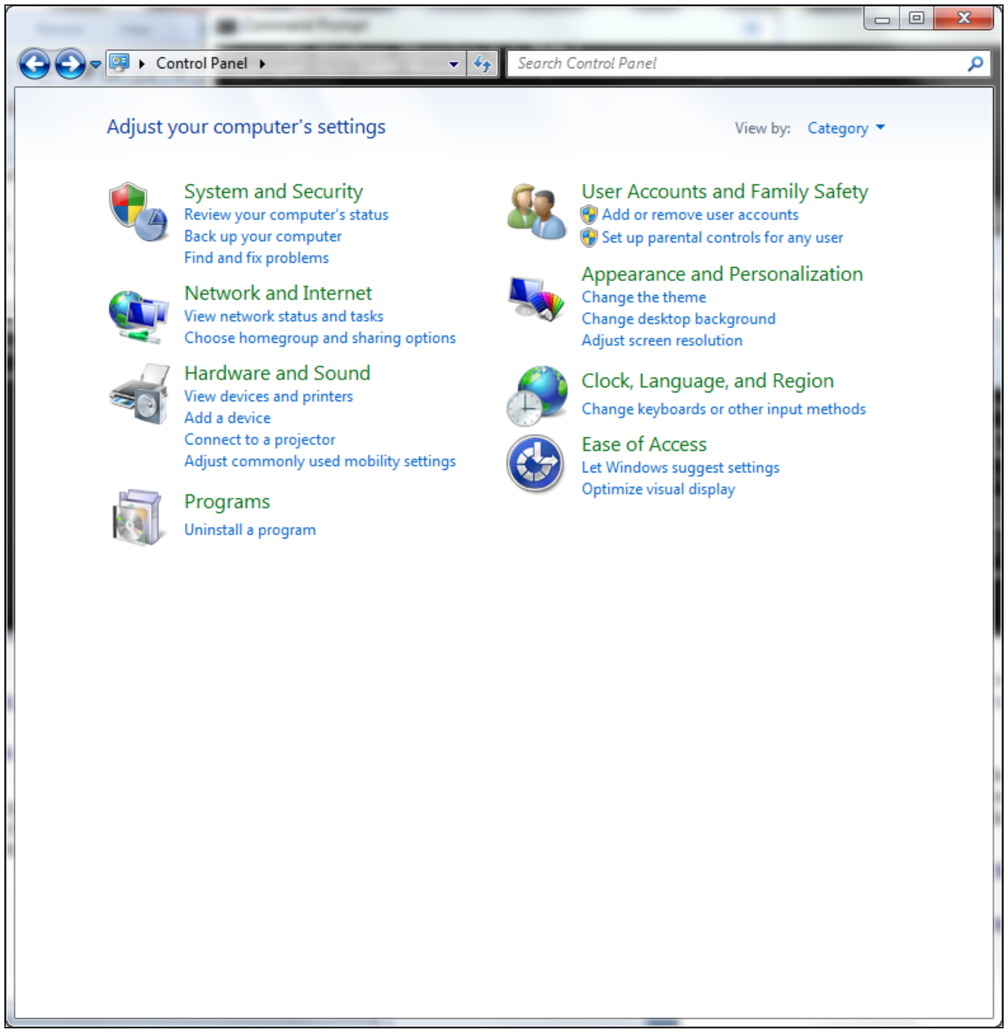
The window will display the current directory. Your directory will differ to some degree. Let's change to the directory for this appendix.

2. Type `cd c:\ArcpyBook\Appendix1`.
3. Type dir to see a listing of the files and sub-directories. You should see only a single Python file called `ListFields.py`:

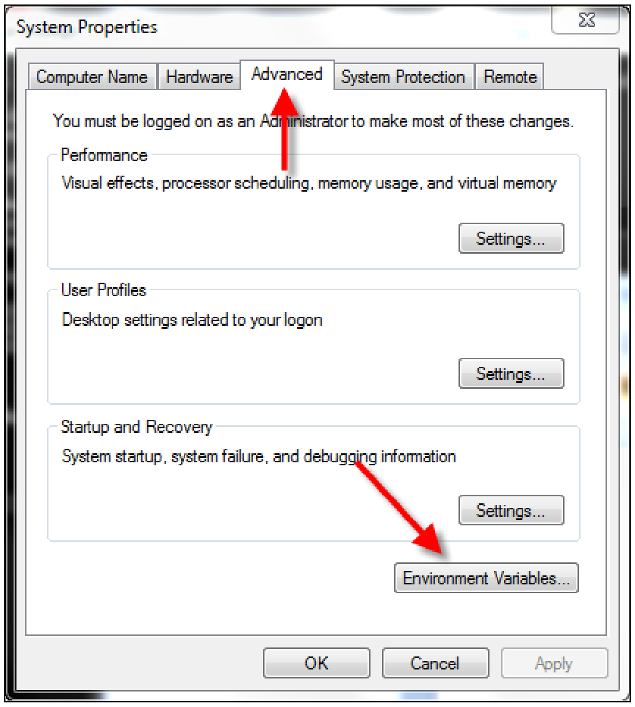


```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Eric Pimpler>cd c:\ArcpyBook\Appendix1

c:\ArcpyBook\Appendix1>dir
 Volume in drive C has no label.
 Volume Serial Number is 5A6B-9994

 Directory of c:\ArcpyBook\Appendix1

12/03/2012  11:40 AM    <DIR>          .
12/03/2012  11:40 AM    <DIR>          ..
12/03/2012  11:40 AM               211 ListFields.py
               1 File(s)            211 bytes
               2 Dir(s)  51,378,937,856 bytes free

c:\ArcpyBook\Appendix1>
```

4. You will want to make sure that the Python interpreter can be run from anywhere in your directory structure. Go to **Start** | **All Programs** | **Accessories** | **System Tools** | **Control Panel**.


Control Panel
Search Control Panel
Adjust your computer's settings
View by: Category
System and Security
Review your computer's status
Back up your computer
Find and fix problems
Network and Internet
View network status and tasks
Choose homegroup and sharing options
Hardware and Sound
View devices and printers
Add a device
Connect to a projector
Adjust commonly used mobility settings
Programs
Uninstall a program
User Accounts and Family Safety
Add or remove user accounts
Set up parental controls for any user
Appearance and Personalization
Change the theme
Change desktop background
Adjust screen resolution
Clock, Language, and Region
Change keyboards or other input methods
Ease of Access
Let Windows suggest settings
Optimize visual display

5. Click on **System and Security**.
6. Click on **System**.
7. Click on **Advanced system settings**.
8. In the **System Properties** dialog box, select the **Advanced** tab and then the **Environment Variables** button as shown in the following screenshot:



System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

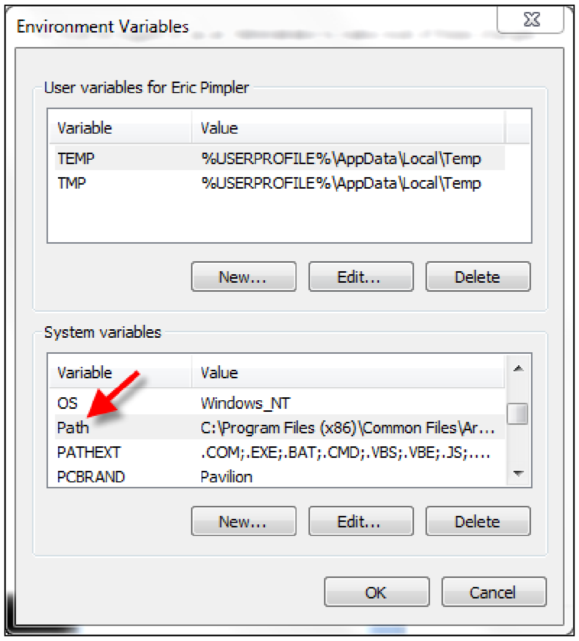
User Profiles

Desktop settings related to your logon

Startup and Recovery

System startup, system failure, and debugging information

9. Find the **Path** system variable seen in the screenshot below and click on **Edit**.



Environment Variables

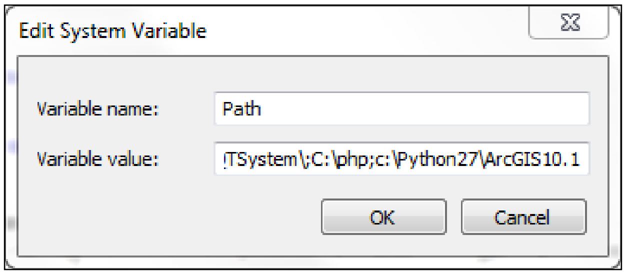
User variables for Eric Pimpler

Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
OS	Windows_NT
Path	C:\Program Files (x86)\Common Files\Ar...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PCBRAND	Pavilion

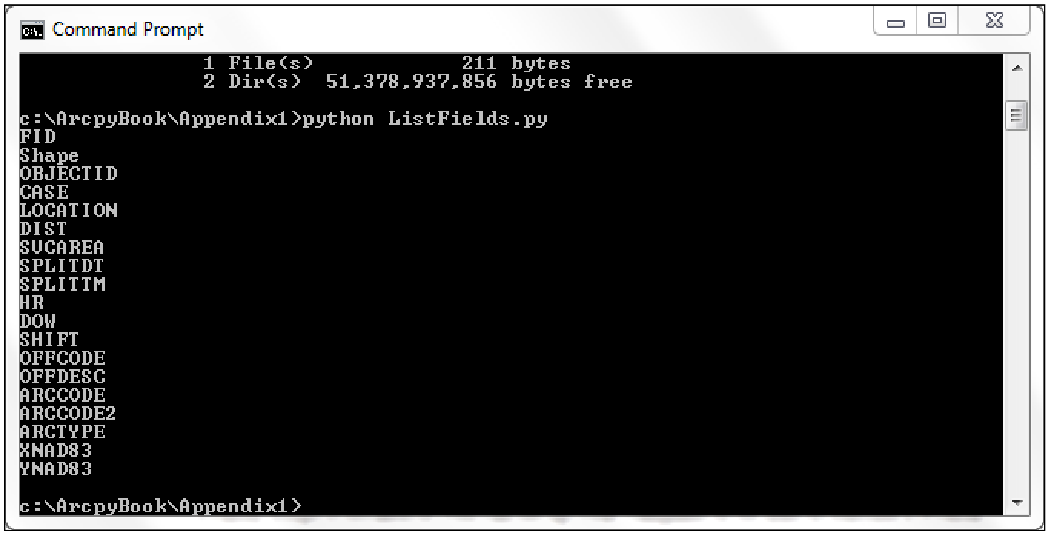
10. Examine the entire text string for the directory `c:\Python27\ArcGIS10.1`. If the text string isn't found, add it to the end. Make sure that you add a semi-colon before adding the path. Now, when you type `python` in the command prompt, it will look through each of the directories in the **Path** system variable, checking for an executable called `python.exe`.



Variable name: Path

Variable value: !TSystem\;C:\php;c:\Python27\ArcGIS10.1

11. Click on **OK** to dismiss the **Edit System Variable** dialog box.
12. Click on **OK** to dismiss the **Environment Variables** dialog box.
13. Click on **OK** to dismiss the **System Properties** dialog box.
14. Return to the command prompt.
15. Type `python ListFields.py`. This will run the `ListFields.py` script. After a brief delay, you should see the following output:



```
               1 File(s)            211 bytes
               2 Dir(s)  51,378,937,856 bytes free

c:\ArcpyBook\Appendix1>python ListFields.py
FID
Shape
OBJECTID
CASE
LOCATION
DIST
SUCAREA
SPLITDT
SPLITTM
HR
DOW
SHIFT
OFFCODE
OFFDESC
ARCCODE
ARCCODE2
ARCTYPE
XNAD83
YNAD83

c:\ArcpyBook\Appendix1>
```

How it works...

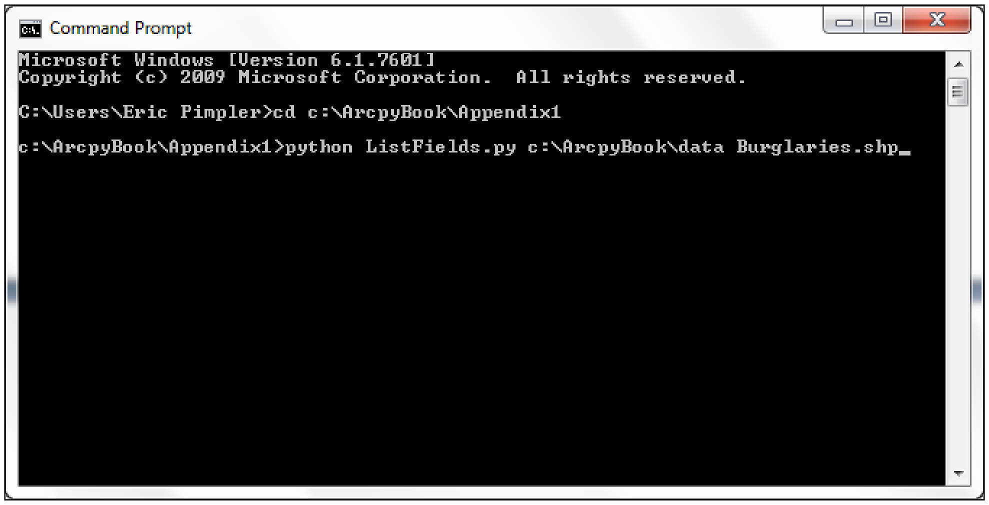
The `ListFields.py` script provided for you in this recipe is a simple script that lists the attribute fields for the `Burglaries_2009.shp` file. The workspace and shapefile name are hardcoded in the script. Typing `python` followed by the name of the script, which is `ListFields.py` in this case, triggered the execution of a script using the Python interpreter. As I mentioned, the workspace and shapefile name were hardcoded in this script. In the next recipe, you will learn how to pass in arguments to the script, so that you can remove the hardcoding and make your script more flexible.

Using sys.argv[ ] to capture command-line input

Instead of hardcoding your scripts with paths to specific datasets, you can make your scripts more flexible by allowing them to accept input in the form of parameters from the command prompt. These input parameters can be captured using Python's `sys.argv[]` object.

Getting ready

Python's `sys.argv[]` object allows you to capture input parameters from the command line when a script is executed. An example is useful for illustrating how this works. Take a look at the following screenshot:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Eric Pimpler>cd c:\ArcpyBook\Appendix1

c:\ArcpyBook\Appendix1>python ListFields.py c:\ArcpyBook\data Burglaries.shp_
```

Each word must be separated by a space. These words are stored in a zero-based list object called `sys.argv[]`. With `sys.argv[]`, the first item in the list, referenced by index `0`, stores the name of the script. In this case, it would be `ListFields.py`. Each successive word is referenced by the next integer. Therefore, the first parameter (`c:\ArcpyBook\data`) will be stored in `sys.argv[1]`, and the second parameter (`Burglaries.shp`) will be stored in `sys.argv[2]`. Each of the arguments in the `sys.argv[]` object can be accessed and used inside your geoprocessing script. In this recipe, you're going to update the `ListFields.py` script to accept input parameters from the command line.

How to do it…

Follow these steps to create a Python script that can accept input parameters from the command prompt using `sys.argv[]`:

1. Open `C:\ArcpyBook\Appendix1\ListFields.py` in IDLE.
2. Import the `sys` module:
   ```
   import arcpy, sys
   ```
3. Create a variable to hold the workspace that will be passed into the script:
   ```
   wkspace = sys.argv[1]
   ```
4. Create a variable to hold the feature class that will be passed into the script:
   ```
   fc = sys.argv[2]
   ```
5. Update the lines of code that set the workspace and call the `ListFields()` function:
   ```
   arcpy.env.workspace = wkspace
   fields = arcpy.ListFields(fc)
   ```
   Your completed script should appear as follows:
   ```
   import arcpy, sys
   wkspace = sys.argv[1]
   fc = sys.argv[2]
   try:
     arcpy.env.workspace = wkspace
     fields = arcpy.ListFields(fc)
     for fld in fields:
       print fld.name
   except:
     print arcpy.GetMessages()
   ```
6. Save the script.
7. If necessary, open the command prompt and navigate to `c:\ArcpyBook\Appendix1`.

8. On the command line, type the following and press on the *Enter* key:

   ```
   python ListFields.py c:\ArcpyBook\data Burglaries_2009.shp
   ```

9. Once again you should see the output detailing the attribute fields for the `Burglaries_2009.shp` file. The difference is that your script no longer has a hardcoded workspace and feature class name. You now have a more flexible script capable of listing the attribute fields for any feature class.

How it works...

The `sys` module contains a list of objects called `argv[]`, which is used to store the input parameters for the command-line execution of a Python script. The first item stored in the list is always the name of the script. So, in this case, `sys.argv[0]` contains the word `ListFields.py`. Two parameters are passed into the script, including the workspace and a feature class. These are stored in `sys.argv[1]` and `sys.argv[2]` respectively. These values are then assigned to variables and used in the script.

Adding Python scripts to batch files

Scheduling your Python scripts to run at prescribed times will require that you create a batch file containing one or more scripts and/or operating system commands. These batch files can then be added to the Windows Scheduler to run at a specific time interval.

Getting ready

Batch files are text files containing command-line sequences for running Python scripts or performing operating system commands. They have a file extension of `.bat`, which Windows recognizes as an executable file. Since batch files simply contain command-line sequences, they can be written with any text editor, though it is recommended that you use a basic text editor such as Notepad, so that you can avoid the inclusion of invisible special characters, which are sometimes inserted by programs such as Microsoft Word. In this recipe, you will create a simple batch file that navigates to the directory containing your `ListFields.py` script and executes it.

How to do it...

Follow these steps to create a batch file:

1. Open a notepad.
2. Add the following lines of text to the file:

   ```
   cd c:\ArcpyBook\Appendix1
   python ListFields.py c:\ArcpyBook\data Burglaries_2009.shp
   ```

3. Save the file to your desktop as `ListFields.bat`. Make sure you change the **Save as Type** drop-down list to **All Files**, or else you'll wind up with a file called `ListFields.bat.txt`.
4. In Windows, navigate to your desktop and double-click on `ListFields.bat` to execute the sequence of commands.
5. A command prompt will be displayed during execution. After the commands have been executed, the command prompt will automatically close.

How it works...

Windows treats a batch file as an executable, so double-clicking on the file will automatically execute the sequence of commands contained within the file in a new command prompt window. All `print` statements will be written to the window. After the commands have been executed, the command prompt will automatically close. In the event that you need to keep track of the output, you can write the statements to an output log file.

There's more...

Batch files can contain variables, loops, comments, and conditional logic. This functionality is beyond the scope of this recipe. However, if you'll be writing and running many scripts for your organization, it's worthwhile to spend some time learning more about batch files. Batch files have been around for a long time, so there is no shortage of information about these files on the Web. For more information about batch files, please consult the Wikipedia page for this topic.

Scheduling batch files to run at prescribed times

Once created, your batch files can then be scheduled to run at prescribed times using the Windows Scheduler.

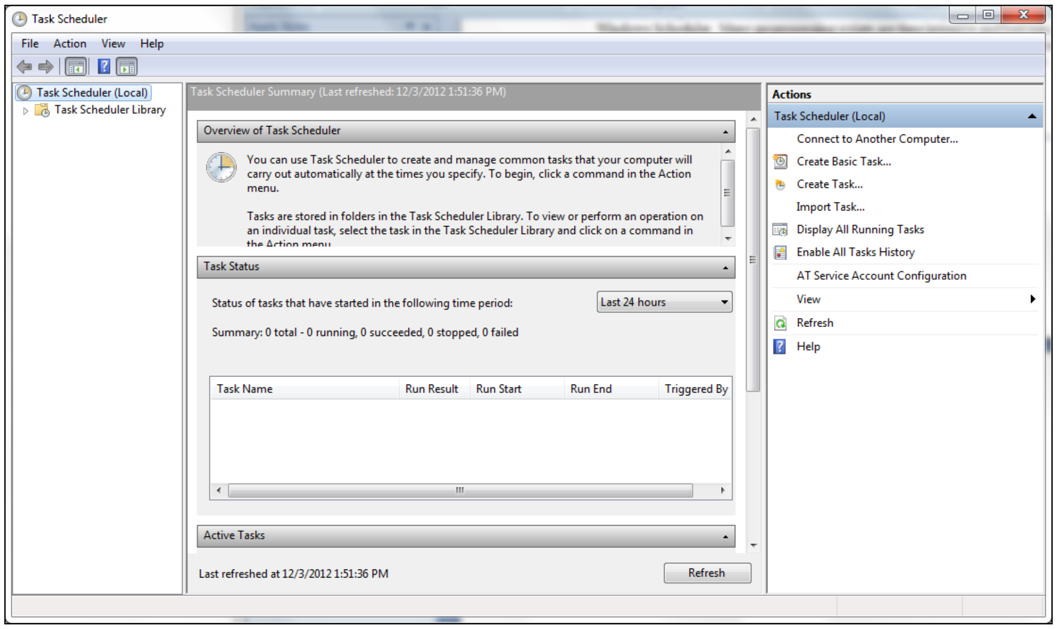
Getting ready

Many geoprocessing scripts are time-intensive and best run after hours when they can take full advantage of system resources and free up your time to concentrate on other tasks. In this recipe, you will learn how to use the Windows Scheduler to schedule the execution of your batch files.

How to do it...

Follow these steps to schedule a batch file with the Windows Scheduler:

1. Open the Windows Scheduler by going to **Start** | **Programs** | **Accessories** | **System Tools** | **Control Panel** | **Administrative Tools**. Select **Task Scheduler**. The scheduler should appear, as shown in the following screenshot:



Task Scheduler Summary (Last refreshed: 12/3/2012 1:51:36 PM)

You can use Task Scheduler to create and manage common tasks that your computer will carry out automatically at the times you specify. To begin, click a command in the Action menu.

Tasks are stored in folders in the Task Scheduler Library. To view or perform an operation on an individual task, select the task in the Task Scheduler Library and click on a command in the Action menu.

Status of tasks that have started in the following time period: Last 24 hours

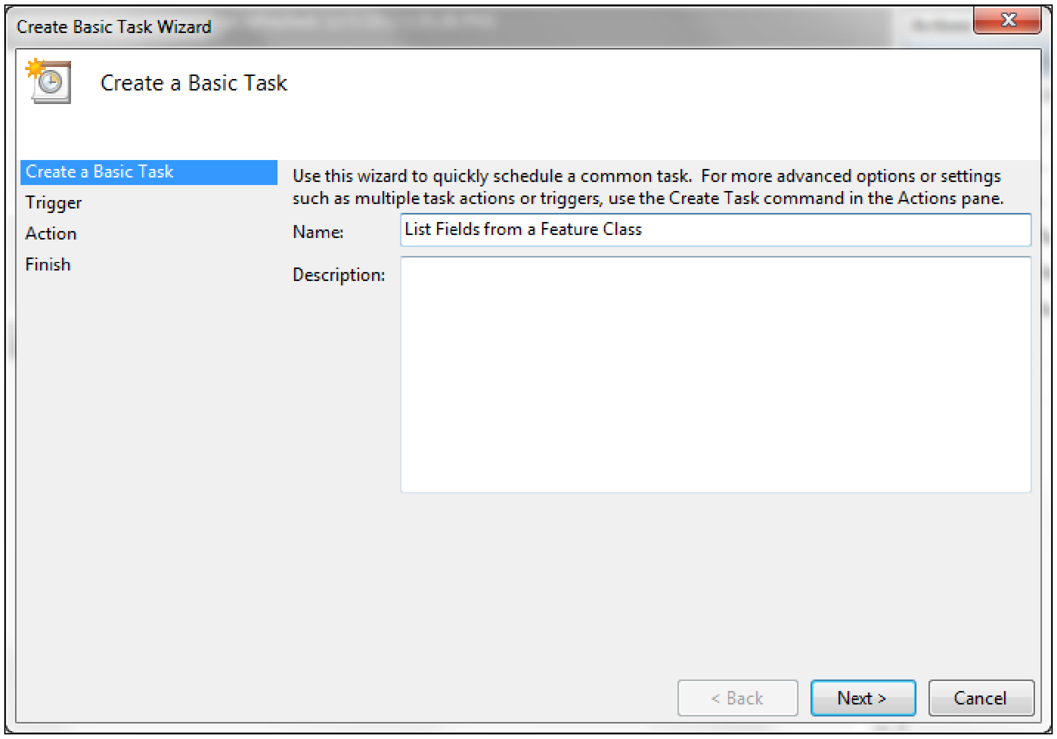
Summary: 0 total - 0 running, 0 succeeded, 0 stopped, 0 failed

Task Name	Run Result	Run Start	Run End	Triggered By

Last refreshed at 12/3/2012 1:51:36 PM

2. Select the **Action** menu item and then **Create Basic Task** to display the **Create Basic Task Wizard** dialog box, as shown in the next screenshot.

3. Give your task a name. In this case, we will call `it List Fields from a Feature Class`. Click on **Next**:

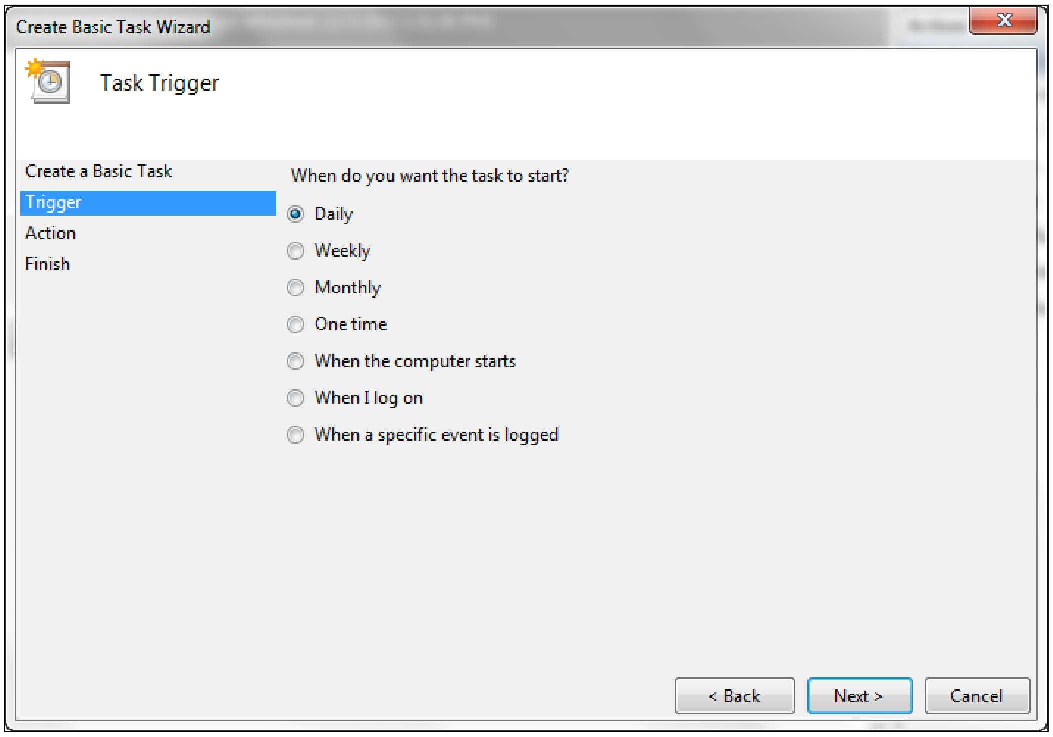


Use this wizard to quickly schedule a common task. For more advanced options or settings such as multiple task actions or triggers, use the Create Task command in the Actions pane.

Name: List Fields from a Feature Class

Description:

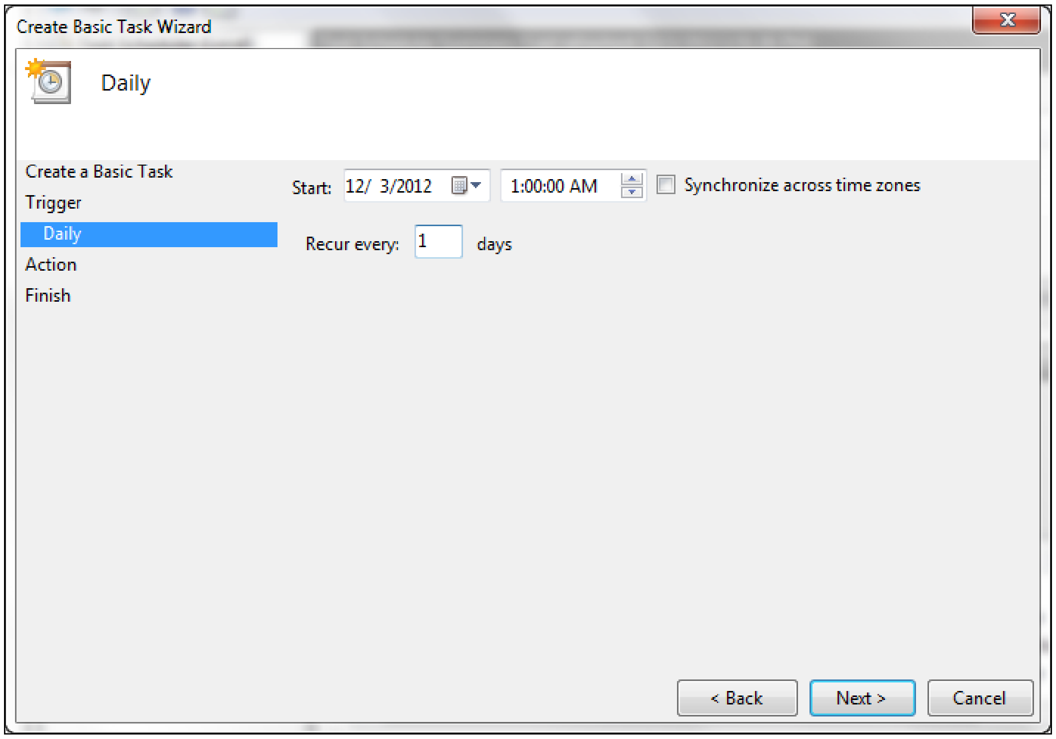
4. Select a trigger for when the task should be executed. This can, and often will be, a time-based trigger, but there can also be other types of triggers such as a user login or computer start. In this case, let's just select **Daily**. Click on **Next**:



When do you want the task to start?

- [✓ Daily]
- Weekly
- Monthly
- One time
- When the computer starts
- When I log on
- When a specific event is logged

5. Select a start date/time as well as a recurrence interval. In the following screenshot, I have selected the date as `12/3/2012`, with time as `1:00:00 AM`, and a recurrence interval of 1 day. So, every day at 1:00 AM, this task will be executed. Click on **Next**:

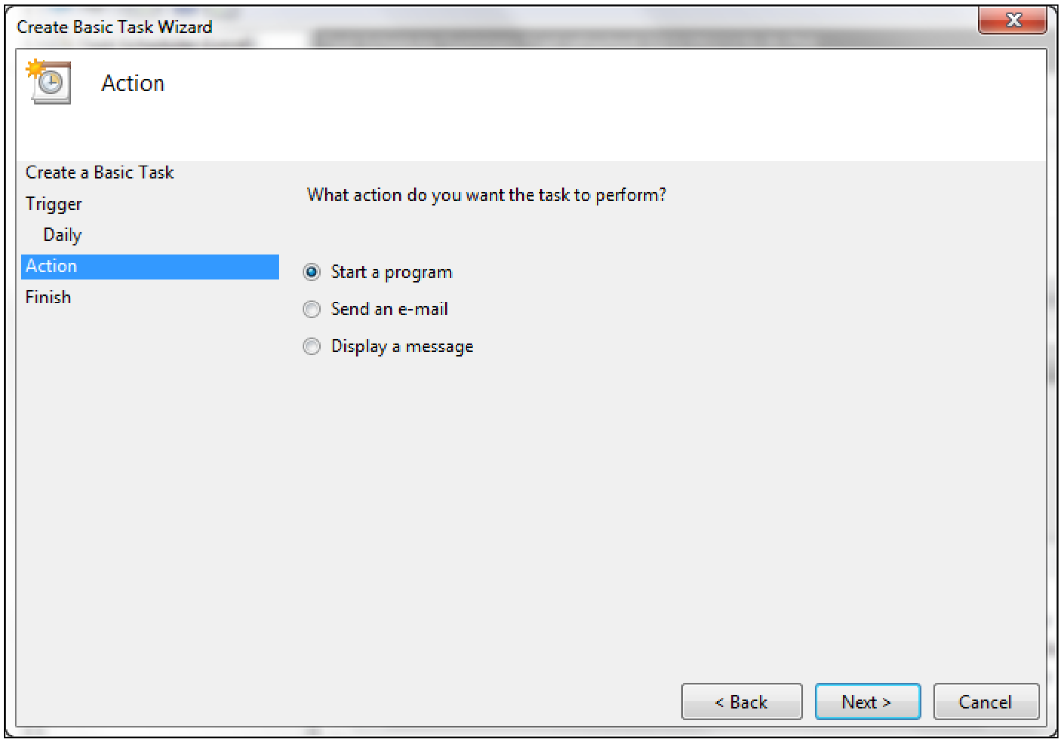

Create Basic Task Wizard

Daily

Start: 12/ 3/2012 1:00:00 AM Synchronize across time zones

Recur every: 1 days

6. Select **Start a program** as the action:



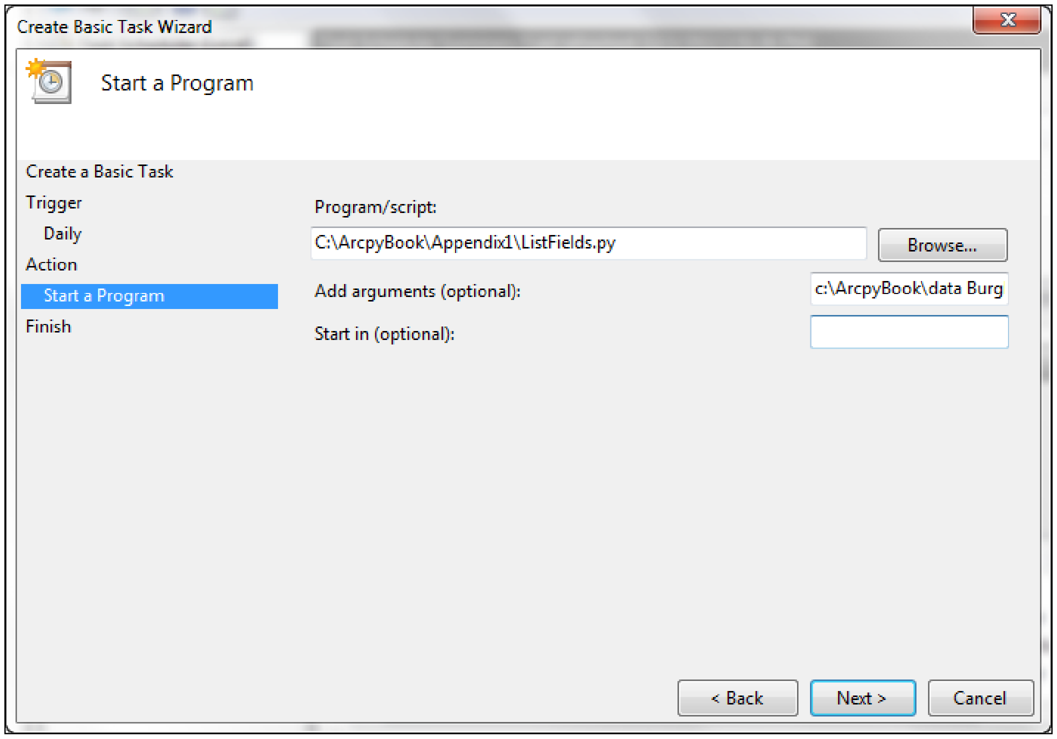
What action do you want the task to perform?

[✓ Start a program]

Send an e-mail

Display a message

7. Browse to your script and add the parameters. Click on **Next**:

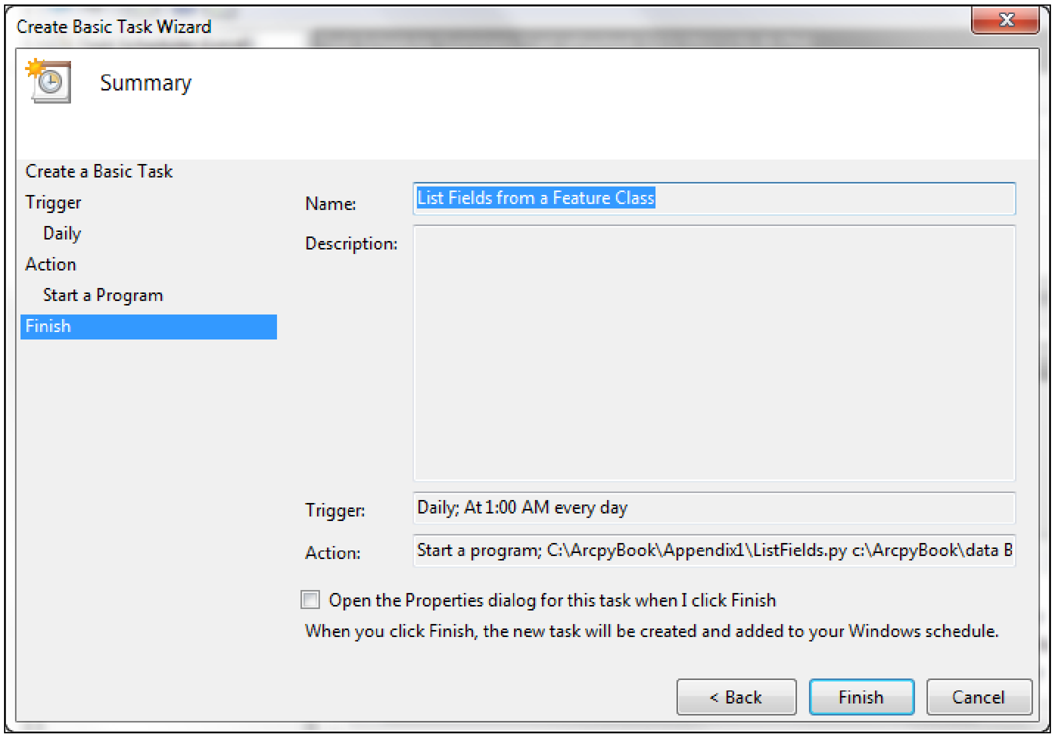


Program/script: C:\ArcpyBook\Appendix1\ListFields.py

Add arguments (optional): c:\ArcpyBook\data Burg

Start in (optional):

8. Click on **Finish** to add a task to the scheduler:



Name: List Fields from a Feature Class

Description:

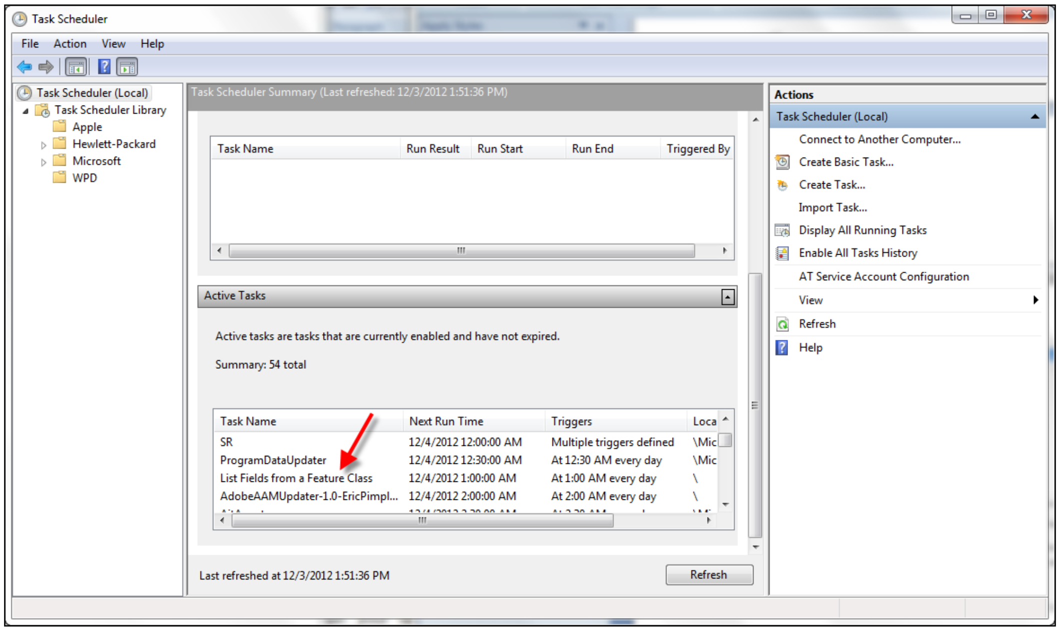
Trigger: Daily; At 1:00 AM every day

Action: Start a program; C:\ArcpyBook\Appendix1\ListFields.py c:\ArcpyBook\data B

Open the Properties dialog for this task when I click Finish

When you click Finish, the new task will be created and added to your Windows schedule.

9. The tasks should now be displayed in the list of active tasks:



Task Scheduler Summary (Last refreshed: 12/3/2012 1:51:36 PM)

Task Name	Run Result	Run Start	Run End	Triggered By

Active Tasks

Active tasks are tasks that are currently enabled and have not expired.

Summary: 54 total

Task Name	Next Run Time	Triggers	Loca
SR	12/4/2012 12:00:00 AM	Multiple triggers defined	\Mic
ProgramDataUpdater	12/4/2012 12:30:00 AM	At 12:30 AM every day	\Mic
List Fields from a Feature Class	12/4/2012 1:00:00 AM	At 1:00 AM every day	\
AdobeAAMUpdater-1.0-EricPimpl...	12/4/2012 2:00:00 AM	At 2:00 AM every day	\

Last refreshed at 12/3/2012 1:51:36 PM

How it works...

The Windows Task Scheduler keeps track of all the active tasks and handles the execution of these tasks when the prescribed trigger is fired. In this recipe, we have scheduled our task to execute each day at 1:00 AM. At that time, the batch file we created will be triggered and the arguments we specified when creating the task will be passed into the script. Using the scheduler to automatically execute geoprocessing tasks after hours without the need for GIS staff to interact with the scripts gives you more flexibility and increases your efficiency. You might also want to consider logging the errors in your Python scripts to a log file, for more information about specific problems.

B
Five Things Every GIS Programmer Should Know How to Do with Python

In this chapter, we will cover the following recipes:

- Reading data from a delimited text file
- Sending e-mails
- Retrieving files from an FTP server
- Creating ZIP files
- Reading XML files

Introduction

In this chapter, you will learn how to write scripts that perform general purpose tasks with Python. Tasks, such as reading and writing delimited text files, sending e-mails, interacting with FTP servers, creating `.zip` files, and reading and writing JSON and XML files. Every GIS programmer should know how to write Python scripts that incorporate this functionality.

Reading data from a delimited text file

File handling with Python is a very important topic for GIS programmers. Text files are often used as an interchange format for exchanging data between systems. They are simple, cross-platform, and easy to process. Comma-and tab-delimited text files are among the most commonly used formats for text files, so we'll take an extensive look at the Python tools available for processing these files. A common task for GIS programmers is to read comma-delimited text files containing x and y coordinates along with other attribute information. This information is then converted into GIS data formats, such as shapefiles or geodatabases.

Getting ready

To use Python's built-in file processing functionality, you must first open the file. Once open, data within the file is processed using functions provided by Python, and finally the file is closed. Always remember to close the file when you're done.

In this recipe, you will learn how to open, read, process, and close a comma-delimited text file.

How to do it…

Follow these steps to create a Python script that reads a comma-delimited text file:

1. In your `c:\ArcpyBook\data` folder, you will find a file called `N_America.A2007275.txt`. Open this file in a text editor. It should appear as follows:

   ```
   18.102,-94.353,310.7,1.3,1.1,10/02/2007,0420,T,72
   19.300,-89.925,313.6,1.1,1.0,10/02/2007,0420,T,82
   19.310,-89.927,309.9,1.1,1.0,10/02/2007,0420,T,68
   26.888,-101.421,307.3,2.7,1.6,10/02/2007,0425,T,53
   26.879,-101.425,306.4,2.7,1.6,10/02/2007,0425,T,45
   36.915,-97.132,342.4,1.0,1.0,10/02/2007,0425,T,100
   ```

   This file contains wildfire incident data derived from a satellite sensor from a single day in 2007. Each row contains latitude and longitude information for the fire along with additional information, including the date and time, the satellite type, confidence value, and others. In this recipe, you are going to pull out only the latitude, longitude, and confidence value.

2. Open IDLE and create a file called `c:\ArcpyBook\Appendix2\ReadDelimitedTextFile.py`.
3. Use the Python `open()` function to open the file for reading:

   ```
   f = open("c:/ArcpyBook/data/N_America.A2007275.txt','r')
   ```

4. Read the content of the text file into a list:

```
lstFires = f.readlines()
```

5. Add a `for` loop to iterate all the rows that have been read into the `lstFires` variable:

```
for fire in lstFires:
```

6. Use the `split()` function to split the values into a list using a comma as the delimiter. The list will be assigned to a variable called `lstValues`. Make sure that you indent this line of code inside the `for` loop you just created:

```
lstValues = fire.split(",")
```

7. Using the index values that reference latitude, longitude, and confidence values, create new variables:

```
latitude = float(lstValues[0])
longitude = float(lstValues[1])
confid = int(lstValues[8])
```

8. Print the values of each with the `print` statement:

```
print "The latitude is: " + str(latitude) + " The longitude is: "
+ str(longitude) + " The confidence value is: " + str(confid)
```

9. Close the file:

```
f.close()
```

10. The entire script should appear as follows:

```
f = open('c:/ArcpyBook/data/N_America.A2007275.txt','r')
lstFires = f.readlines()
for fire in lstFires:
  lstValues = fire.split(',')
  latitude = float(lstValues[0])
  longitude = float(lstValues[1])
  confid = int(lstValues[8])
  print "The latitude is: " + str(latitude) + " The longitude is:
" + str(longitude) + " The confidence value is: " + str(confid)
f.close()
```

11. Save and run the script. You should see the following output:

```
The latitude is: 18.102 The longitude is: -94.353 The confidence
value is: 72
The latitude is: 19.3 The longitude is: -89.925 The confidence
value is: 82
The latitude is: 19.31 The longitude is: -89.927 The confidence
value is: 68
```

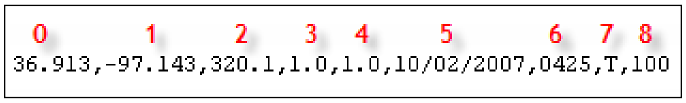
```
The latitude is: 26.888 The longitude is: -101.421 The confidence
value is: 53

The latitude is: 26.879 The longitude is: -101.425 The confidence
value is: 45

The latitude is: 36.915 The longitude is: -97.132 The confidence
value is: 100
```

How it works...

Python's `open()` function creates a file object, which serves as a link to a file residing on your computer. You must call the `open()` function on a file before reading or writing data in a file. The first parameter for the `open()` function is a path to the file you'd like to open. The second parameter of the `open()` function corresponds to a mode, which is typically read (`r`), write (`w`), or append (`a`). A value of `r` indicates that you'd like to open the file for read-only operations, while a value of `w` indicates you'd like to open the file for write operations. If the file you open in write mode already exists, it will overwrite any existing data in the file, so be careful using this mode. Append mode (`a`) will open a file for write operations, but instead of overwriting any existing data, it will append data to the end of the file. So, in this recipe, we have opened the `N_America.A2007275.txt` file in read-only mode.

The `readlines()` function then reads the entire contents of the file into a Python list, which can then be iterated. This list is stored in a variable called `lstFires`. Each row in the text file will be a unique value in the list. Since this function reads the entire file into a list, you need to use this method with caution, as large files can cause significant performance problems.

Inside the `for` loop, which is used to loop through each of the values in `lstFires`, the `split()` function is used to create a list object from a line of text that is delimited in some way. Our file is comma-delimited, so we can use `split(",")`. You can also split based on other delimiters such as tabs, spaces, or any other delimiter. This new list object created by `split()` is stored in a variable called `lstValues`. This variable contains each of the wildfire values. This is illustrated in the following screenshot. You'll notice that latitude is located in the first position, longitude is located in the second position, and so on. Lists are zero based:



```
36.913,-97.143,320.1,1.0,1.0,10/02/2007,0425,T,100
```

Using the index values (which reference latitude, longitude, and confidence values), we create new variables called `latitude`, `longitude`, and `confid`. Finally, we print each of the values. A more robust geoprocessing script might write this information into a feature class using an `InsertCursor` object.

There's more...

Just as is the case with reading files, there are a number of methods that you can use to write data to a file. The `write()` function is probably the easiest to use. It takes a single string argument and writes it to a file. The `writelines()` function can be used to write the contents of a list structure to a file. Before writing data to a text file, you will need to open the file in either a write or append mode.

Sending e-mails

There will be occasions when you may need to send an e-mail from a Python script. An example might be an alert for the successful completion or error in a long-running geoprocessing operation. On these and other occasions, sending an e-mail can be helpful.

Getting ready

Sending an e-mail through a Python script will require that you have access to a mail server. This can be a public e-mail service, such as Yahoo, Gmail, or others. It can also use outgoing mail servers configured with applications, such as Microsoft Outlook. In either case, you'll need to know the host name and port of the e-mail server. The Python `smtplib` module is used to create connections to the mail server and to send e-mails.

The Python `email` module contains a `Message` class that represents e-mail messages. Each message contains both headers and a body. This class can't be used to send e-mails; it just handles its object representation. In this recipe, you'll learn how to use the `smtp` class to send e-mails containing an attachment through your script. The `Message` class can parse a stream of characters or a file containing an e-mail using either the `message_from_file()` or `message_from_string()` functions. Both will create a new `Message` object. The body of the mail can be obtained by calling `Message.getpayload()`.

> We are using the Google Mail service for this exercise. If you already have a Gmail account, then simply provide your username and password as the values for these variables. If you don't have a Gmail account, you'll need to create one or use a different mail service to complete this exercise; Gmail accounts are free.

How to do it...

Follow these steps to create a script that can send emails:

1. Open IDLE and create a file called `c:\ArcpyBook\Appendix2\SendEmail.py`.
2. In order to send e-mails with attachments, you're going to need to import the `smtplib` module along with the `os` module, and several classes from the e-mail module. Add the following `import` statements to your script:

   ```
   import smtplib
   from email.MIMEMultipart import MIMEMultipart
   from email.MIMEBase import MIMEBase
   from email.MIMEText import MIMEText
   from email import Encoders
   import os
   ```

3. Create the following variables and assign your Gmail username and password as the values. Do keep in mind that this method of e-mailing from your Python script can invite problems, as it requires that you include your username and password:

   ```
   gmail_user = "<username>"
   gmail_pwd = "<password>"
   ```

4. Create a new Python function called `mail()`. This function will accept four parameters: `to`, `subject`, `text`, and `attach`. Each of these parameters should be self-explanatory. Create a new `MIMEMultipart` object and assign the `from`, `to`, and `subject` keys. You can also attach the text of the e-mail to this new `msg` object using `MIMEMultipart.attach()`:

   ```
   def mail(to, subject, text, attach):
     msg = MIMEMultipart()
     msg['From'] = gmail_user
     msg['To'] = to
     msg['Subject'] = subject

     msg.attach(MIMEText(text))
   ```

5. Attach the file to the e-mail:

   ```
     part = MIMEBase('application', 'octet-stream')
     part.set_payload(open(attach, 'rb').read())
     Encoders.encode_base64(part)
     part.add_header('Content-Disposition',
        'attachment; filename="%s"' % os.path.basename(attach))
     msg.attach(part)
   ```

6. Create a new SMTP object that references the Google Mail service, passes in the username and password to connect to the mail services, sends the e-mail, and closes the connection:

   ```
   mailServer = smtplib.SMTP("smtp.gmail.com", 587)
   mailServer.ehlo()
   mailServer.starttls()
   mailServer.ehlo()
   mailServer.login(gmail_user, gmail_pwd)
   mailServer.sendmail(gmail_user, to, msg.as_string())
   mailServer.close()
   ```

7. Call the `mail()` function, passing in the recipient of the e-mail, a subject for the e-mail, the text of the e-mail, and the attachment:

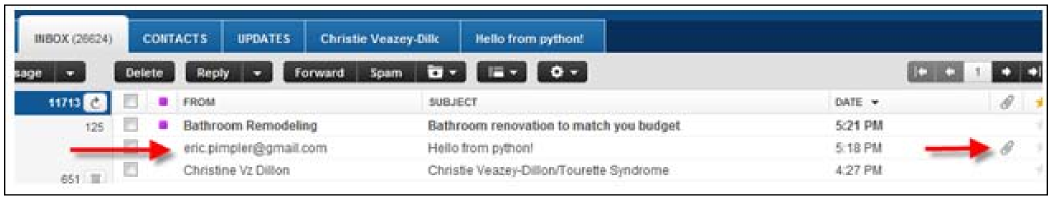
   ```
   mail("<email to send to>",
   "Hello from python!",
   "This is an email sent with python",
   "c:/ArcpyBook/data/bc_pop1996.csv")
   ```

8. The entire script should appear as follows:

```
import smtplib
from email.MIMEMultipart import MIMEMultipart
from email.MIMEBase import MIMEBase
from email.MIMEText import MIMEText
from email import Encoders
import os

gmail_user = "<username>"
gmail_pwd = "<password>"

def mail(to, subject, text, attach):
 msg = MIMEMultipart()

 msg['From'] = gmail_user
 msg['To'] = to
 msg['Subject'] = subject

 msg.attach(MIMEText(text))

 part = MIMEBase('application', 'octet-stream')
 part.set_payload(open(attach, 'rb').read())
 Encoders.encode_base64(part)
```

```
 part.add_header('Content-Disposition',
     'attachment; filename="%s"' % os.path.basename(attach))
 msg.attach(part)

 mailServer = smtplib.SMTP("smtp.gmail.com", 587)
 mailServer.ehlo()
 mailServer.starttls()
 mailServer.ehlo()
 mailServer.login(gmail_user, gmail_pwd)
 mailServer.sendmail(gmail_user, to, msg.as_string())
 mailServer.close()

mail("<email to send to>",
 "Hello from python!",
 "This is an email sent with python",
 "bc_pop1996.csv")
```

9. Save and run the script. For testing, I used my personal Yahoo account as the recipient. You'll notice that my inbox has a new message from my Gmail account; also notice the attachment:



FROM	SUBJECT	DATE
Bathroom Remodeling	Bathroom renovation to match you budget	5:21 PM
eric.pimpler@gmail.com	Hello from python!	5:18 PM
Christine Vz Dillon	Christie Veazey-Dillon/Tourette Syndrome	4:27 PM

How it works...

The first parameter passed into the `mail()` function is the e-mail address that will receive the e-mail. This can be any valid e-mail address, but you'll want to supply a mail account that you can actually check, so that you can make sure your script runs correctly. The second parameter is just the subject line of the e-mail. The third parameter is the text of the e-mail. The final parameter is the name of a file that will be attached to the e-mail. Here, I've simply defined that the `bc_pop1996.csv` file should be attached. You can use any file you have access to, but you may want to just use this file for testing.

We then create a new `MIMEMultipart` object inside the `mail()` function, and assign the `from`, `to`, and `subject` keys. You can also attach the text of the e-mail to this new `msg` object using `MIMEMultipart.attach()`. The `bc_pop1996.csv` file is then attached to the e-mail using a `MIMEBase` object and attached to the e-mail using `msg.attach(part)`.

At this point, we've examined how a basic text e-mail can be sent. However, we want to send a more complex e-mail message that contains text and an attachment. This requires the use of MIME messages, which provides the functionality to handle multi-part e-mails. MIME messages need boundaries between the multiple parts, along with extra headers to specify the content being sent. The `MIMEBase` class is an abstract subclass of `Message` and enables this type of an e-mail to be sent. Because it is an abstract class, you can't create actual instances of this class. Instead, you use one of the subclasses, such as `MIMEText`. The last step of the `mail()` function is to create a new SMTP object that references the Google Mail service, passes in the username and password to connect to the mail services, sends the e-mail, and closes the connection.

Retrieving files from an FTP server

Retrieving files from an FTP server for processing is a very common operation for GIS programmers and can be automated with a Python script.

Getting ready

Connecting to an FTP server and downloading a file is accomplished through the `ftplib` module. A connection to an FTP server is created through the FTP object, which accepts a host, username, and password to create the connection. Once a connection has been opened, you can then search for and download files.

In this recipe, you will connect to the National Interagency Fire Center Incident FTP site and download a Google Earth format file for a wildfire in Alaska.

How to do it...

Follow these steps to create a script that connects to an FTP server and downloads a file:

1. Open IDLE and create a file called `c:\ArcpyBook\Appendix2\ftp.py`.
2. We'll be connecting to an FTP server at the NIFC. Visit their website at `http://ftpinfo.nifc.gov/` for more information.
3. Import the `ftplib`, `os`, and socket modules:

   ```
   import ftplib
   import os
   import socket
   ```

4. Add the following variables that define the URL, directory, and filename:

   ```
   HOST = 'ftp.nifc.gov'
   DIRN = '/Incident_Specific_Data/ALASKA/Fire_
   Perimeters/20090805_1500'
   FILE = 'FirePerimeters_20090805_1500.kmz'
   ```

5. Add the following code block to create a connection. If there is a connection error, a message will be generated. If the connection was successful, a success message will be printed:

```
try:
  f = ftplib.FTP(HOST)
except (socket.error, socket.gaierror), e:
  print 'ERROR: cannot reach "%s"' % HOST
print '*** Connected to host "%s"' % HOST
```

6. Add the following code block to anonymously log in to the server:

```
try:
  f.login()
except ftplib.error_perm:
  print 'ERROR: cannot login anonymously'
  f.quit()
print '*** Logged in as "anonymous"'
```

7. Add the following code block to change to the directory specified in our `DIRN` variable:

```
try:
  f.cwd(DIRN)
except ftplib.error_perm:
  print 'ERROR: cannot CD to "%s"' % DIRN
  f.quit()
print '*** Changed to "%s" folder' % DIRN
```

8. Use the `FTP.retrbinary()` function to retrieve the KMZ file:

```
try:
  f.retrbinary('RETR %s' % FILE,
     open(FILE, 'wb').write)
except ftplib.error_perm:
  print 'ERROR: cannot read file "%s"' % FILE
  os.unlink(FILE)
else:
  print '*** Downloaded "%s" to CWD' % FILE
```

9. Make sure you disconnect from the server:

```
f.quit()
```

10. The entire script should appear as follows:

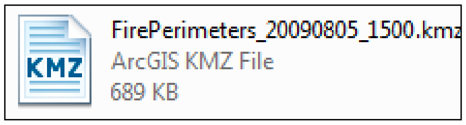
```
import ftplib
import os
import socket
```

```
HOST = 'ftp.nifc.gov'
DIRN = '/Incident_Specific_Data/ALASKA/Fire_
Perimeters/20090805_1500'
FILE = 'FirePerimeters_20090805_1500.kmz'

try:
  f = ftplib.FTP(HOST)
except (socket.error, socket.gaierror), e:
  print 'ERROR: cannot reach "%s"' % HOST
print '*** Connected to host "%s"' % HOST

try:
  f.login()
except ftplib.error_perm:
  print 'ERROR: cannot login anonymously'
  f.quit()
print '*** Logged in as "anonymous"'

try:
  f.cwd(DIRN)
except ftplib.error_perm:
  print 'ERROR: cannot CD to "%s"' % DIRN
  f.quit()
print '*** Changed to "%s" folder' % DIRN

try:
  f.retrbinary('RETR %s' % FILE,
     open(FILE, 'wb').write)
except ftplib.error_perm:
  print 'ERROR: cannot read file "%s"' % FILE
  os.unlink(FILE)
else:
  print '*** Downloaded "%s" to CWD' % FILE
f.quit()
```

11. Save and run the script. If everything is successful, you should see the following output:

```
*** Connected to host "ftp.nifc.gov"

*** Logged in as "anonymous"

*** Changed to "/Incident_Specific_Data/ALASKA/Fire_
Perimeters/20090805_1500" folder

*** Downloaded "FirePerimeters_20090805_1500.kmz" to CWD
```

12. Check your `c:\ArcpyBook\Appendix2` directory for the file. By default, FTP will download files to the current working directory:


FirePerimeters_20090805_1500.kmz
ArcGIS KMZ File
689 KB

How it works...

To connect to an FTP server, you need to know the URL. You also need to know the directory and filename for the file that will be downloaded. In this script, we have hardcoded this information, so that you can focus on implementing the FTP-specific functionality. Using this information we then created a connection to the NIFC FTP server. This is done through the `ftplib.FTP()` function, which accepts a URL to the host.

Anonymous logins are accepted by the `nifc.gov` server, so we connect to the server in this manner. Keep in mind that if a server does not accept anonymous connections, you'll need to obtain a username/password. Once logged in, the script then changes directories from the root of the FTP server to the path defined in the `DIRN` variable. This was accomplished with the `cwd(<path>)` function. The `kmz` file was retrieved using the `retrbinary()` function. Finally, you will want to close your connection to the FTP server when you're done. This is done with the `quit()` method.

There's more...

There are a number of additional FTP-related methods that you can use to perform various actions. Generally, these can be divided into directory-level operations and file-level operations. Directory-level methods include the `dir()` method to obtain a list of files in a directory, `mkd()` to create a new directory, `pwd()` to get the current working directory, and `cwd()` to change the current directory.

The `ftplib` module also includes various methods for working with files. You can upload and download files in binary or plain text format. The `retrbinary()` and `storbinary()` methods are used to retrieve and store binary files, respectively. Plain text files can be retrieved and stored using `retrlines()` and `storlines()`.

There are several others methods on the FTP class that you should be aware of. Deleting a file can be done with the `delete()` method, while renaming a file can be accomplished with `rename()`. You can also send commands to the FTP server through the `sendcmd()` method.

Creating ZIP files

GIS often requires the use of large files that will be compressed into a `.zip` format for ease of sharing. Python includes a module that you can use to decompress and compress files in this format.

Getting ready

Zip is a common compression and archive format and is implemented in Python through the `zipfile` module. The `ZipFile` class can be used to create, read, and write `.zip` files. To create a new `.zip` file, simply provide the filename along with a mode such as `w`, which indicates that you want to write data to the file. In the following code example, we are creating a `.zip` file called `datafile.zip`. The second parameter, `w`, indicates that a new file will be created. A new file will be created or an existing file with the same name will be truncated in the write mode. An optional compression parameter can also be used when creating the file. This value can be set to either `ZIP_STORED` or `ZIP_DEFLATED`:

```
zipfile.ZipFile('dataFile.zip', 'w',zipfile.ZIP_STORED)
```

In this exercise, you will use Python to create file, add files, and apply compression to a `.zip`. You'll be archiving all the shapefiles located in the `c:\ArcpyBook\data` directory.

How to do it...

Follow these steps to learn how to create a script that build a `.zip` file:

1. Open IDLE and create a script called `c:\ArcpyBook\Appendix2\CreateZipfile.py`.
2. Import the `zipfile` and `os` modules:

   ```
   import os
   import zipfile
   ```

3. Create a new `.zip` file called `shapefiles.zip` in write mode and add a compression parameter:

   ```
   zfile = zipfile.ZipFile("shapefiles.zip", "w", zipfile.ZIP_STORED)
   ```

4. Next, we'll use the `os.listdir()` function to create a list of files in the data directory:

   ```
   files = os.listdir("c:/ArcpyBook/data")
   ```

5. Loop through a list of all the files and write to the `.zip` file, if the file ends with `shp`, `dbf`, or `shx`:

   ```
   for f in files:
     if f.endswith("shp") or f.endswith("dbf") or f.endswith("shx"):
       zfile.write("C:/ArcpyBook/data/" + f)
   ```

6. Print out a list of all the files that were added to the zip archive. You can use the ZipFile.namelist() function to create a list of files in the archive:

   ```
   for f in zfile.namelist():
       print "Added %s" % f
   ```

7. Close the `.zip` archive:

   ```
   zfile.close()
   ```

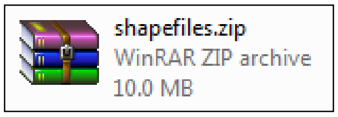
8. The entire script should appear as follows:

   ```
   import os
   import zipfile

   #create the zip file
   zfile = zipfile.ZipFile("shapefiles.zip", "w", zipfile.ZIP_STORED)
   files = os.listdir("c:/ArcpyBook/data")

   for f in files:
     if f.endswith("shp") or f.endswith("dbf") or f.endswith("shx"):
       zfile.write("C:/ArcpyBook/data/" + f)

   #list files in the archive
   for f in zfile.namelist():
       print "Added %s" % f

   zfile.close()
   ```

9. Save and run the script. You should see the following output:

   ```
   Added ArcpyBook/data/Burglaries_2009.dbf
   Added ArcpyBook/data/Burglaries_2009.shp
   Added ArcpyBook/data/Burglaries_2009.shx
   Added ArcpyBook/data/Streams.dbf
   Added ArcpyBook/data/Streams.shp
   Added ArcpyBook/data/Streams.shx
   ```

10. In Windows Explorer, you should be able to see the output `.zip` file as shown in the following screenshot. Note the size of the archive. This file was created without compression:

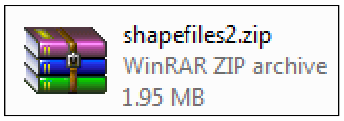


shapefiles.zip
WinRAR ZIP archive
10.0 MB

11. Now, we're going to create a compressed version of the `.zip` file to see the difference. Make the following changes to the line of code that creates the `.zip` file:

```
zfile = zipfile.ZipFile("shapefiles2.zip", "w", zipfile.ZIP_
DEFLATED)
```

12. Save and re-run the script.
13. Take a look at the size of the new `shapefiles2.zip` file that you just created. Note the decreased size of the file due to compression:



shapefiles2.zip
WinRAR ZIP archive
1.95 MB

How it works...

In this recipe, you created a new `.zip` file called `shapefiles.zip` in write mode. In the first iteration of this script, you didn't compress the contents of the file. However, in the second iteration, you did by using the `DEFLATED` parameter passed into the constructor for the `ZipFile` object. The script then obtained a list of files in the data directory and looped through each of the files. Each file that has an extension of `.shp`, `.dbf`, or `.shx` is then written to the archive file using the `write()` function. Finally, the names of each of the files written to the archive is printed to the screen.

There's more...

The contents of an existing file stored in a ZIP archive can be read using the `read()` method. The file should first be opened in a read mode, and then you can call the `read()` method passing in a parameter that represents the filename that should be read. The contents of the file can then be printed to the screen, written to another file, or stored as a list or dictionary variable.

Reading XML files

XML files were designed as a way to transport and store data. They are platform independent, since the data is stored in a plain text file. Although similar to HTML, XML differs in that HTML is designed for display purposes, whereas XML data is designed for data. XML files are sometimes used as an interchange format for GIS data going between various software systems.

Getting ready

XML documents have a tree-like structure composed of a root element, child elements, and element attributes. Elements are also called **nodes**. All XML files contain a **root** element. This root element is the parent to all other elements or child nodes. The following code example illustrates the structure of an XML document. Unlike HTML files, XML files are case sensitive:

```
<root>
 <child att="value">
 <subchild>.....</subchild>
 </child>
</root>
```

Python provides several programming modules that you can use to process XML files. The module that you use should be determined by the module that is right for the job. Don't try to force a single module to do everything. Each module has specific functions that they are good at performing. In this recipe, you will learn how to read data from an XML file using the `nodes` and `element` attributes that are a part of the document.

There are a number of ways that you can access nodes within an XML document. Perhaps, the easiest way to do so is to find nodes by tag name and then walk the tree containing a list of the child nodes. Before doing so, you'll want to parse the XML document with the `minidom.parse()` method. Once parsed, you can then use the `childNodes` attribute to obtain a list of all child nodes starting at the root of the tree. Finally, you can search the nodes by tag name with the `getElementsByTagName(tag)` function, which accepts a tag name as an argument. This will return a list of all child nodes associated with the tag.

You can also determine if a node contains an attribute by calling `hasAttribute(name)`, which will return a `true/false` value. Once you've determined that an attribute exists, a call to `getAttribute(name)` will obtain the value for the attribute.

In this exercise, you will parse an XML file and pull out values associated with a particular element (node) and attribute. We'll be loading an XML file containing wildfire data. In this file, we'll be looking for the `<fire>` node and the `address` attribute from each of these nodes. The addresses will be printed out.

How to do it...

1. Open IDLE and create a script called `c:\ArcpyBook\Appendix2\XMLAccessElementAttribute.py`.
2. The `WitchFireResidenceDestroyed.xml` file will be used. The file is located in your `c:\ArcpyBook\Appendix2` folder. You can see a sample of its contents as follows:

```
<fires>
  <fire address="11389 Pajaro Way" city="San Diego"
state="CA" zip="92127" country="USA" latitude="33.037187"
longitude="-117.082299" />
  <fire address="18157 Valladares Dr" city="San Diego"
state="CA" zip="92127" country="USA" latitude="33.039406"
longitude="-117.076344" />
  <fire address="11691 Agreste Pl" city="San Diego"
state="CA" zip="92127" country="USA" latitude="33.036575"
longitude="-117.077702" />
  <fire address="18055 Polvera Way" city="San Diego"
state="CA" zip="92128" country="USA" latitude="33.044726"
longitude="-117.057649" />
</fires>
```

3. Import `minidom` from `xml.dom`:

```
from xml.dom import minidom
```

4. Parse the XML file:

```
xmldoc = minidom.parse("WitchFireResidenceDestroyed.xml")
```

5. Generate a list of nodes from the XML file:

```
childNodes = xmldoc.childNodes
```

6. Generate a list of all the `<fire>` nodes:

```
eList = childNodes[0].getElementsByTagName("fire")
```

7. Loop through the list of elements, test for the existence of the `address` attribute, and print the value of the attribute if it exists:

```
for e in eList:
  if e.hasAttribute("address"):
    print e.getAttribute("address")
```

8. Save and run the script. You should see the following output:

```
11389 Pajaro Way
18157 Valladares Dr
11691 Agreste Pl
18055 Polvera Way
18829 Bernardo Trails Dr
18189 Chretien Ct
17837 Corazon Pl
18187 Valladares Dr
18658 Locksley St
18560 Lancashire Way
```

How it works...

Loading an XML document into your script is probably the most basic thing you can do with XML files. You can use the `xml.dom` module to do this through the use of the `minidom` object. The `minidom` object has a method called `parse()`, which accepts a path to an XML document and creates a **document object model** (**DOM**) tree object from the `WitchFireResidenceDestroyed.xml` file.

The `childNodes` property of the DOM tree generates a list of all nodes in the XML file. You can then access each of the nodes using the `getElementsByTagName()` method. The final step is to loop through each of the `<fire>` nodes contained within the `eList` variable. For each node, we then check for the `address` attribute with the `hasAttribute()` method, and if it exists we call the `getAttribute()` function and print the address to the screen.

There's more...

There will be times when you will need to search an XML document for a specific text string. This requires the use of the `xml.parsers.expat` module. You'll need to define a search class derived from the basic `expat` class and then create an object from this class. Once created, you can call the `parse()` method on the search object to search for data. Finally, you can then search the nodes by tag name with the `getElementsByTagName(tag)` function, which accepts a tag name as an argument. This will return a list of all child nodes associated with the tag.

Index