A Python Primer for ArcGIS®

Workbook I

Nathan Jennings

Copyright © 2015 Nathan Jennings
All rights reserved.
ISBN: 1505893321
ISBN-13: 978-1505893328

ACKNOWLEDGEMENTS **7**

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

WORKBOOK I: THE FUNDAMENTALS **21**

Chapter 1 Python and ArcGIS **23**
Overview 23
Python and ArcGIS Versions 24
How Python is used with ArcGIS 24
Python Development Environments 25
Relationship to ModelBuilder 27
Python Shell in ArcGIS 27
Use of Scripts with Geoprocessing Tools 29
Getting Help 30
ArcGIS 30
Python 31
Python and ArGIS Errors 31
Python Syntax Errors 31
ArcGIS Error Codes 32
Common Methods for Handling Errors 34

Chapter 2 ModelBuilder and Python **35**
Overview of ModelBuilder 35
ModelBuilder Python Script Caveats 36

Chapter 2 Demos **39**

Demo 2a: Using ModelBuilder to Create a Python Script: the Preferred Way **41**

Demo 2b: Using ModelBuilder to Create a Python Script: the Non-Preferred Way **53**

Exercise 2 - Create a New Feature Class and Add Fields **57**

Chapter 2 Questions **59**

Chapter 3 Python and ArcGIS Constructs **61**
Overview 61
Using Python IDLE for Code Development 61
Using the Python Shell for Code Testing 63
Syntax 64
Case Sensitivity 64
Naming Conventions 64
Indentation 65
Comments 66
Creating and Using Variables 68
String and Number Variables 70
Strings, Data Path, and Workspace Conventions 71
Lists 74
Conditional Statements and Loops 75
import Modules 76
try: and except: Blocks 77
Special Considerations for Query Strings in Python 78
Single and Double Quotes 78
"Triple" Double Quotes 78
General Structure of a Useful Python Script 80
Title, Author, Date, and Script Comments 82
import Modules 82
Variable Definitions (and Python Function Definitions) 82
Code Body 82
Running a Python Script 83
Check Module 83
Run Module 84
Handling Errors 85
Summary 86

Exercise 3 - Write a Simple Python Script **87**

Chapter 3 Questions **91**

Chapter 4 Writing a Basic Geoprocessing Python Script **93**
Overview 93
Getting Ready to Create an ArcGIS Geoprocessing Python Script 93
Using Pseudo-code to Outline Geoprocessing Tasks 93
arcpy Module Overview 95
Workspace Definitions and Data Path Variables 96
Alternative Method for Workspaces Definitions and Data Paths 97
Define Variables 98
Hard Coded Parameters 98
Parameters Using Variables 99
Add and Modify Geoprocessing Functions 99
Search ArcGIS Help 100
ArcGIS Toolbox Aliases 100
Summary 101

Chapter 4 Demo Writing a Clip Features Script ... 103

Exercise 4 - Add the Buffer Routine to the Clip Features Script ... 111

Chapter 4 Questions ... 113

ACCESSING DATA, DEMOS, AND CODE ... 115

REFERENCES ... 116

INDEX ... 117

Acknowledgements

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

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

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

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

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

Introduction

For the last several years, Esri has supported the use of the 'open-source scripting language' Python for many of its geoprocessing tools and functions within ArcGIS. Python is 'platform independent', so it serves as a good single common scripting language for different operating systems as well as for different versions of ArcGIS. As ArcGIS development moves forward, organizations and individuals will not need to maintain geoprocesses using outdated multiple scripting languages, such as Arc Macro Language (AML™) and Avenue™, neither of which are officially supported any longer.

Professionals in Geographic Information Systems (GIS) and newcomers to GIS will want, and need, to learn Python. Knowing how to program in Python will be beneficial to their careers. The same will be true of organizations that have a long history of scripting development and that wish to transition to more current geoprocessing standards.

Objectives and Goals

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

The workbooks accomplish the following objectives:

1. Provides a framework for code developers of different skill sets, to design logical geoprocesses.
2. Teaches how to design logical coding structures that include proper constructs for

   - error handling,
   - troubleshooting processes,
   - logic, and
   - scripting problems.

3. Introduces common Python constructs, illustrating how they are implemented with ArcGIS geoprocessing tools.
4. Teaches code developers how to obtain help with Python and ArcGIS geoprocessing functions, in the process of building their own code writing skill.
5. Introduces some of the new functionality of Python and ArcGIS, such as the mapping and data access modules.
6. Shows how to integrate custom-built scripts with the ArcToolbox™
7. Shows how to make and auto-run custom scripts.

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

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

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

Structure of the Workbooks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The author uses the following typeface conventions throughout the book:

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

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

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

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

Data and Demos

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

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

City of Sacramento – city related vector data and historical 1991 aerial photos
County of Sacramento – parcel and street subsets
CalAtlas – Landsat Thematic Mapper (TM) satellite imagery subset

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

Accessing the Data, Demos, and Code

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

Required Software

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

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

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

Older Versions of ArcGIS and Python

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

Reporting Errata

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

Prerequisite Knowledge and Skill

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

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

Problem Solving

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

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

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

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

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

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

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

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

Developing Geoprocessing Workflows

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

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

The workflow can take many forms, such as an outline of steps or a workflow diagram indicating the relationships between one step and another or how one step may be related to many steps. For example, one source dataset may be used in multiple geoprocesses. This kind of workflow is often seen when designing a geoprocessing model in ModelBuilder. The workflow can be fairly simple, involving a small number of geoprocessing tasks. Conversely, it can be complex, involving many data sources, processing steps, feedback (looping) mechanisms, and many outputs (geospatial data, tables, maps, web services, etc.).
The following code represents an example of an actual outline and an actual script developed by the author for a specific task. Notice the comments (marked with a # sign) that briefly describe the specific geoprocessing task. The commented steps (sometimes referred to as "pseudo-code") provide the framework for the script. The comments were actually written first before any specific ArcGIS geoprocesses were created. The author could outline the general set of steps and thereafter, could determine if other geoprocessing steps were required. These, then, could be researched for syntax, parameter, data, and data type requirements. The development of the script through this iterative process then informs and modifies the outline.

```
#1. Access a table using a search cursor
#   sort the data in ascending order based
#   on the Code attribute

srows = arcpy.SearchCursor(parts_table, "", "", "", 'Code A')

irows = arcpy.InsertCursor(sorted_parts_table)

#2. Update the "sorted parts table with the sorted
#   records from the existing table

print "Sorting Parts Table..."
print >> log, "Sorting Parts Table..."

for srow in srows:

    irow = irows.NewRow()
    irow.Code = srow.Code
    irow.Description = srow.Description

    irows.InsertRow(irow)

#3. Remove existing attribute domain from field


print "Updating Domain for " + signs_fc + "..."
print >> log, "Updating Domain for " + signs_fc + "..."
```

As another example, the figure below shows a diagram developed for a geoprocessing workflow to perform data maintenance on traffic signs for the City of Sacramento. This workflow is used by GIS staff in the city's department of transportation. A GIS data management document accompanies the workflow that provides specific GIS data processing tasks that the staff uses to update and manage the city's traffic sign inventory. The city's GIS staff uses this workflow to discuss similar activities for other departments. A full discussion of the workflow and geoprocesses implementation can be found in the Winter 2009 issue of *ArcUser*™ (**http://www.Esri.com/news/arcuser/0109/streetsigns.html**).

Both examples illustrate general processes for outlining and developing documentation. Get in the habit of making one or the other—or both. Good commenting and diagrams can serve a wide variety of purposes.

Workbook I: The Fundamentals

Workbook I introduces the Python scripting language and how it relates to ArcGIS.

Chapter 1 focuses on how Python can be used with ArcGIS and its relationship with ModelBuilder, since some readers may already have experience with ModelBuilder. Chapter 1 also introduces the Python script Interactive Development Environment (IDE), called IDLE, so that the reader has a basic understanding of where to write actual Python script. The reader is provided a high level overview of how to obtain help with both Python and ArcGIS as well identifying errors that are likely to occur when developing Python code.

Chapter 2 focuses on ModelBuilder and provides a general overview of how ModelBuilder operates and how it can be used to develop geoprocessing logic and can ultimately be exported to Python script where it can be more fully developed.

Chapter 3 reviews the primary Python constructs that will be used throughout *A Python Primer for ArcGIS*. This chapter introduces these concepts at a broad level where many of them will be more fully discussed in the context of their use with ArcGIS geoprocessing tasks and functions.

Chapter 4 covers the fundamental concepts of writing a geoprocessing Python script using ArcGIS. The chapter discusses the required Python and arcpy modules for developing a geoprocessing script as well as reviewing some of the important topics as defining variables, workspaces, and data paths. The author also provides some guidance to obtain additional help so the programmer can become more self sufficient as a programmer as he or she continues to develop their programming skills.

At the end of Chapters 3 and 4, the reader has a chance to write a simple Python script as well as a simple geoprocessing script that will serve as the launch point to develop more complex geoprocessing routines using Python and ArcGIS.

Chapter 1 Python and ArcGIS

Overview

ArcGIS uses Python in several ways.

1. *Python Window* - to write Python code and run geoprocessing routines from within ArcMap or ArcCatalog
2. *Custom or Python Script Tools* – look and function similar to conventional geoprocessing routines within ArcToolbox
3. *Python Plugin* – provides some simple tools and user interfaces to run Python code
4. *Standalone Scripts* – develop custom, multi-step geoprocessing functionality to automate routine and iterative tasks.

A Python Primer for ArcGIS focuses on the last situation. Being able to understand the Python fundamentals, processing logic, pre-requisite routines, trouble shooting, error handling, and code refinement are keys to creating successful geoprocessing routines in any of the above options. Without mastering the fundamentals, any of the Python methods will be difficult to successfully develop. *A Python Primer for ArcGIS* covers many of the fundamental geprocessing tasks that many GIS professionals and users of ArcGIS encounter. Many of the concepts and examples discussed throughout this book can be applied and developed into more complex Python scripts.

The ArcGIS Help makes specific reference to two different uses of Python scripts:

1. *Python Script Tool* – this is a script that is written with the intent to be used in a custom ArcToolbox and used within an open instance of ArcMap or ArcCatalog.

2. *Standalone Python Script* - this is a script that is written with the intent to be run or executed outside of an open instance of ArcMap or ArcCatalog and may be used in a Windows scheduler program to have the script run automatically without user involvement.

The author makes reference of this distinction because it will impact how some Python scripts are written and error handling is developed. Throughout the workbooks the reader will find commentary on certain scripting methods that can be used to develop scripts for the ArcGIS Toolbox. *Workbook III*, Chapter 10 is devoted to the development of a custom ArcToolbox, developing a Python script to accept user input, binding the Python script to the custom tool, and creating help documents for the custom tool. Creating standalone programs throughout the book provides the opportunity for the developer to obtain the full experience of

- developing a geoprocessing task strategy,
- logically designing and writing the correct syntax to perform the geoprocessing routines and tasks, and
- solving syntax and scripting logic problems that arise in almost any geoprocessing workflow or script.

Python and ArcGIS Versions

Certain versions of Python work with certain versions of ArcGIS and may affect how Python code is structured and implemented. Some of the methods are processed differently depending on the version of Python and ArcGIS. Scripts developed for older versions of ArcGIS can be run within ArcGIS 10.0 or later, provided the scripts are properly written, reference the correct version of ArcGIS, and do not contain deprecated ArcGIS or Python functionality or syntax. Some Python geoprocessing functionality (e.g. lists and cursors) has changed slightly with newer versions of ArcGIS and Python as improvements are made. If the Python programmer develops code for one version of ArcGIS, but the end user will likely use a different version of ArcGIS, the scripts should be tested for that particular configuration and may require a different version of Python to be installed on the system implanting the code.

How Python is used with ArcGIS

The author is often asked if Python can be used to change the look and feel of ArcGIS or create a custom toolbar or “button” to perform a special function within ArcGIS. With ArcGIS 10.1 and the introduction of the Python plugin, the answer is “yes.” A Python plugin can be created for a select set of tools, buttons, menus, combo boxes, and application extensions. Other programming languages, such as C# or VB, running on the .NET framework, handle more comprehensive look and feel issues such as forms, grouped tools, etc. However, a more common method to provide a user interface with ArcGIS is through creating a custom “tool” (or script tool) stored within the a custom ArcToolbox and uses custom Python scripts. Often the script tools have parameters that are filled in like other ArcToolbox tools. *Workbook III*, Chapter 10 focuses on coupling Python scripts with a custom ArcToolbox tool.

Python is intended to help automate geoprocessing tasks that are often run in batch mode or through a scheduled process. Long ago, in computer time, Arc Macro Language (AML) or Visual Basic for Applications served this need. Currently, Visual .NET or cross-platform C++ is used to create custom ArcGIS applications or toolsets that require a user to interact with the ArcGIS interface (via ArcObjects® software development kits - SDKs) and the mapping environment. Other resources are available to address these topics and are beyond the scope of this book. See the following websites for more information **resources.arcgis.com** or **support.esri.com**.

Python Development Environments

Many scripting environments exist for developing Python code. One can develop code simply by using a word processor or blank text file. Notepad, Notepad++, WordPad, Word, or other word processor application can be used.

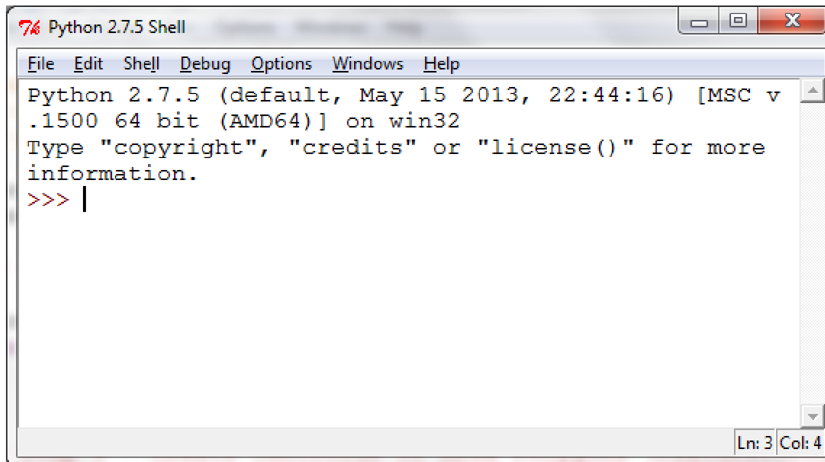
*Integrated Development Environment (IDLE)**

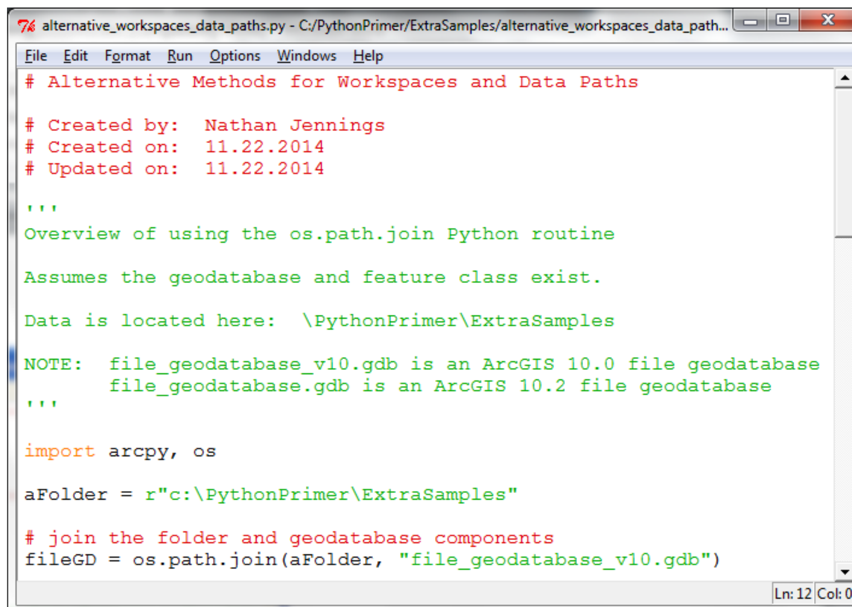
When a developer installs the Python application from the ArcGIS install media or from the *python.org* site, a Python editor space, *Integrated Development Environment* (IDE) - (IDLE), is available that allows for color coding of key words and some simple tools to assist the developer create and edit code. Often, when an existing script is opened with Python IDLE, two windows appear:

1. *IDLE* - the script editor to develop code

2. *Python Shell* - which reports back print statements and error messages when the script is run from within IDLE. The Python Shell can also be used to write simple snippets of code for testing, but it is not intended to write fully functional Python scripts.

These elements pop up by default when pressing the Python development environment icon. This development space also offers some basic error checking such as proper indentation and end of line statements. A user can change some of the look and feel of the application; however, this is limited. The IDLE script editing environment is easy to use and does not require additional software or configuration of the editing environment to write code. The following screen shot shows both the Python Shell and IDLE that contains a sample of a script.





It is recommended that the Python IDLE interface be used for working through the demonstrations and exercises in this book. There are many much more robust script editing environments available on the Internet, but many are not free and almost all require some finessing of the settings to work well with ArcGIS. A short review of different IDEs can be found here.

http://wiki.python.org/moin/IntegratedDevelopmentEnvironments

In his professional career, the author has used PythonWin and Eclipse, but almost exclusively uses the standard IDLE interface for the purposes of this text.

*From *Learning Python*, 4th edition, pg. 58 a footnote indicates IDLE is named after Eric Idle, from Monty Python.

Relationship to ModelBuilder

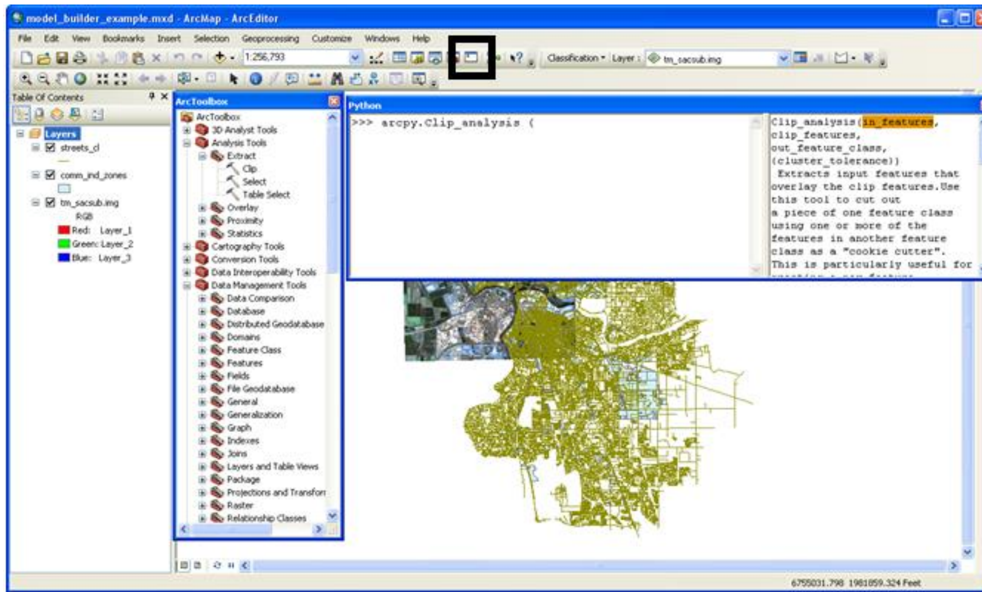
Python closely relates to Esri's ModelBuilder at least in the generation of simple straightforward scripts that do not use loops or conditional statements. A developer can use the ModelBuilder to develop portions of workflows and generate some process logic based on the ArcGIS Tools. A developer can actually spend a lot of time working in ModelBuilder until a proper set of geoprocesses, inputs, outputs, and intermediates are developed. Caveat: ModelBuilder is not an environment for developing loops and conditionals that the user would wish to export to Python.

Once a developer is satisfied with the workflow, a Python script can be exported from ModelBuilder where the developer can continue to work on and refine the scripting process. Chapter 2 describes how ModelBuilder can be used to develop Python scripts and some caveats when using ModelBuilder for code development.

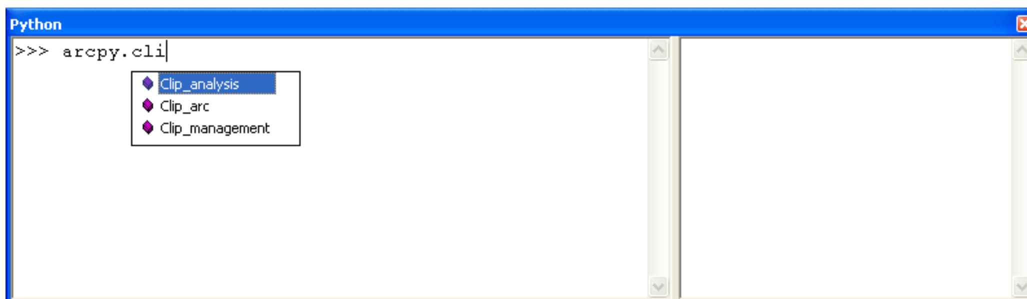
Python Shell in ArcGIS

In addition to opening the Python Shell on its own, it is also available from within ArcGIS. The user can click the Python window button to load the Python shell where Python syntax can be written and processed, including ArcGIS geoprocessing functions. Writing ArcGIS processes within this environment also provides some auto-completion of code and provides some help content for the specific geoprocessing function. Readers may find this useful in checking Python syntax for developing scripts; however, a Python editor will primarily be used for code development, especially standalone scripts.

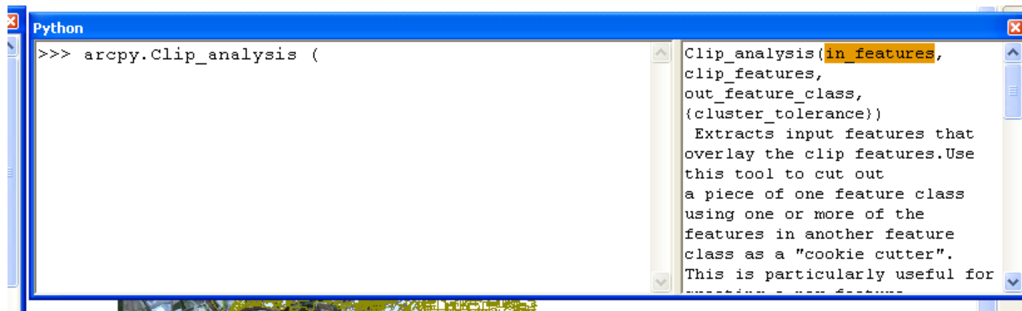
The figure below shows ArcMap with the Python Shell open within ArcMap. It is opened by clicking on the Python button (outlined by the square). Python scripts can be written and processed within this window. In addition, when ArcGIS functions are written, some code completion is available and the specific ArcGIS tool help appears on the right. Function parameters are highlighted as they are encountered when writing the Python script.



The following figure shows the Python Shell opened within ArcMap while writing the Clip routine. As the developer types in the Python syntax, the scripting window provides a list of possible geoprocessing routines. As more characters are typed in, the list becomes shorter. The developer can click on the specific geoprocessing function and the Python syntax will be completed up to this point (e.g. *arcpy.Clip_analysis*).



The figure below shows the Clip routine with the first parameter highlighted on the right side indicating that the developer needs to type in a value for this parameter. The parameter may be a specific value (i.e. a specific name of a feature class). As additional parameters are filled in, the highlighted text on the right changes. When the required parameters are filled in by the developer, the ArcGIS geoprocess can be completed with an ending parenthesis. At this point, with a tap of the Return key, Python will process the routine. Other routines and Python syntax can be written and processed in the same manner. Since this code is written inside ArcMap and not in the Python IDLE environment, it cannot be executed outside of ArcMap.



Although *A Python Primer for ArcGIS* ultimately focuses on writing code outside of the ArcGIS environment, the newcomer to Python programming and ArcGIS may find it beneficial to develop code snippets within ArcMap or ArcCatalog so that the less experienced code developer becomes familiar with some of the geoprocessing syntax and start to develop troubleshooting skills. These skills are required to build functional code for performing geoprocessing tasks. Consult ArcGIS Help for more information for writing Python syntax within ArcGIS using the Python Shell.

Use of Scripts with Geoprocessing Tools

The ArcGIS geoprocessing tools found in the ArcGIS toolbox are fundamental to the development of Python scripts for ArcGIS. Essentially, developers are using the ArcGIS geoprocessing tools to create custom built automated processes that assist analysts and data managers in automating routine tasks that need to be implemented frequently or with numerous iterations. These tasks, if performed manually, could take significant hours or days to complete. For example, an analyst could take a parcel data set from a local tax assessor group and extract a subset of data. He might then want to join a number of tax and owner related tables together. Next, he might wish to run a series of queries and computations based on land use and the number of units found on each parcel. The final product might be a custom parcel data set that is used by a local code enforcement department. Such important data sets may need updating each week. Performing this set of tasks could take a number of hours to do on a weekly basis. A script can be developed that performs all of the above tasks to create, maintain, and update this custom dataset and it can be performed during off-peak hours. Doing so frees up the GIS worker to focus on more analytical tasks that require direct involvement with data, analysis, and cartography.

Getting Help

A typical question of new Python developers for ArcGIS is where to get help and assistance. The Esri ArcGIS Help and support site are primary sources for gaining information and insight into Python syntax for ArcGIS and how it is used for specific tools. However, since Python is an open source application development software, some questions regarding Python may not be addressed within the ArcGIS Help or Esri support environment. Python, on the other hand, has a much broader user base than just GIS and currently, there are numerous generic clearinghouses for Python on the web.

More advanced Python methods (e.g. dictionaries, arrays, functions, etc.) that can be used with ArcGIS geoprocessing objects will likely be researched using Internet searches for these Python methods and by consulting a Python text, the **python.org** website, or studying other developers' code.
A Python Primer for ArcGIS contains some sample scripts written by the author. Additional scripts or code snippets can be found at the author's website, **http://www.urbandalespatial.com**.

The reader is encouraged to review and use these resources among others on the Web such as these, among others:

ESRI support, user forums, blogs, training, ESRI Python forums, and the GIS Stack Exchange.

In addition, if the reader resides in California, the author teaches an online GIS Programming class at American River College, Sacramento, CA and through the UC Davis Extension in Davis, CA.

The author is available for professional consultation and independent projects and training via UrbandaleSpatial. Visit **www.urbandalespatial.com** or contact the author at **nate.jennings@urbandalespatial.com**.

ArcGIS

Specific to Python programming, the ArcGIS ArcToolbox Help for specific geoprocesses will typically be the first point of investigation to get assistance on developing proper syntax and parameters. ArcGIS has improved the scripting help for many of its geoprocesses. In many cases, Python code developers can copy and paste code directly from the help and then modify the syntax accordingly. Many references are made to ArcGIS Help so the newcomer to both ArcGIS and Python has some starting points to assist themselves to research and develop code solutions.

Secondarily, an application developer can access the online ArcGIS resources. Essentially, the online help provides the code developer a world-wide GIS community. Since many people are using and developing under ArcGIS and Python, developers will often find code snippets or entire programs that provide some of the functionality they are looking for. In many cases, the developer will need to write and modify any code that is obtained off the Internet.

Python

As mentioned above, Python has a wide ranging user base that covers many specific disciplines. The Python Help is completely online and with a little effort a code developer can figure out how to use the Python examples. For those who like to have a hard copy text, a variety of books can be purchased on-line or at major book retailers. See the References section at the end of the book for a list of resources, books, and websites related to Python and ArcGIS.

Python and ArGIS Errors

Scripting errors are inevitable when writing programming code. Learning how to understand and decipher error codes and messages will be important for trouble shooting coding and logic problems. Two groups of errors will typically be encountered when writing Python script for ArcGIS:

1. Python related syntax errors, such as typos, indentation, and Python structure

2. ArcGIS errors, which are those related to the incorrect or missing parameters or incorrect data types used in the ArcGIS geoprocessing tools, methods, and properties.

Python syntax errors will likely be identified with the Check Module routine that can be found within the Python editor as well as any error message handling provided by the code developer (such as the use of print statements or the `traceback` module). ArcGIS errors will likely be identified through the use of print statements, custom error handling (using specific `except:` code blocks for different kinds of errors), or the `traceback` module which can identify specific ArcGIS error codes that can be referred to in the ArcGIS help. See more details by referring to ArcGIS Help specific to error codes, tool errors, and error handling.

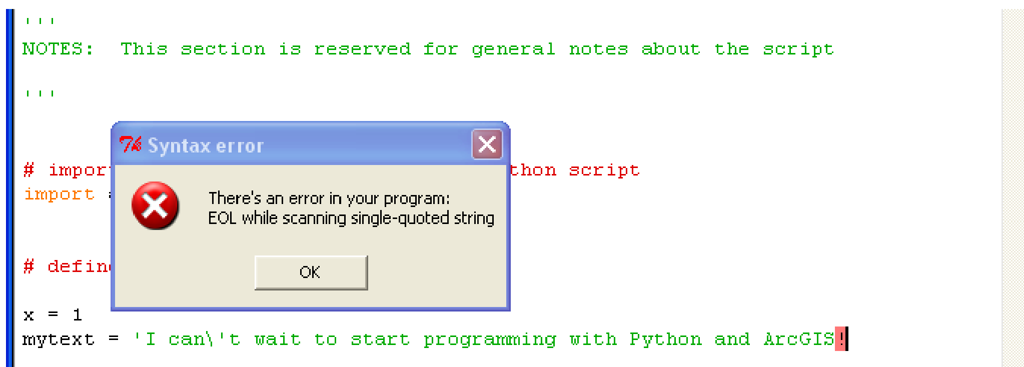
Python Syntax Errors

Once new programmers learn some of the basic Python constructs, most of the Python related errors result from

- mistyping,
- differences in capitalization with variables,
- indentation
- using the proper data type, and
- forgetting to add geoprocessing parameters, colons, and parentheses.

Keep this list in mind and refer to it often!!! Remember, many of your initial problems will be in this list.

These problems will typically show up when the programmer clicks the Check Module from the Run menu in the Python Script Editor. Most of the time the line of code with the problem will be highlighted or the cursor placed at the line with the suspected problem. In some cases the problem may actually exist before the highlighted line, so the programmer will want to review the lines of code preceding the actual error. See the figure below that shows a Python syntax error after running the Check Module.



ArcGIS Error Codes

Error codes have also been a major subject of discussion in the GIS programming community and have not been very well explained. Sorry!!! One major advancement with Esri Python scripting is the use of Esri error codes to help the developer troubleshoot and report errors back to the user (see figure below). Many of the error codes can be typed into ArcGIS Help or the Internet using the error code and "ArcGIS" to find out more details about the specific error (e.g. `ERROR 000800`). See the ArcGIS Help under **Geoprocessing—Geoprocessing tool reference—Tool errors and warnings**. The error and warning sections are divided into folders of error code ranges so the user can easily find specific error documentation. In many cases, the errors are related to

- poorly written syntax by the code developer,
- incorrect tool parameter data type,
- missing tool parameter,
- tool parameters are in the incorrect order,
- the incorrect information has been entered by the end user of a script or tool.

The error below refers to the ArcGIS ERROR 000800 that indicates the tool parameters are not valid. In this case the code developer will want to review the data types of the parameters and the order of the parameters to make sure they are correct before using them in the geoprocessing routine.

```
Name = 'Downtown'
ARCPY ERRORS:
Failed to execute. Parameters are not valid.
ERROR 000800: The value is not a member of NEW_SELECTION | ADD_TO_SELECTION | REMOVE_FROM_SELECTION
| SUBSET_SELECTION | SWITCH_SELECTION | CLEAR_SELECTION.
Failed to execute (SelectLayerByAttribute).


PYTHON ERRORS:
Traceback Info:
  File "C:\GIS Programming\PythonPrimer\Extra_Samples\Escape_Quotes_v1.py", line 35, in <module>
    arcpy.SelectLayerByAttribute_management(nh_FLayer, "OLD_SELECTION", query)

Error Info:
     <class 'arcgisscripting.ExecuteError'>: Failed to execute. Parameters are not valid.
ERROR 000800: The value is not a member of NEW_SELECTION | ADD_TO_SELECTION | REMOVE_FROM_SELECTION
| SUBSET_SELECTION | SWITCH_SELECTION | CLEAR_SELECTION.
Failed to execute (SelectLayerByAttribute).
```

Some Python error handling syntax can be found on several of the ArcGIS tool Help documents (usually found in an `except:` block) and be incorporated into custom code. The error syntax can often be used throughout many scripts and thus can be "recycled." The reader will find the same error handling code throughout many of the examples in this book. The error handling syntax was actually found while reviewing specific ArcToolbox Python example scripts. The exception code script can be found in the **Exception_code.py** file located in the **\PythonPrimer\Chapter01** folder of the supplemental material. The author encourages the use of error handling code when developing scripts.

In addition, code developers can create specific code messages to report issues back to the user when errors are encountered. These can be reported back to the Tool progress window when the script is used as part of a custom ArcGIS tool. The author encourages the use of the `traceback` module and `except:` blocks to assist with handling errors. Refer to the **Exception_code.py** script, too. More details about error handling are discussed throughout the book and accompanying material. *Workbook II*, Chapter 8 specifically discusses developing custom error handlers.

Common Methods for Handling Errors

A number of straight forward and commonly used methods are available to troubleshoot scripting errors. Three types have already been mentioned:

1. Using `print` statements within a Python script
2. Using `except:` blocks (associated with `try:` blocks)
3. Using the `traceback` module used to capture ArcGIS Errors

`print` Statements

Print statements can be added at any time throughout a Python script. Print statements can simply print a message out indicating that the script progressed to a certain line or it can print out a value of a variable, or a counter for a loop among others. Typically, print statements print to the Python Shell while the code developer is working on the code. In addition, print statements can be printed to “log” files to capture progress of automated scripts. See *Workbook II*, Chapter 8 for more details.

`except:` Blocks

`except:` blocks are used in conjunction with `try:` blocks and provide an area of the code to handle errors that occur throughout the code. `except:` blocks can contain simple `print` statements or can contain more involved code that involve the use of variables and different kinds of message similar to those in the above figure.

`traceback` module

Often found within `except` blocks is the use of the `traceback` module elements, such as those shown in the above figure that captures ArcGIS error messages and can be printed to a Python Shell window or to an ArcGIS progress dialog box. Exception code that reports back specific ArcGIS error messages can be very helpful when troubleshooting ArcGIS tool syntax, especially for code developers that are relatively new to the ArcToolbox geoprocessing capability.

Additional commentary regarding error handling can be found in Chapter 3. A discussion of developing custom error messages and handling can be found in *Workbook II*, Chapter 8.

Chapter 2 ModelBuilder and Python

Before jumping into Python, let us introduce ModelBuilder. ModelBuilder is fairly straightforward for combining geoprocesses and is often widely used by seasoned GIS analysts to build multi-step processes. However, developing and augmenting Python scripts generated from ModelBuilder can pose some additional challenges. Python code can be generated from a model and sometimes the code is not in a "tidy" form that is directly usable as a standalone script. This problem can make learning Python for ArcGIS more difficult. From a programming perspective the final task when using ModelBuilder is to generate a Python script which can be edited and further developed in a Python editor. Some caveats are mentioned at the end of the chapter when using ModelBuilder for Python script development. Some of the challenging concepts and geoprocessing requirements will be more fully discussed in the next section. These requirements include feature layers, table views, working with selected record sets, and looping, among others.

Overview of ModelBuilder

ModelBuilder is a graphical interface often used with ArcMap. An analyst can quickly build geoprocesses using pre-existing ArcGIS Toolbox tools, other models, or script tools. ModelBuilder has been available in ArcGIS for a number of years. Many GIS professionals are familiar with and use ModelBuilder to generate multi-step geoprocesses for tasks such as site suitability, site assessment, the movement of materials over a landscape, and changes over time, among others. Before Python was available, ModelBuilder was the primary method for generating multi-step geoprocesses. Since *A Python Primer for ArcGIS* focuses on developing Python script and code development, this chapter describes some basic ModelBuilder operations with the primary goal of generating a Python script that can then be modified within a Python script editor and implemented outside of ArcGIS.

Esri has continued to make improvements and add functionality to ModelBuilder such as looping, conditional statements, and the ability to embed other models and scripts. Many geoprocesses can be developed, built, and run completely within ModelBuilder without the need of a Python script. If the only instance of a geoprocess is within ArcGIS or embedded within an ArcGIS Server environment, then ModelBuilder will meet this need. But suppose the following:

- A GIS analyst wants to be able to use and process data from different environments,
- and reset workspaces throughout a process,
- and auto run geoprocesses
- all of this without any user interaction or dependency on
    - ArcGIS being open or
    - ArcGIS Server using the model . . .

In these situations, developing standalone Python scripts will meet these needs and meet them more easily. The developer can then make these scripts available for other non-GIS users to integrate their own back-end or scripting routines.

For a thorough discussion of ModelBuilder, readers are encouraged to refer to the ArcGIS Help documents under **Geoprocessing—Geoprocessing with ModelBuilder**. *A Python Primer for ArcGIS* will refer to some of the basic concepts with the ultimate goal of exporting a model to a Python script. Later, *Workbook III*, Chapter 10 will describe how to integrate a Python script into ArcToolbox, similar to how a ModelBuilder model can be run within a custom ArcGIS Toolbox.

ModelBuilder Python Script Caveats

Developing geoprocesses in ModelBuilder with the intent of generating Python scripts can pose some frustrating challenges to the code developer if the code developer is not aware of some of the nuances of how ArcGIS references and uses data, workspaces, and data paths. One large caveat occurs in using data from the Table of Contents versus accessing data directly from disk. This circumstance can result in different Python syntax. When code developers try to run a standalone script without specific paths set to data on disk, the script will likely result in error. This can be a large issue for more complex models that are then exported to a Python script. <u>This is the primary reason why the author cautions the reader about using ModelBuilder for developing Python scripts.</u>

The two scripts below show variable references pointing to the same data. The only difference is that the first script points to data that have fully defined paths (i.e. the specific locations to the data on disk). The second script references only the “name” of the data in the Table of Contents. The name of the layer is actually a “feature layer” (an in-memory representation of the data) and does not make a specific reference to the location of the data on disk. This is the “inconvenience of convenience.” ArcGIS allows the user to drag-and-drop items directly from the table of contents or to be selected by drop-down boxes when setting parameters. This is convenient to the non-programmer, but it is inconvenient for the script developer. The

first script will run correctly as a standalone script, while the second will not run until the local variables make a specific reference to data on disk.

```
# Model_to_Python_Script_with_Data_Paths.py
# Created on: 2014-11-22 16:28:08.00000
#   (generated by ArcGIS/ModelBuilder)
# Description:

# Import arcpy module
import arcpy


# Local variables:
point_fc = "C:\\PythonPrimer\\Extra_Samples\\
           file_geodatabase.gdb\\point_fc"

polygon_fc = "C:\\PythonPrimer\\Extra_Samples\\
           file_geodatabase.gdb\\polygon_fc"

output_fc = "C:\\PythonPrimer\\Extra_Samples\\out_fc.shp"

# Process: Clip
arcpy.Clip_analysis(point_fc, polygon_fc, output_fc, "")
```

```
# Model_to_Python_Script_No_Data_Paths.py
# Created on: 2014-11-22 16:23:23.00000
#   (generated by ArcGIS/ModelBuilder)
# Description:

# Import arcpy module
import arcpy


# Local variables:
point_fc = "point_fc"
polygon_fc = "polygon_fc"
output_fc = "C:\\PythonPrimer\\Extra_Samples\\out_fc.shp"

# Process: Clip
arcpy.Clip_analysis(point_fc, polygon_fc, output_fc, "")
```

By default, when data is loaded into the Table of Contents, ArcMap automatically (and transparently) creates the feature layer. It then can be used in a model or other geoprocess when ArcMap or ArcCatalog are opened. However, when creating standalone scripts, other pre-requisite steps may be required for the script to function properly (e.g. defining data paths, workspaces, feature layers, and table views among others). The programmer needs to be aware of the limitations and restrictions of ModelBuilder when using it to develop Python scripts, especially those intended for standalone implementation.

It has been the experience of the author that using ModelBuilder is a good tool to assist with developing the straightforward sections of a geoprocessing workflow and the general set of steps. ModelBuilder can also be useful for a limited examination of Python scripting logic for standalone scripts. However, for making on-going changes and modifications to the Python script, it is neither efficient nor beneficial to modify the ModelBuilder structures so as then to "regenerate" a new Python script. The main focus of this text is to illustrate how to use the major ArcGIS geoprocessing constructs with Python syntax.

A script can be generated from ModelBuilder using these steps:

1. Create a new model
2. Add geoprocessing tools from ArcToolbox
3. Fill in the parameters for each tool (by browsing to the actual path to the correct data)
4. Save the model
5. Test the model
6. Refine the model as required
7. Export model to Python script
8. Make additional edits in a Python script editor

Chapter 2 Demos

The following demonstrations show a generally good practice for creating a ModelBuilder model and then exporting that model into a Python script. There is a preferred way and a non-preferred way. **Demo 2a** outlines the general steps and uses the preferred method for creating a model to generate a Python script. **Demo 2b** demonstrates the (easily confused) non-preferred way (where data from the Table of Contents is used for the input) versus assigning parameters to full data paths to point to data on disk. **Demo2.mxd** in the **Chapter02** folder contains the data referenced in the demos and can be used by the reader to create their own models.

Demo 2a: Using ModelBuilder to Create a Python Script: the Preferred Way

Demo 2a illustrates the generally preferred method for creating a Python script from ModelBuilder. This is particularly important for the new user. ArcGIS tool parameters are not particularly user-friendly. When building a model for export, the process creates the variables and fills the parameters automatically. Two methods can be used to create a new model using ModelBuilder: 1) click the ModelBuilder button or 2) create a new model within a custom ArcToolbox. After the new model canvas is open, the development of the model and the export of the model to Python script remains the same. The code developer can then modify the code using a Python script editor. The reader can create a new model and script in any folder they choose by following along the demo. Note: to save a model, it must be stored in an existing or custom toolbox.

Open the **Demo2.mxd** from the **Chapter02** folder.

STEP 1 - Create a new model using one of the two methods described below.

Method 1: Create a New Model using the ModelBuilder button from ArcMap or ArcCatalog

Method 2: Create a New Model by creating a model in a custom ArcToolbox

Method 1

Start ModelBuilder from the ArcMap or ArcCatalog interface by clicking on the ModelBuilder button.

An new empty model canvas appears.

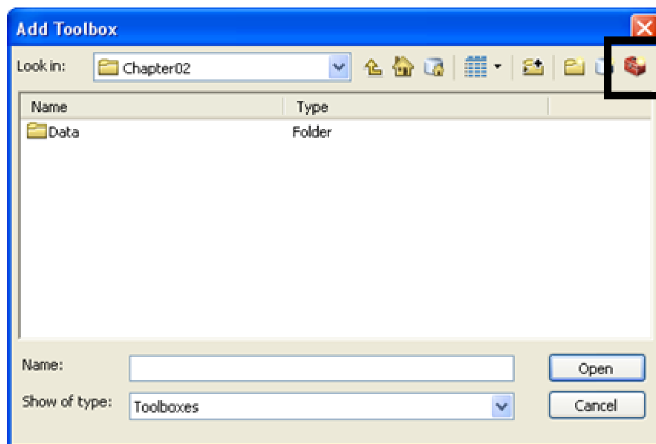
Method 2

A custom ArcToolbox is an easy way to keep custom models or scripts organized. Custom toolboxes can also be saved and provided to other users. To avoid clutter and confusion, tools are kept in toolboxes.

1. Create a Custom ArcToolbox

    a. To create a custom toolbox, right click on the ArcToolbox and choose Add Toolbox.

After the user chooses "**Add Toolbox**," the following dialog box appears.

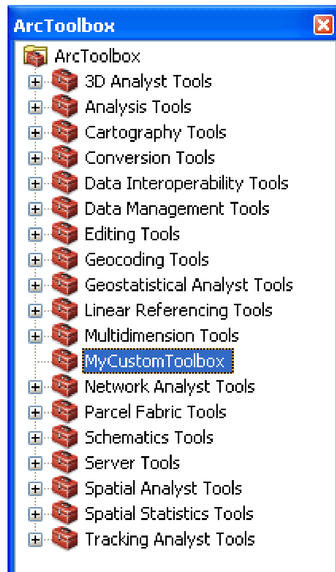


    b. Browse to the **\PythonPrimer\Chapter02** folder (or a folder of your choice). Click on the New ArcToolbox button shown above.

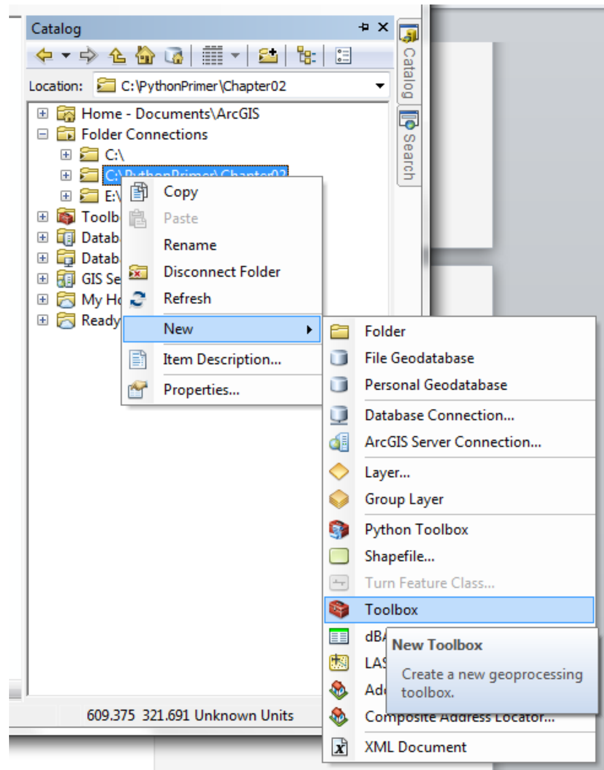
    Note: It is a good idea to store data, scripts, and toolboxes in a common folder structure so that data and scripts are easy to find.

    c. Provide a new name for the new toolbox (e.g. **MyCustomToolbox**).

d. Click Open to add the new toolbox to the ArcToolbox. The new toolbox will appear in the ArcToolbox interface.

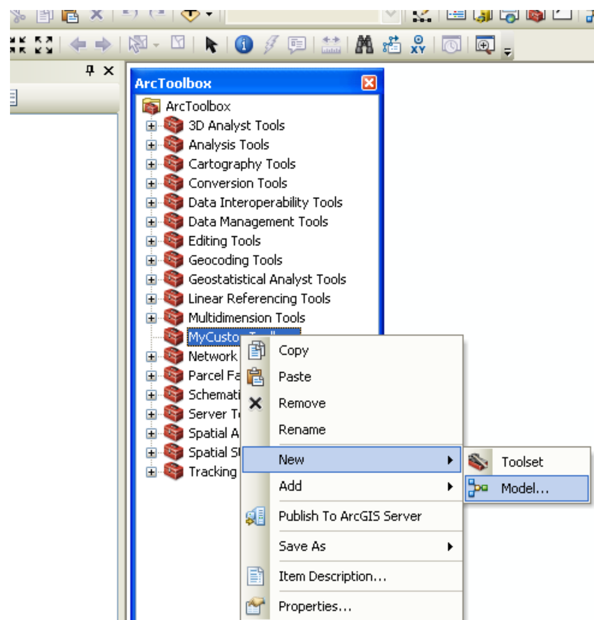


*Alternatively, a new toolbox can be created by going to the ArcCatalog Tab within ArcMap or open ArcCatalog, browse to an existing folder connection, browse to a specific folder or geodatabase, and then right-click—**New—Toolbox**. After creating the new toolbox, assign a useful name to it. See below.*

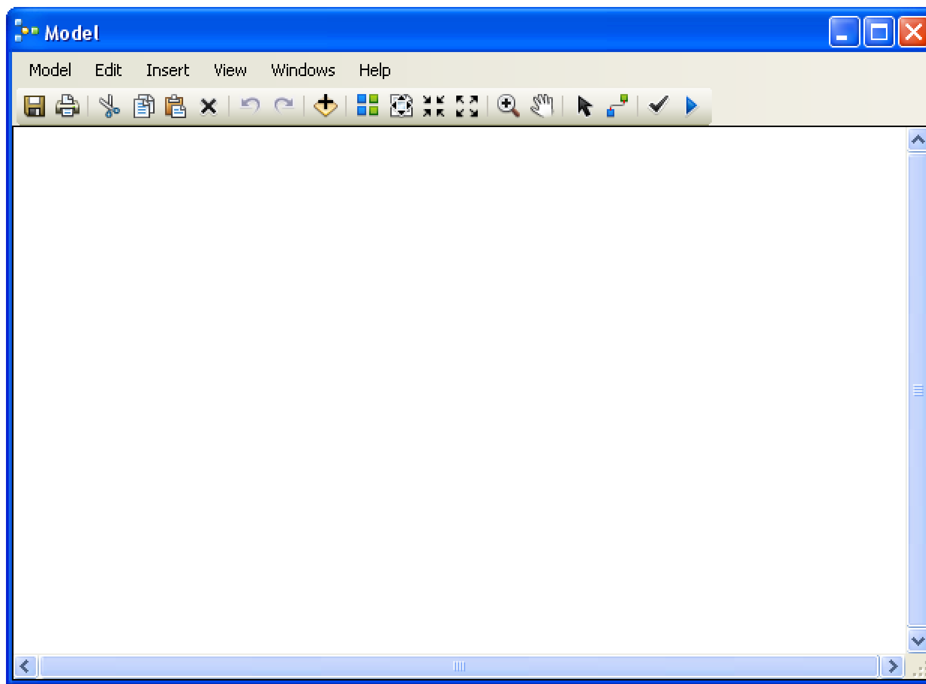


2. Create a New Model to the Custom Toolbox

To add a new model to the toolbox, right click on the new toolbox created above and choose **New—Model**.



The following screen is shown using either method to create a new model.



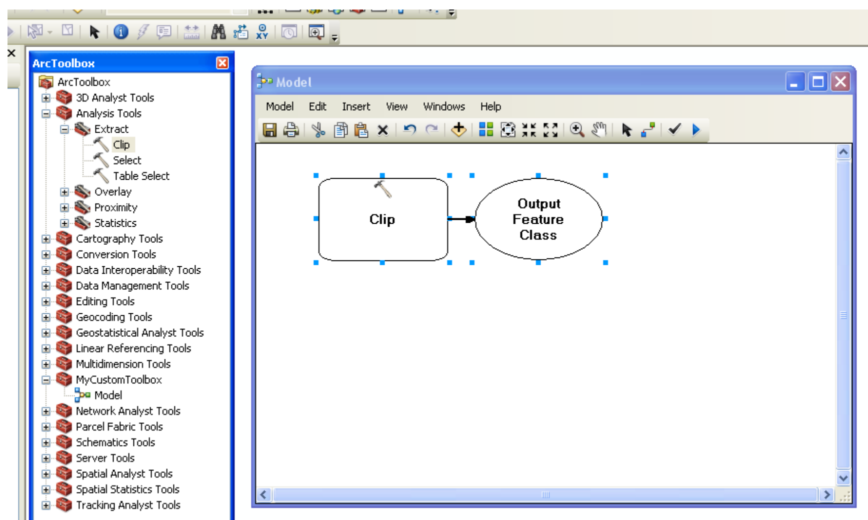
An empty ModelBuilder window appears. Various tools are shown at the top of the screen to add data, zoom in/out, pan the model, make connections between model objects, validate, and run the model. Refer to the ArcGIS Help for a more detailed discussion of each button and option (ArcGIS Help documents under **Geoprocessing—Geoprocessing with ModelBuilder**).

Typically, the person designing the model will select geoprocessing tools from ArcToolbox and drag them into the ModelBuilder window. In the example below, the user clicked the Clip tool and dragged it into the ModelBuilder window. Confusingly, as we shall learn, dragging and dropping *"tools"* is fine. However, the user should avoid dragging and dropping *"layers"* from the Table of Contents (TOC) and should instead browse for their data location on the computer!

STEP 2 - Add a Geoprocessing Tool to ModelBuilder

After creating a new model using one of the methods above, select a geoprocessing tool for the model. The user can drag and drop a tool from the ArcToolbox to the model.

Add the **Clip** routine (from the **Analysis—Extract** toolset) to the model.



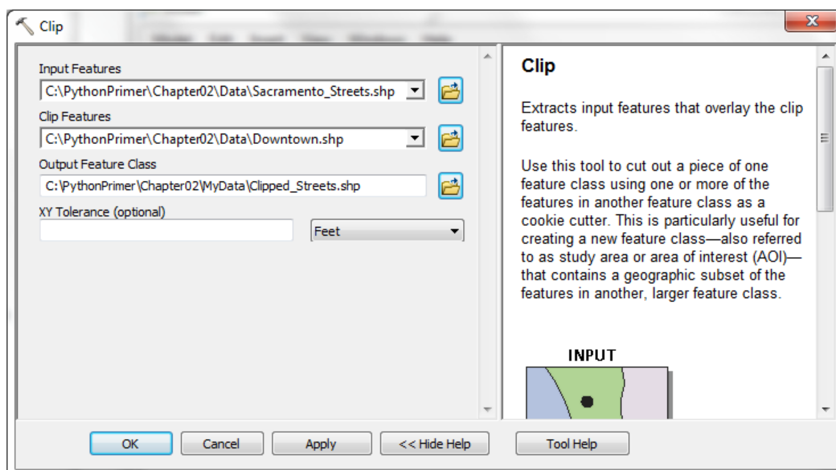
The Clip routine is shown in a "non-ready" empty state indicating that it cannot be run without some additional information. In this case, the model needs some input data, tool parameters set, and an output.

STEP 3 - Set Parameters for a Geoprocessing Tool

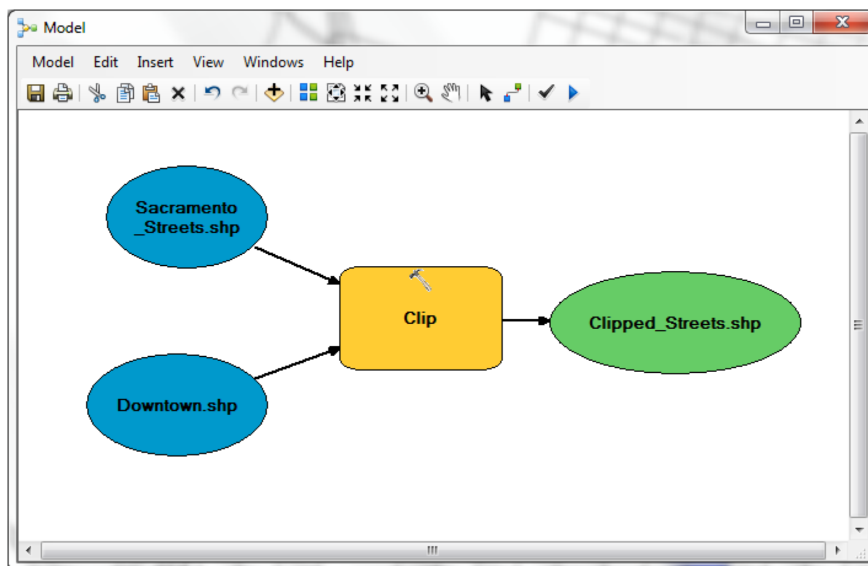
Double click on the Clip routine to bring up the tool parameter dialog box. This is the preferred method for filling the tool parameters. Navigate down to the physical location of the data from the **Chapter02\Data** folder to fill in the parameters.

Input Features: **Sacramento_Streets.shp**
Clip Features: **Downtown.shp**
Output Feature Class: **Clipped_Streets.shp** (named typed in by user)

The output folder will default to the "default" geodatabase. Make sure to change the output folder is **Chapter02\MyData** when adding the "Clipped_Streets.shp" output feature class. Note the Clipped_Streets.shp file will not (yet) exist in the "MyData" folder.



Once the parameters are set and the OK button is clicked, the appearance of the model changes (the model graphics will change color). Use the pan and zoom in/out tools as needed to move or re-center the model. Select each input and output and change the shape size to make the model more readable.



Notice that the model has inputs and an output and the model objects are colored (filled in). This indicates that a model is in a “ready” state and can be run.

STEP 4 - Rename and Save the Model

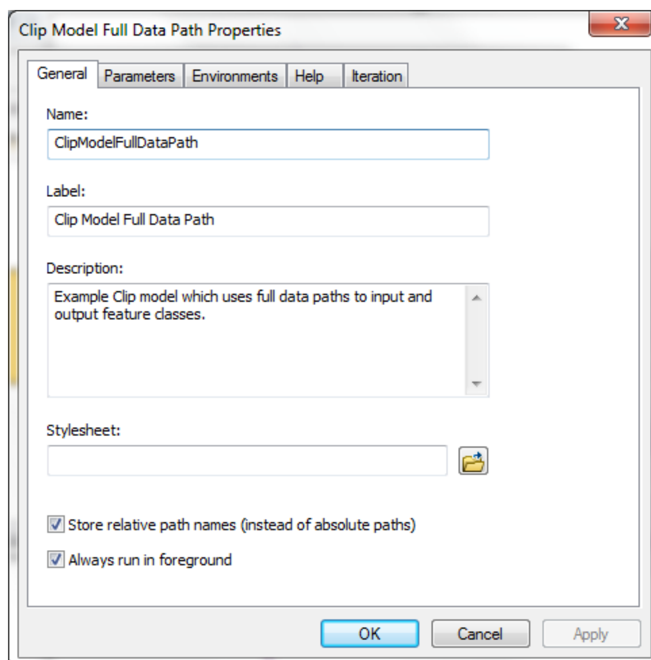
Before saving the model, the default model name needs to be changed.

a. Click on **Model—Model Properties**
b. Rename the model
c. Click OK
d. Click Save from the Model menu

If Method 1 is used to create the model, a custom toolbox will need to be created to save this model. Navigate to a folder (e.g. **\PythonPrimer\Chapter02**) and create a new toolbox using the steps outlined in Method 2. The model for this demo is **Clip Model Full Data Path** and can be found in the **\Chapter02\ClipModels.tbx** toolbox.

The model in the custom ArcToolbox will show the new name.

NOTE: The model name must not include any spaces, underscores, or special characters. The label can have spaces, underscores, and special characters. The label appears in the ArcToolbox as the model name. A description can be added as well which will show up in the tool help. This might seem to be a tedious task, but—much like commenting within the actual code—is an excellent habit worth developing.



STEP 5 - Test the Model

Before exporting to a Python script, the model should be run to determine if it will provide the desired results. Since ModelBuilder is often run within an ArcMap session, the user can check to make sure the proper input feature classes and parameters are set up as well as check the output to determine if the results of the model are correct. This can help reduce issues with data sources and output before working on code refinements.

Run the model.

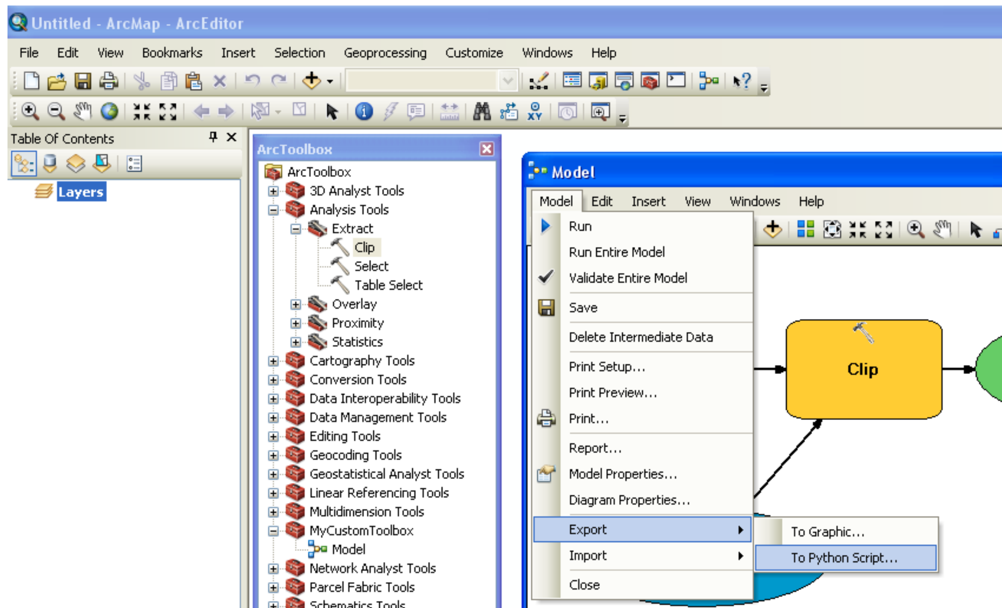
STEP 6 - Refine the Model (as required)

If the model produces an error, make changes, resave it, and run again.

STEP 7 - Export the Model to a Python Script

Once a model has been tested and reviewed, the model can be exported to a Python script.

a. Click on the **Model** menu and then choose **Export—To Python Script.**
b. Choose a location and file name for the Python script. The extension for the script will be (*.py*). NOTE: A **Clip_Data_1.py** script can be found in the **Chapter02** folder for review.



STEP 8 - Modify the Python Script

The script created above can be opened in a Python script editor and modifications can be made. In this demo no further modifications are required, but the script can be reviewed. To open the script, first start a Python editor (likely **All Programs—ArcGIS—Python(*version*)—IDLE**) and then browse for the Python file created above. Alternatively, the developer can browse for the script and right click on the file name and choose **Edit with IDLE** to open with a Python editor.

Either method for opening a Python script will automatically provide the Python Shell window and a separate script window showing the Python code. The script below shows the code generated from the model.

```
# Clip_Data_1.py
# Created on: 2014-11-23 10:24:58.00000
#   (generated by ArcGIS/ModelBuilder)
# Description:
# Example Clip model which uses full data paths to input and output feature
classes.

# Import arcpy module
import arcpy


# Local variables:
Sacramento_Streets_shp =
"C:\\PythonPrimer\\Chapter02\\Data\\Sacramento_Streets.shp"

Downtown_shp = "C:\\PythonPrimer\\Chapter02\\Data\\Downtown.shp"

Clipped_Streets_shp =
"C:\\PythonPrimer\\Chapter02\\MyData\\Clipped_Streets.shp"

# Process: Clip
arcpy.Clip_analysis(Sacramento_Streets_shp, Downtown_shp,
Clipped_Streets_shp, "")
```

ModelBuilder Generated Python Script Commentary

Notice in the above script that all of the information to run the geoprocess is provided. An area of comments is provided that includes the name of the Python script, the date the script was created, and a description if the developer added a description to the model. The `arcpy` module line is added as well as a section for default variable names to the data paths for the geoprocessing tool. Finally the actual geoprocessing line is added which uses the variable names.

The basic structure of a Python script is provided. The code developer can now make modifications to the code as desired. For example, the default variable names may need to be changed, relative paths set up for accessing data, adding looping structures, `try` and `except` blocks, and adding custom print statements and error handling code may be desired.

Demo 2b: Using ModelBuilder to Create a Python Script: the Non-Preferred Way

This demonstration follows the processes described above to generate the Python script for the ArcGIS Clip routine using the non-preferred method using data from the Table of Contents as model inputs. The resulting Python script will not work as a standalone script. This script can be compared to the script created above.

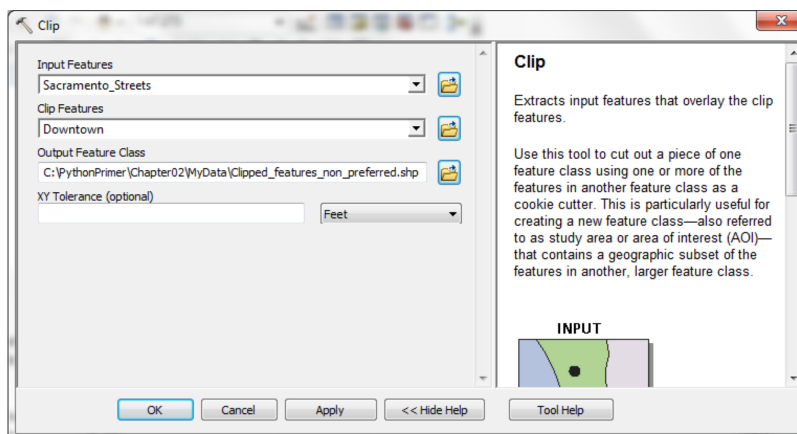
This demo uses the same **Demo2.mxd** and data as **Demo2a**.

The Clip model is created in the same fashion as above. Instead of filling in the parameters using full data paths, the data from the Table of Contents are used.

1. A new model is created in the same custom toolbox created above
2. The Clip routine is added to the new model
3. Double click on the Clip routine
4. Select data from the drop down lists for the input data. The lists include the layers found in the Table of Contents.

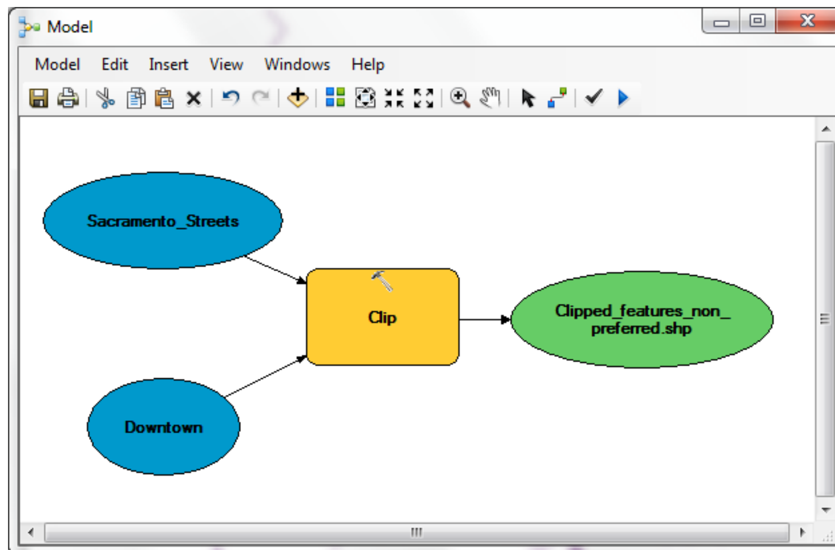
    a. Input Features – **Sacramento_Streets**
    b. Clip Features – **Downtown**
    c. Output Feature Class – Navigate to the **Chapter02\MyData** folder and add a shapefile name (e.g. **Clipped_features_non_preferred.shp**)

The Clip routine will look like the following:



Notice that the input parameters do not contain a full data path to the respective feature classes.

5. Click OK. The model will contain colored graphics which represents the model is in a ready state and can be run.



6. The model name is changed using the same methods as above. (This model has been saved as **Clipped Model Table of Contents** and can be found in the **ClipModels.tbx** toolbox file in the **Chapter02** folder).
7. The model can be run, if desired. In this case, it is not necessary.
8. Click on **Model—Export—To Python Script** to export the model to a Python script. The script created from this step is called **Clip_Data_2.py** in the **Chapter02** folder.
9. The script can be opened in Python IDLE for review.

The following script represents the output from the above process.

```
# Clip_Data_2.py
# Created on: 2014-11-23 11:08:40.00000
#   (generated by ArcGIS/ModelBuilder)
# Description:

# Import arcpy module
import arcpy


# Local variables:
Sacramento_Streets = "Sacramento_Streets"
Downtown = "Downtown"
Clipped_features_non_preferred_shp =
"C:\\PythonPrimer\\Chapter02\\MyData\\Clipped_features_non_preferred.shp"

# Process: Clip
arcpy.Clip_analysis(Sacramento_Streets, Downtown,
Clipped_features_non_preferred_shp, "")
```

Notice the two lines under local variables highlighted above. These variables are not pointing to data on disk because the full path to the data are not shown. The output feature class does show the full path because it was specifically defined in the model. This script will not run as a standalone script until these two lines are revised to point to data on disk.

So, even though it is easy to choose and use “data” from the Table of Contents when building a model, the script that is generated from the model will required augmentation and refinement so that it can run as a standalone script.
The reader needs to appreciate the difference between these two methods when using ModelBuilder to generate scripts. Using the first method (shown in Demo2a) is preferred over using the methods outlined in Demo2b. Using data on disk as parameters in models will help the script generated from the model run more readily as a standalone script. Script modifications will still be required, but they will be easier to identify to make the changes.

In addition, with either method, numerous variables can be created from the process of exporting a model to a Python script that will need to be refined after the script is generated. For simple models and models with few steps, this may not be too much of a problem, but for more complex models, this will be a very time consuming process to make changes in a model and then export it to a Python script. For the more complex processes, it is recommended to refine the model as best as possible, then export the model to a Python script, and then make any further or future refinements in Python script and not in ModelBuilder.

Exercise 2 - Create a New Feature Class and Add Fields

In this exercise, the user will create a new model that creates a new feature class and add some fields to the new feature class using existing ArcGIS Tools. Use the **Create Feature Class** (in the **Data Management—Feature Class** toolset) and **Add Field** (in the **Data Management—Fields** toolset) ArcGIS tools. Research the ArcGIS Help as necessary to compile a model. NOTE: The Add Field tool will need to be used multiple times to add more than one attribute field. Test the model and export to Python script. Test to see if the script runs from Python IDLE. Note any problems, issues, or successes.

Requirements

1. In the custom toolbox created above, create a new model for Exercise 2.

2. Give the model a unique name, e.g. **CreateFeatureAddFields**

3. Add the ArcGIS tools mentioned above to the model with the following conditions.

    a. Create a feature class type of your choice (point, line, or polygon)

    b. Add several attribute fields to the table. Use a mix of data types (text or numbers). For text fields, provide a length. For number fields use short or long integer or provide a precision.

    c. Make sure to connect the Add Field tool(s) to the Create Featureclass Tool or other Add Field tools in the model.

4. Use data path locations from the tool's browser for parameters. Do not use data that is already present in the ArcMap Table of Contents.

Chapter 2 Questions

The questions below refer to the exercise.

1a. What is the full toolbox organization (i.e. Toolbox—Toolset—Tool) for the Create Feature Class tool?

1b. What are the required parameters for this tool?

2a. What is the full toolbox organization for the AddFields tool?

2b. What are the required parameters for this tool?

3. How many times do you need to use the AddField tool?

4. What is required to set the feature class location?

5. What kind of feature class location did you use for the Create Featureclass tool?

6. In the Add Field tool help, can you set the precision and scale using a feature class in a file or personal geodatabase?

7. After creating the model, did the model run? Yes/No. If not, describe some of the methods used to attempt to fix the model? Did your modifications fix the model?

Export the model to a Python script (even if you were not able to fix it).

8. What modules were imported for the script?

9. How many "Local Variables' were created for this script? Do any of them refer to the same value? If so, which one(s)? Do you think any of the variables can be eliminated? Why/Why not?

Chapter 3 Python and ArcGIS Constructs

Overview

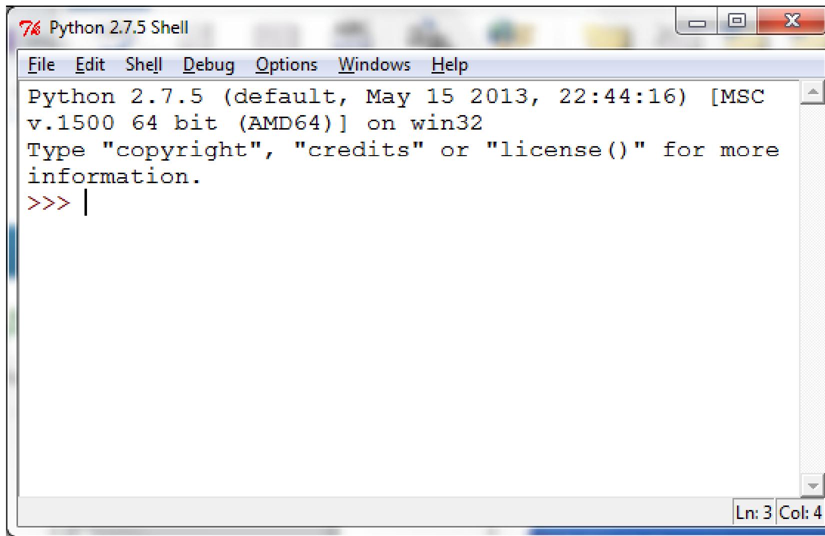
When we extend beyond the functionality of ModelBuilder, a number of basic Python programming fundamentals and ArcGIS constructs need review before writing Python script. These fundamentals will be used extensively when developing code and will likely cause significant frustration if they are not strictly adhered to. This section covers the basic and most commonly used Python structures when programming Python scripts for ArcGIS. Consult a Python book (such as *Learning Python*) or visit the **python.org** site for more details and for a more comprehensive discussion of these structures as well as others.

Using Python IDLE for Code Development

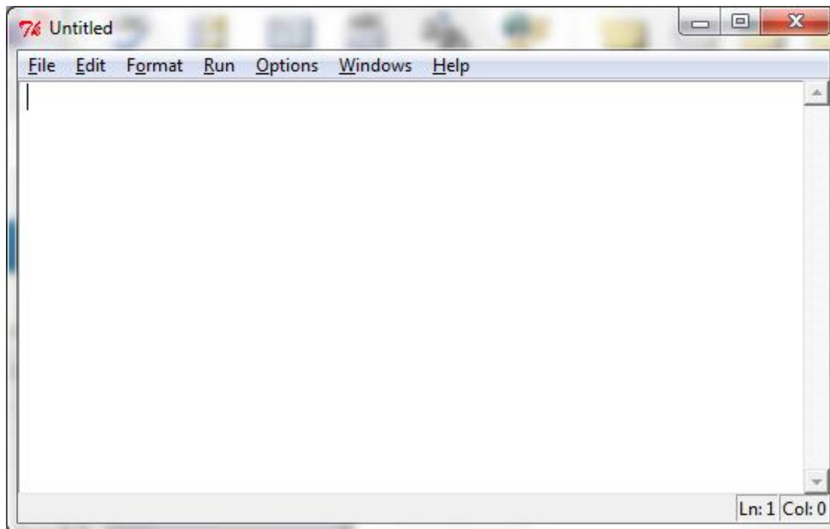
As mentioned above, the Python ***I**nteractive **D**eve**l**opment **E**nvironment* (IDLE) is provided as part of the Python install and will be used throughout this book for demonstrating code. For simplicity's sake, IDLE is recommended when working through the programs within the book as well as with the reader's own code. More robust IDE's are beyond the purview of this text.

Starting Python IDLE

When a code developer clicks on **All Programs—ArcGIS—Python (version)—IDLE** (where ArcGIS installs Python) or possibly **All Programs –Python (version)—IDLE** from Windows, the Python Shell appears.



The code developer can go to **File—Open** and then browse for an existing Python script. For a new script, choose **File—New Window**. In either case a new Python scripting window appears. This is where Python script will be written and edited.

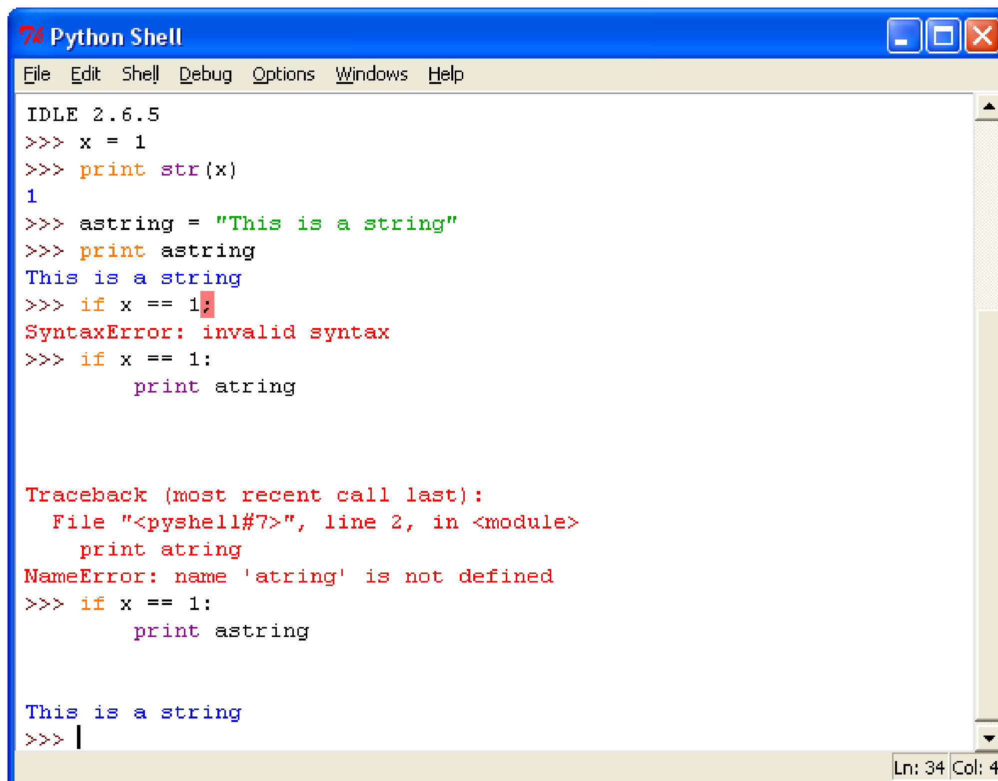


Alternatively, if a Python script file is found in the Windows Explorer, the user can right-click and choose **Edit with IDLE** which will bring up a new Python Shell and load the script into a Python script editor.

Before getting started, read through the rest of this chapter to get an overview of the common elements of Python programming and its organization. At the end of the chapter, the reader will have an opportunity to write a “First Script” that demonstrates some of the concepts described in this chapter.

Using the Python Shell for Code Testing

The code developer may find it useful to type in specific lines (snippets) of Python script without writing, saving, and troubleshooting code. Python script can be written interactively in the Python Shell window and the results shown after completing the lines of code. For example, the figure below shows some variables and an “if” statement typed into the Python Shell showing the results immediately after typing the lines. Notice that error messages will pop up when syntax is written incorrectly.



The Python Shell can also be accessed and used within ArcGIS by clicking on the Python Window button as described in the previous chapters. New code developers may find the ArcGIS Python Window useful since it has the benefit of providing some in-line help, code completion, and additional help documentation for a specific geoprocessing routine. The code can be copied and pasted into the Python IDLE environment to build standalone scripts.

Syntax

Python is a fairly flexible scripting language, provided the developer follows a few coding rules. Any comprehensive Python book as well as the **python.org** site contains hundreds of pages on all of the various constructs Python has to offer, however, most of the commonly used coding structures are needed to write most ArcGIS Python scripts. Because users should have immediate access to a good economical text at all times, the author will occasionally make reference to O'Reilly's *Learning Python*.

Case Sensitivity

Python code developers need to pay attention to the capitalization and naming of variables, values, and other programming constructs (*Learning Python*, p. 225). Typing the same name using a mix of capitalization will be interpreted by Python as distinct names or values. For example, the two names below that show different capitalization will refer to different values (in this case, two distinct feature class shapefiles). Using the same name with different capitalization can lead to difficulty in troubleshooting code and may produce unintended consequences. As with the use of invalid pathnames, this is one of the most common of Python mistakes for the beginner in ArcGIS.

```
featureclass = "aFeatureClass.shp"
FeatureClass = "aDifferentFeatureClass.shp"
```

Naming Conventions

A code developer should adopt a naming convention for a program. This will help code become more legible for the developer and the end user. This will also help the code developer find and debug problems.

- Common examples can use a lower case abbreviation for a data type and then a name starting with an upper case letter (also known as "CamelCase") or lower_case_with_underscores.

For example,

```
strFieldname =  'parcel'
```

may represent a variable name that is set to the word "parcel" (i.e. a string of characters) that represents the name of an attribute field.

The developer can also simply use:

```
fieldname = 'parcel'
```

where the variable name ("fieldname") still represents the word "parcel."

Important! The programmer should maintain a standard practice of naming objects. The name should be meaningful, represent the object, and possibly the type of data (e.g. strings of characters, numbers, feature classes, images, tables, fields, etc.).

Indentation

Indentation is a key requirement when writing Python script. It is the third greatest source of errors in beginning to script in Python. The novice programmer should take it on faith that, whatever the inconveniences of indentation, this method of denoting contiguous blocks of code is far easier than methods in other programming languages. First of all, one is saved a lot of heartache just by not have to count one's `end` lines.
Indentation tells Python that a set of code will be processed as a contiguous block. Indentation is used when implementing looping structures such as `while` and `for` loops, `try: except:` blocks, and conditional statements such as `if` or `else` statements. The figure below shows an example using a `for` loop and the `if` statement constructs. Notice that both the `for` loop and the `if` statement show indentation. The indented lines of code are considered a "block of code" and processed line by line.

```
import arcpy

featclass = "C:/PythonPrimer/data/streets.shp"

# Use Describe method to access feature class properties
desc = arcpy.Describe(featclass)

# Get a list of fields for the feature class
fields = desc.fields

# Loop through the fields
for field in fields:
    # Indent starts here for the "for" loop

    # Use "if" statement to check for a specific field name
    if field.name == "Parcel":
        # Another indent starts here for the "if" statement

        #...other code
```

For basic code development, the Python IDLE script editing tool does a good job of properly indenting code. The developer can also use the Check Module within Python IDLE before running any code that may identify improperly indented lines. Consistent indentation is important when developing code because Python interprets the indentation as a contiguous block of code.

A "tab" or a specific number of spaces (e.g. 4 spaces) indicates an "indent." The default indentation is a "single tab" or four spaces and can be modified in the **Python—Options—Configure IDLE** properties for the Python IDLE script editor. For example, if indentation consists of four (4) spaces or a single tab, then this format should be used throughout the script. Some Python editors provide a setting to modify any indentation defaults. See the specific Python editor help regarding indentation settings.

Comments

Comments are used to provide in-line documentation and commentary about the code. The author uses two different methods for making comments. The figure below shows both methods being used in a Python script. The more common method uses a hash symbol ("#") at the beginning of each line of commentary. This tells the Python compiler to skip over and not read what's written until the next lines without '#' marks.

The other method is a convenience when longer paragraphs of commentary are needed at the header of the code. This is often the location within the code to document code functionality and to record editorial comments when the code changes. This method, as shown below, begins with three single quotes and ends with three quotes. The reader should note that the "three single quotes" are interpreted by Python as a "string". If this method is used within "indented" sections of code, the proper indentation must be strictly maintained; otherwise Python will result in possible code errors. Shorter comments beginning with "#" should be used most often throughout the code. A good practice is to create a separate technical document for both code developers and end-users of the script that explain the functionality of the code as well as the functionality of the overall script.

```
#Exercise 1

# Created by: <author's name>
# Created on: <date>
# Updated on: <date>

'''
Notes:  This section is reserved for general notes about the script.
Multiple lines can be written here or the triple quotes can be used to
comment a block of code.  NOTE:  Proper indentation must be maintained when
using triple single quotes.
'''
```

Comments are helpful because they often provide useful notes about the program, can be used to create an outline for the script, and provide some additional information about program variables and geoprocesses.

Creating and Using Variables

Python often requires defined terms, called *variables* that the programmer uses to write concise and flexible code. A variable is essentially a name that is assigned a specific value that can be used in different parts of the code. For example, a variable can represent different feature classes that are used as an input to a "clip" geoprocessing routine. The code below illustrates the use of variables to define feature class names and the buffer distance for the ArcGIS Clip routine.

```
in_fc = 'c:\\temp\\aFeatureClass.shp'
clip_fc = 'c:\\temp\\aPolygon_FC.shp'
out_fc = 'c:\\temp\\anOutFC.shp'
buf_dist = '200 FEET'

arcpy.Clip_analysis(in_fc, clip_fc, out_fc, buf_dist)
```

The programmer can use the defined variables `in_fc`, `clip_fc`, `out_fc`, and `buf_dist` in any place in a script rather than needing to write the full paths and the specific buffer distance each time they are required in the Python script. Variables can be assigned to almost any value or reference. Variables can be assigned to numbers, character strings, dates, lists, etc. Variables can also be assigned to data paths, work spaces, tables, feature classes, databases, feature data sets, feature layers, file names, images, etc. In many cases, the variables are assigned to numbers or character strings (e.g. a buffer distance, a specific value such as an index number or counter, a table or feature class name, a work space, a query, or a file name). The code below shows some of these examples.

```
import arcpy
from arcpy import env # environment module

# Some examples of common variables used with ArcGIS and Python
# NOTE:  String values (such as names of workspaces, feature
# class names or any alphanumeric value) can be represented in
# “single”, “double”, or “triple double” quotes.

# an ArcGIS Workspace

env.workspace = "c:\\temp\\"

# a directory (folder); not a workspace

datapath = "c:\\temp\\"


# an existing shapefile stored in the c:\temp folder

inshapefile = datapath + "parcels.shp"

# an existing file geodatabase (not a workspace)

fGDB = "c:\\temp\\fGDB.gdb"

# an existing fGDB feature class within an existing
# file geodatabase

infeatureclass = fGDB + "\\" + "streets"

# String Examples

aName = 'ESRI' # a text string assigned a value

# a text string representing a shapefile name

aShapefileName = 'parcels.shp'

# a text string to define a query ("where clause") statement

query = """"APN" = '01234567890000'"""
```

```
# Number examples

# a variable x assigned to the value 0 (e.g. to initialize a loop # counter)

x = 0

aFloatNumber = 4.5 # a decimal number

# a variable y that equals the sum of two numbers
# (in this case 0 and 4.5)

y = x + aFloatNumber

bufDist = 100 # this value can be used in the buffer routine

# NOTE:  the number must be "converted" to a string
# to work properly with the Buffer parameter syntax

arcpy.Buffer_analysis (<input_fc>, <output_fc>, \
                       str(buffDist) + " Feet")
```

String and Number Variables

Some of the most common uses of variables are to define strings (i.e. a series of alpha numeric characters) for names, data paths, workspaces, queries, featureclass names, feature layer names, table view names, join table names, specific field names, etc. Strings are enclosed in single or double quotes, or even “triple double” quotes. Numbers may also need to be assigned to variables so that they can be used and changed as needed. Some common uses of number variables include (number calculations, counters, values in ArcGIS geoprocessing tool parameters such as distance, area, unit measures, etc.). Numbers do not have quotes around the value, as do strings.*

*NOTE: Some numbers may be “cast” as strings so that they can be used in query strings, print statements, and ArcGIS geoprocessing parameters that require strings that have both numbers and text characters. For example, to cast a number as a string, the following syntax can be used `str(<a number>)`. Consult other Python texts and the Python website for more details on casting values.

The reader will find examples throughout the demo scripts and exercises that use the `str()` routine to cast numbers as strings. Both strings and number make up the majority of variable definitions used in ArcGIS parameters.

Strings, Data Path, and Workspace Conventions

Data paths and workspaces (i.e. locations to data on disk) are some of the first lines of code written for a geoprocess and variables that reference these locations will be achieved by using strings. Data path variables are simply strings that represent a specific folder, geodatabase, feature dataset, feature class, or file location on disk. Data paths do not carry any special properties like ArcGIS workspaces do. ArcGIS workspaces are written in a similar way, but instead of a variable, the `arcpy.env.workspace` is assigned to the string. Workspaces can reference folders, geodatabases, or feature (or raster or LiDAR) datasets. Depending on the need (in the script and/or a specific geoprocessing routine's parameter requirements), the strings that make up the data path or workspace can be created in different ways.

As shown in the table below, data paths and workspaces are represented as strings using different forms of the folder (directory) "delimiter" (i.e. the backslash, double backslash, or forward slash), depending on the operating system of the computer. Generally speaking, Windows uses the backslash whereas Linux or Unix uses the forward slash. Because data paths and workspaces use strings to represent a location on disk, different methods can be used. Some will work and some will not depending on how the string and "delimiters" are written.

A variety of data path and workspace syntax will be discovered when researching and reviewing code from other sources. This can often be confusing to the newcomer to Python programming and can be difficult to determine which type of string syntax to use. The following table show different ways to represent a data path or workspace.

String	`data_path`	Workspace	Error/Issue
“c:\\folder\\” “c:\\folder\\fileGDB.gdb\\”	x	x	
“c:\\folder” “c:\\folder\\fileGDB.gdb”	x	x	
“c:/folder/” “c:/folder/fileGDB.gdb/”	x	x	
“c:/folder” “c:/folder/fileGDB.gdb”	x	x	
r”c:\folder\\” r”c:\folder\fileGDB.gdb\\”	x	x	
r”c:\folder\” r”c\folder\fileGDB.gdb\”			End of Line (EOL) error during Check Module
r”c:\folder” r”c:\folder\fileGDB.gdb”	x	x	
“c:\folder” “c:\temp” “c:\folder\fileGDB.gdb” “c:\temp\fileGDB.gdb”			Result may reference special “escape” characters (e.g. “\t”, “\f”, “\n”, etc) in the string

`data_path` represents a variable name; workspace is (`arcpy.env.worksapce`)

As shown most of the string formats are valid as either a data path or workspace. Those that are not valid will produce an end of line (EOL) error or the string will be translated by Python as a special “escape” character (for example, “\t” is interpreted by Python as a “tab,” so “c:\temp” becomes “*c:<tab>emp*”. The “r” at the beginning of some of the examples indicate that the string in double quotes will be treated “literally” meaning that each character will represent the specific character. In the case of the strings starting with “r” and ending in a single “\”, the potential end of line (EOL) errors may occur when validating the code using the Python Check Module.

For the strings that are valid, the use of the path or workspace can have some differences, depending on the need.

For example, if the code developer needs to combine two variables, one representing the data path or workspace and another to represent a feature class name to form a full path to the feature class, the data path or workspace will need to end with the “\\” or “/” or use a different method to create the proper path format.

```
data_path = "c:\\temp\\"  # (or "c:/temp/" or r"c:\temp\\")

feature_class = data_path + "featureclass.shp"

print feature_class
# (will produce:  c:\temp\feature_class.shp)
```

For a geodatabase and feature class, the following syntax is used. (Remember shapefile feature classes are not stored in geodatabases and are a different feature class format than geodatabase feature classes):

```
data_path = "c:\\folder\\fileGDB.gdb\\"

# (or "c:/folder/fileGDB.gdb/" or
# r"c:\folder\fileGDB.gdb\\")

feature_class = data_path + "featureclass"

print feature_class
# (will produce:  c:\folder\fileGDB.gdb\featureclass)
```

If the trailing "\\" or "/" in data_path is missing, then the resulting feature class path will be:

`c:\tempfeature_class.shp` or
`c:\folder\fileGDB.gdbfeatureclass`

Notice the missing "\" in the path above. If two strings represent different parts of a data path, then the proper trailing "\\" (or "/") is required to formulate the correct path structure.

Also note, in some cases (such as the `CreateTable` routine), only the folder or geodatabase name is required. In this instance a variable would be assigned:

```
data_path = "c:\\temp"     # (or "c:/temp" or r"c:\temp")

data_path = "c:\\folder\\fileGDB.gdb"

# (or "c:/folder/fileGDB.gdb or
# r"c:\folder\fileGDB.gdb")
```

In this case the trailing “\\” or “/” are not used because the geoprocessing routine (`CreateTable`) does not require it.

As has been shown, a variety of methods are available to create strings for data paths or workspaces, and depending on the requirements, trailing “\\” or “/” will be required.

It is recommended to adopt one of the methods above by utilizing the “\\” or “/” and use it consistently throughout the script. In addition, using print statements to confirm that variables produce the desired “string” output and locations to data on disk is highly recommended at this step and can help the code developer troubleshoot problems.

Lists

A Python list is a special structure that can store and index a number of related elements. For example, this list could be a list of names, a list of numbers, a list of fields in a feature class, a list of feature classes in a folder or geodatabase, a list of images (rasters), etc. A set of specific ArcGIS list data methods are also available. See *Workbook II*, Chapter 7 and the ArcGIS Help under **Listing Data** for more information on creating and using lists. More general information is available in Ch. 8 of *Learning Python*.
In general Python lists are created in this manner:

```
# define a list
aList = [1, 2, 3, 4, 5]
NamesList = [‘Fred’, ‘Wilma’, ‘Barney’, ‘Betty’]
```

The values in the [ ] represent specific elements in the list. The values in a list are accessed by obtaining the index (i.e. the location of the value in the list. Python lists indexes begin with the value of zero (0), so the “first” element (`aList[0]`) in the list is the value 1.

To access a specific Python list element, the following can be written:

```
aList[1]   # the value in the list at second position in the list is 2

print NamesList[3]   # will result in the string in position 4 printed to
                     # the screen
```

As mentioned above, a list can contain numbers, names, fields, etc. A standard Python list can be used in this case; however, ArcGIS provides some special list methods that can be used directly. For example, the following “creates” a list of attribute fields from a feature class.

```
fieldlist = arcpy.ListFields(“parcels.shp”)
```

The following looping structure can then be used to operate on each element in the list.

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

A more thorough discussion of lists for ArcGIS and Python is discussed in *Workbook II*, Chapter 7. Code developers may find the standard Python list structure useful in some scripts. The author's website **urbandalespatial.com** has some examples of Python lists being used in ArcGIS Python scripts.

Conditional Statements and Loops

Another common set of Python structures is the use of conditional statements and looping structures. This section introduces the concepts; other chapters will demonstrate the use of these structures. Conditional statements provide a means for deciding whether an action takes place within a script based on one or more conditions. Looping structures allow a block of code to "iterate" multiple times, often with a set of conditions, so that the block is not "unbounded" or can "stop" after a limited number of iterations. If loops are not "bounded," the code can run indefinitely, which is not desired (a frequent error in beginning programming!). Notice also that all conditional and loop statements use indentation as part of the coding syntax.

Three kinds of conditional and looping structures are used in Python:

1. **If Statements** – used to "test" a condition or set of conditions. Optional statements include the use of `elif` and `else` statements.

The general form of an `if` statement is:

```
if <condition is true>:
    # this block of script is processed
    # can be multiple lines

elif <a different set of conditions are true>:
    # this block of script is processed
    # can be multiple lines

else:
    # this block of script is processed
    # (if all other if and elif conditions fail)
    # can be multiple lines
```

2. **While loops** – used to iterate a block of code "while" a conditional statement is true.

The general form of a `while` statement is:

```
while <condition is true>:
    # process a block of code
    # can be multiple lines
```

3. **For loops** – used to iterate a block of code "for each element" in a sequence or group of elements.

```
for <item> in <sequence>:
    # process a block of code
    # can be multiple lines
```

import Modules

Python requires specific modules to be imported (i.e. provided to the Python processes at the time of running a script) so that different operations can take place (such as running ArcGIS functions, string processing, math, or operating system functions). Think of modules themselves as toolboxes that hold the tools your scripts will need to successfully run a script. This may seem quite strange to the beginning programmer.

The first "real" lines of code "import" the modules into the Python processing environment where their "tools" (or functionality) are needed. The developer must import those modules using the "import" command. ArcGIS and the proper version of Python must be installed on the system where the ArcGIS Python script is executed. Data sources (feature classes, tables, geodatabases (except SDE) can exist on computers, servers, and networks, that do not have ArcGIS installed on them.)

```
import arcpy, os, sys, traceback
```

The above line imports various Python modules that will be used in a Python script. These are four very frequently used modules—both in the workbooks and in Python/ArcGIS programming in general. It is worth the reader's time to study up on these modules in order to better grasp what each has to offer.

arcpy – provides access to all of the ArcGIS geoprocessing functions including the mapping (`mapping`), data access (`da`), spatial analyst (`sa`), and geostatistical (`ga`), and modules. The `arcpy` module is required to perform any ArcGIS geoprocessing.

os – provides access to operating system functionality such as file and directory path operations. The `os` module is optional in many cases, but required for some file, folder, and data source management operations that use standard Python syntax. Consult a Python text or website for more details. A good basic reference of functionality is available at **http://docs.python.org/2/library/os.path.html**.

sys – is used to access by Python system functions and are often found in defining variables for user input such as:

```
myshapefile = sys.argv[1]  # variable accepts user input for
                           # the first real argument or
                           # parameter from a custom
                           # ArcGIS tool that uses
                           # a Python script
```

The `sys.argv[]` structure will be discussed in more detail in *Workbook III*, Chapter 10 which reviews how create and implement a custom ArcTool that uses a Python script. Refer to **http://effbot.org/librarybook/sys.htm** for a discussion on the `sys` module.

traceback – is used for error handling; this module is not required, but it is often useful for handling errors and the same blocks of code can be reused in many Python scripts. For most Python scripts developed for ArcGIS only the `arcpy` module is required and the `sys`, `os`, and `traceback` modules will often be used. Consult a Python text or the Python website for a full account of Python modules. A good beginning is **http://stackoverflow.com/questions/12791698/python-traceback-module-for-beginners**.

try: and except: Blocks

Python, as with any other programming language, does tell you when you've made a mistake—often cryptically. `try:` and `except:` blocks ease the pain of error handling. They are the essence of good programming practice, even though they are not covered thoroughly until the very last section of *Learning Python* (pp575-ff). Scripts can be written without the use of `try:` and `except:`; however, don't try it! Without these, troubleshooting code and process errors can be more difficult to remedy. `try:` and `except:` must be used together. Proper indentation is required so that Python knows to process the lines of code as a block. For this book, the most common implementations of `try:` and `except:` will be used. Consult a Python text or the Python website for a more in depth discussion of `try:` and `except:`. See **http://www.tutorialspoint.com/python/python_exceptions.htm**.

Special Considerations for Query Strings in Python

When defining query strings some special considerations are needed. These can include backslashes, forward slashes, single and double quotes—and even—“triple” double quotes, AND “escape” characters! A brief discussion of how each of these is used is provided below. Additional commentary can be found throughout the rest of the book, especially in *Workbook II*, Chapter 5 which discusses queries and selecting data. In addition, the reader can consult a Python text or the Python website where these topics are covered in more detail. The text in this book will use one method consistently, and that may be the best approach. Be able to recognize the alternative methods, as any mixture of them may be encountered, but decide to use one form religiously.

Single and Double Quotes

Single and double quotes are used to enclose character strings and operate the same; however, if single quotes are used in string names, it is recommended that the single quote be “escaped” as described above. In some cases the quotes, themselves, will require “escaping.” This issue is often encountered when developing query strings for the “where clause” parameter for several of the ArcGIS functions such as `SelectLayerByAttribute` and the `SearchCursor`.

“Triple” Double Quotes

Another option to define strings is the use of “triple double quotes.” More recent Python examples in ArcGIS show the use of triple double quotes to define query strings. This method can provide a more consistent method versus the convoluted use of escape characters when using single and double quotes. The syntax can look a little strange, but is also provides a method to make troubleshooting query syntax a little easier.

The following code shows three examples of creating a sample query string using single, double, and triple double quotes, and escape charaters. All three methods produce the same string format result:

```
"NAME" = 'Downtown'


# a variable
Neighborhood = 'Downtown'

# query string with single quotes and escape characters
# "NAME" represents an attribute field

query = '"NAME" = \'' + Neighborhood + '\''

print query

# query string with 'double  quotes' and escape charaxters

query = "\"NAME\" = '" + Neighborhood + "'"

print query


# query string with 'triple  double quotes'


query = """"NAME" = '""" + Neighborhood + """'"""

print query
```

For file-based data (such as shapfiles, file geodatabases, and ArcSDE databases double quotes are required to surround the field name. See *Workbook II*, Chapter 5 or the ArcGIS Help topic **Building a query expression** in ArcGIS that discusses proper format for different data types when creating query strings and other ways to “delimit” field names in different GIS formats.

The general form of the query is “<field_name>” = <value> such that “Name” is the field name and ‘Downtown’ is the name (string value), for example, the name of a neighborhood.

General Structure of a Useful Python Script

A Python script is typically processed from the first line to the last line. A typical structure of a Python script is shown below and has the following general sections.

1. Title and Script Commentary/Notes
2. Import modules
3. Variable assignments (also Python function definitions)
4. Code Body
    a. `try:` block that contains the functional code of the script
    b. `except:` block – for error handling

```
# A typical geoprocessing script layout

# Created by:  <author>
# Created on:  <date>
# Updated on:  <date>

'''
NOTES:  This section is reserved for general notes about the script

'''

# import the required modules for the python script
import arcpy, os, sys, traceback


# define common variables here

'''
Can include:

1. Workspaces
2. Directory or folder paths
3. Input/Ouput files, feature classes, tables, etc.
4. log file variables (e.g. print processing statements to a file)

'''

try:

    # put geoprocessing code here

except:

    # put error handling code here
```

As shown above, the script has a number of sections. These help code developers organize their code.

NOTE: This script above can be used as a guide in the exercise below.

Title, Author, Date, and Script Comments

Typically, the first section of a Python script shows some comments that provide a title of the script as well as the author, date, dates of code changes, etc. Comments can use a "#" or a series of three single quotes (''') that blocks out a "region" of code. This structure is commonly used to comment special lines, such as special notes about the script, or blocks of code that an author wants to keep, but not process. Single line comments can be used to provide some in-line documentation for the script. Remember that triple single quotes used to comment out sections of code must also use proper indentation when used within the code body.

import Modules

With just a line or two, the required modules can be imported as shown above. <u>The import modules must precede all other functioning lines of code.</u>

Variable Definitions (and Python Function Definitions)

NOTE: Custom Python function definitions created by the code developer are often placed above any major section of Python script. Custom Python functions (known as "decorators") are not discussed in this book. The reader is encouraged to consult a Python text or the **python.org** website for more information. One useful introduction is at **http://simeonfranklin.com/blog/2012/jul/1/python-decorators-in-12-steps/**

The only requirement for variable definitions is that they be assigned before being used. Unlike some programming languages, variables do <u>not</u> have to be explicitly defined above the body of the script; however, defining some variables at the top of a script does make it more convenient for both the code developer and user to readily identify these variables, if changes are required. Common examples include:

a. Workspace paths
b. Folder/Directory paths
c. Variable name that represents a log file (e.g. a text file to print statements to)
d. Input and Output data file, feature class, feature layer, table, or table view names

Code Body

The majority of the code will be written within the `try:` block. Any conditional statements, loops, and geoprocessing functions are typically written within the `try:` block. In addition, the `except:` block will contain any error messages that the code developer wishes to handle. *Workbook II*, Chapter 8 will discuss error handling in more detail.

Running a Python Script

Once a Python script has been written and saved, the script can be run to actually process the script. Often, after changes have been made to the script, a three step process is used to run the script:

1. Save the script
2. Use the Check Module to check the script for any Python errors
3. Run the script

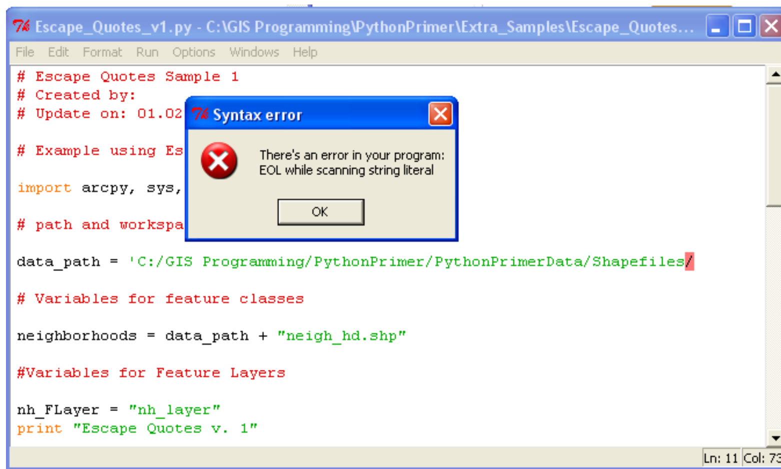
Check Module

Once edited code is saved, a code developer can check the Python script for such common Python syntax errors, such as:

- indentation,
- closed parentheses,
- missed colons,
- misspelled key words, and
- quotation marks, backward and forward slashes, among others.

To do so, the developer clicks **Run—Check Module** from the IDLE scripting window.

If errors appear, IDLE will highlight the line. All Python errors must be remedied before running a Python script; otherwise, the script will present an error message. The figure below shows an error message after running the Check Module. The location in the line with the error is highlighted. This error will need to be fixed before the script can be "Run." In the author's experience, most Python and ArcGIS related errors occur as a result of mistyping, improper Python syntax, and the parameters are not properly set to run ArcGIS functions.



The developer should note that the Python IDLE script editor does not check for ArcGIS function syntax. It also does not perform function completion. Additional errors may occur after the check is successfully achieved. The developer will want to:

- check ArcGIS function syntax,
- check to make sure variables are set to the proper values, and
- check that query syntax is written correctly.

One should consult ArcGIS Help for specific geoprocessing tool parameters, data type, syntax and examples. In addition, the developer may wish to test some of the geoprocessing functions “on the fly”, using the ArcGIS Python Shell (sometimes noted as the Python Window in geoprocessing tool help documents).

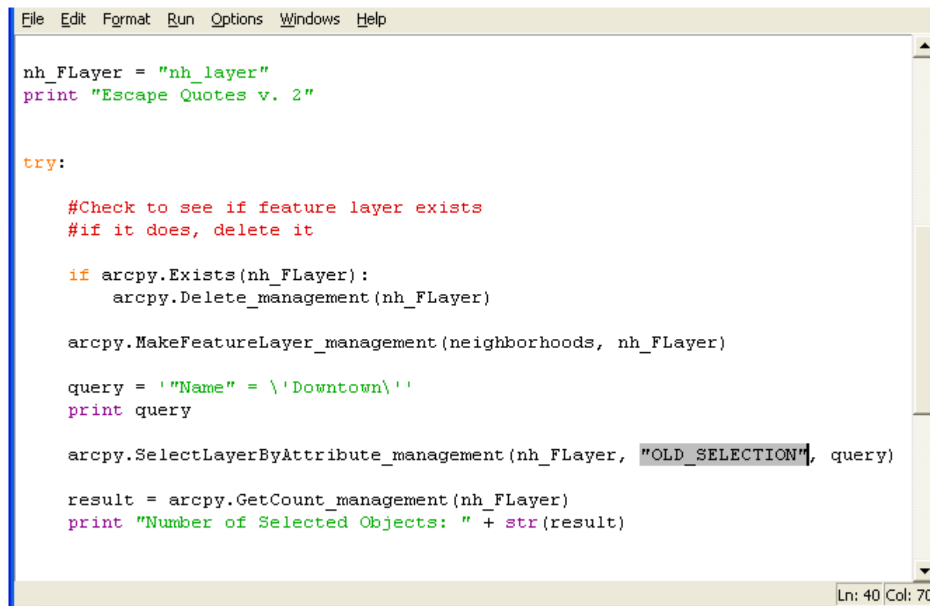
Run Module

Once the Python script has been checked and/or fixed for Python syntax errors, the script can be run to process the lines of code. To do so, the developer clicks **Run—Run Module** from the IDLE scripting window.

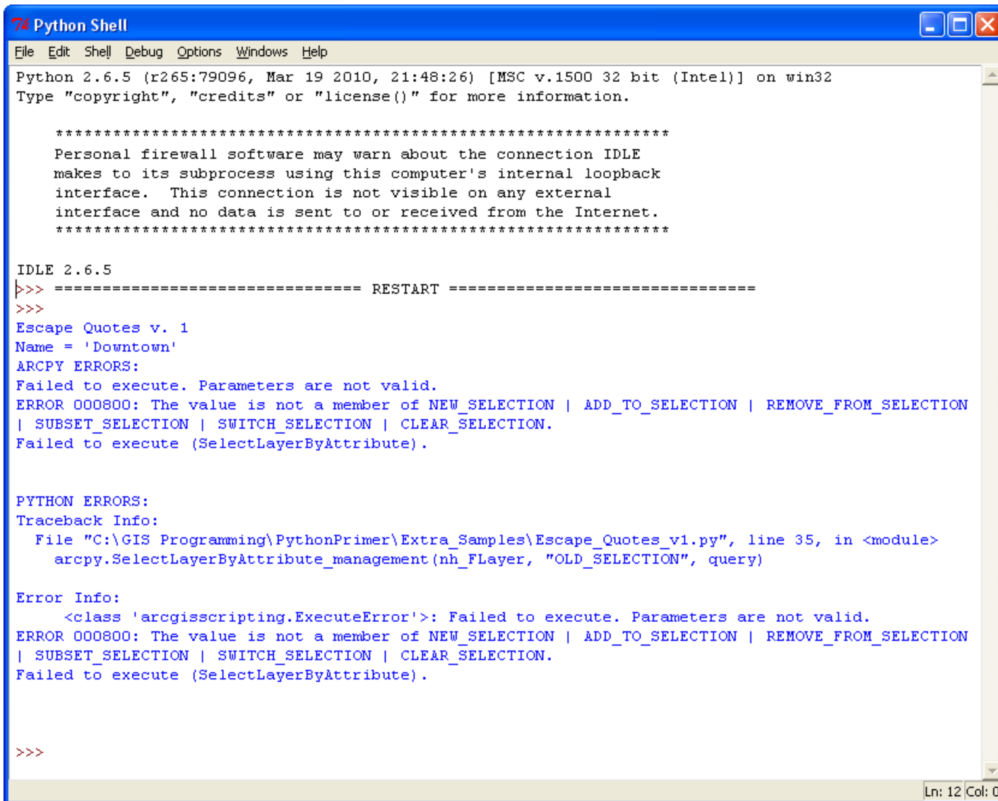
Handling Errors

If the code generates an error before successfully running the script, the developer will want to review the script again for syntax and logic problems. Once these problems are corrected, the script needs to be saved and checked before re-running. Python will indicate the line of code where the program fails. Note: processing errors may occur <u>before</u> the line indicated in the Python error message. The figure below shows a Python script that passes the Check Module operation, but fails when the program runs. Note the parameter `"OLD_SELECTION"` highlighted below. `"OLD_SELECTION"` is not a valid selection type for `SelectLayerByAttribute` and hence an error message is produced.

NOTE: An error message is produced because the script contains code in the `except:` block that traces (checks for) ArcGIS problems. Without this exception block, the script would run without error, but the final print statement would not print out because the program does actually produce an error. It will stop running before the final print line (in the `try:` block).



If a Python script is run as a standalone script (without being run through a custom ArcGIS tool), error messages and print statements are "printed" to the Python Shell window. The figure below shows the error messages reported back from a script to the Python Shell. The specific error indicates that the function parameters are not valid and lists the valid types that the *SelectLayerByAttribute* function accepts. The error message also indicates the specific line of the error. In addition, the specific ArcGIS Error is noted (`ERROR 000800`). A user can look this error up by doing a search on the error code in the ArcGIS Help. More details on ArcGIS errors are discussed in the next chapter.



Summary

Chapter 3 provided an overview of the fundamental Python constructs will be used in almost all ArcGIS geoprocessing (and general Python) scripts. Many of these fundamentals will be used and referred to throughout the rest of the book and in one's own coding. The author encourages that the reader study and review these fundamentals, while working through the script exercises for this chapter.

Exercise 3 - Write a Simple Python Script

Python Concepts

Script layout and organization
`import` modules
Variable definitions
`try: except:` code blocks
`print` statement
Python IDLE functionality

This exercise demonstrates the overall organization of a Python script and some of the methods to create variables. There are no ArcGIS functions used in this script. Create a simple script with the following conditions. Optionally, answer the following questions below.

NOTE: Use the **\PythonPrimer\Chapter03\General_Layout.py** script to begin writing the script below; otherwise, open a blank Python scripting window and add the following syntax. Use the text as needed to help write the script.

NOTE: If the reader does a copy/paste of the electronic version of the text, the quotes may need to be retyped within the Python scripting editor.

Open a copy of the **General_Layout.py** script

1. Start the Python IDLE editor from **Start—All Programs—ArcGIS—Python<version>—IDLE**
2. Click **File—New Window** to open a blank Python scripting window
3. From the Python IDLE window click **File—Open** and browse for the **General_Layout.py** script file. (Alternatively, the reader can browse the Windows Explorer and go to the **Chapter03** folder and right click on the **General_Layout.py** script and choose **Edit with IDLE**.

Start writing the script

4. Add a title, name, creation date, and notes section
5. Add the `import os`, `sys`, and `traceback` modules
6. Create the following variables:

a. String variable assigned to the following text (all on one line of code).

```
"""I can't wait to start programming with Python and ArcGIS!"""
```

b. A number variable, `x = 1`

7. Create a `try:` block and add the following code. Make sure to properly indent (usually one tab). The `if` statement will be indented because it will be placed inside the try: block. The print statements in the "if" block will be indented. See the text above and consult a Python text or resource if necessary. The first print statement will be written on a single line.

```
if x == 1:
    print "This is my " + str(x) + "st Python script \n" + <put your
string variable name here>
    print "Program Successful!"
```

Make sure to replace the `<put your string variable here>` (including the <>) with the name of your variable.

8. Add an `except:` block and add the following code. Make sure to properly indent. The second print statement is on one line. The code shown here has been formatted to fit the page.

```
print "Program failed."
print """z is not defined.  Assign "z" a value and re-run
the script."""
```

9. Add a comment line for each statement in the `try:` and `except:` block.
10. Above the `try:` block add the following statement

```
print "Starting program…"
```

11. Save the script (make sure to save the file as a *.py* file name extension).
12. Check the script for any errors by going to the script window where the code was typed into and click **Run→ Check Module**. If any errors appear, resolve them, save the changes, and recheck the script. Most of the errors at this point should be related to typos, missed quotes, variable names, and indentation. See above for comments on using the Check Module.

For this program, search through the code and make sure the syntax is typed correctly. If needed, the solution can be consulted, but get in the habit of attempting to "troubleshoot" the code on your own and become familiar with the resources listed in the text.

13. After any errors are resolved, click **Run→Run Module** from the script editing window. Review the results in the Python Shell.

Make the following change to the code, save and check the script, and run again.

14. Add the following line within the “if” statement after the first print statement and then re-run the script. Notice what happens in the script.

```
print z
```

Chapter 3 Questions

1. What are the following used for?

    a. Python IDLE
    b. Python Shell

2. What two key syntax elements are important standards to follow, when writing Python code?

3. What are variables and how are they used?

4. What is a good recommendation for creating variable names?

5. Give an example for each of the variable types:

    a. String variable
    b. Number variable

6. What are the conditional statements and looping structures typically used in Python?

7. What are the `try:` and `except:` blocks used for?

8. Briefly describe what the Check Module and Run Module function do in Python IDLE.

The following questions focus on the script created in Exercise 3.

1. Why is a “triple double” set of quotes used in the first statement being assigned to a variable? (HINT: Change the “triple double quotes” in the first statement to a single set of single quotes. Note the changes in the Python syntax).

2. Why are “triple double” quotes used in the print statement in the `except` block? What if the “triple double” quotes are changed to a single set of “single quotes.” Is there any different in the syntax? Does the script still function properly? Note any changes when a single set of “double” quotes is used to encapsulate the `print` statement. Note: that the “z” uses double quotes in the print statement.

3. In the print statement above, what does the "`\n`" represent?

   What does it do when the script is run? Do a search on **python.org** or consult a Python text (such as *Learning Python*).

4. When `print z` is added, describe what occurs in the script?

Chapter 4 Writing a Basic Geoprocessing Python Script

Overview

This chapter will expand on the Python fundamentals for Chapter 3 and introduce more ArcGIS-specific concepts, organization, and syntax. The latter part of the chapter will focus on setting up workspaces, variables, and the Clip and Buffer geoprocessing routines. These will be used to demonstrate the first geoprocessing script using Python and ArcGIS.

Getting Ready to Create an ArcGIS Geoprocessing Python Script

Before starting a Python script the code developer should consider what kind of geoprocessing tasks will be accomplished and how the script will be organized. For example, if a series of geoprocessing tasks have already been identified and performed manually, a list of the tasks and the order in which they are implemented can serve as the basic outline for the Python script. If, on the other hand, a code developer is not sure of the geoprocessing tasks, then the developer should consider consulting the ArcGIS help and performing some of the individual tasks manually in ArcMap to discover the appropriate geoprocesses and the order in which they need to be organized. In addition, the code developer should consider developing the geoprocessing workflows described in the Introduction. Lastly, and always importantly, the author recommends using ModelBuilder, as described in Chapter 2, to assist in the code development process.

Using Pseudo-code to Outline Geoprocessing Tasks

A common method for outlining coding tasks is the use of “pseudo-code.” Pseudo-code is no more than an outline of coding tasks in plain English. Similar to developing an outline for a paper, book, or written document, pseudo-code begins with the broad general tasks and then becomes refined to include more specific tasks, ideas to test, and notes for further consideration. Code development can take weeks or months to create, refine, and test, so having an in-line set of documentation can be extremely helpful.

The following example script does not have much actual code, but it is a good start at an outline for general coding tasks.

```
# Pseudo-code for Geoprocessing

# Author: <author>
# Created on: <date>
# Updated on: <date>

'''

NOTES: This section is reserved for general notes about the script

'''

# import python modules neede to run script
import arcpy, os, sys, datetime, traceback



# add and define variables, data paths, environment variables



try:

    # Create feature layer

    # Select features by attribute

    # Query:  Select all land use types that are single family

    # Compute land use value

    # Research parcel data to determine proper values

    # Add new field for land use value

    # Calculate land use value Structure Value * 0.0125

    # Copy all rows to an output table
    # Use CopyRows function


except:

    # add error message handling here
```

Before beginning code development, some of the elements specific to ArcGIS geoprocessing need further explanation.

arcpy Module Overview

The `arcpy` module is the required module for performing ArcGIS geoprocessing tasks in Python. It sets up the reference to any of the available geoprocessing methods and properties provided by Esri in ArcGIS.

Examples of geoprocessing methods are any of the geoprocessing routines found in the ArcToolbox. For example,

Clip, for clipping features
Buffer, for buffering features
CreateFeatureClass, for creating feature classes
MakeFeatureLayer or *MakeTableView*, for creating feature layers or table views
AddField or *DeleteField*, for adding to, or deleting fields from, a table

Examples of geoprocessing properties usually refer to specific elements of feature classes, tables, images, and spatial reference parameters, among others. Some properties include:

Field names
Data types (points, lines, and polygons)
Spatial extent parameters such as the coordinate system
Number of bands in an image

Alternatively, specific import modules can be referenced and used in Python such as the `os` and `sys` modules. This can simplify code, but the developer needs to be aware of how the modules are referenced, since some confusion can occur and/or syntax can be more difficult to interpret. See the **ArcGIS Web Help** under **Importing ArcPy**.

The script below shows one method of importing all of the `arcpy` modules and all of the mapping routines that can be used for working with ArcMap documents while simplifying the code development. The use of "`from`" and "`import *`" alleviates the need for the code developer to write "`arcpy.mapping`" in front of each mapping routine (i.e. `MapDocument` and `ListDataFrames`). They are great time savers. *Workbook II*, Chapter 9 covers the `arcpy.mapping` module in more detail.

```
# Alternative to import specific modules into Python

import arcpy
from arcpy.mapping import *

# Get MXD and then list the data frames

mxd = MapDocument(r"c:\temp\ProjectMap.mxd")
dfs = ListDataFrames(mxd, "Parcels")[0]
```

Note the use of the "r" delimiter and the single backslash in the above example to process the data path as raw string as opposed to the double backslash that does not require the "r" delimiter. If "r" is not used with single the single backslash, then a double backslash "\\" or a forward slash "/" would be required. See Chapter 3 for more details.

Without the use of `from arcpy.mapping import *` the full reference to the `arcpy.mapping` module is required for all `arcpy.mapping` routines (e.g. `MapDocument` and `ListDataFrames`). This explicit referencing to the `arcpy.mapping` module at almost every step can increase the chances of error.

```
# arcpy.mapping module "NOT" explicitly imported into
# Python

import arcpy

# Get MXD and then list the data frames

mxd = arcpy.mapping.MapDocument(r"c:\temp\ProjectMap.mxd")
dfs = arcpy.mapping.ListDataFrames(mxd, "Parcels")[0]
```

Workspace Definitions and Data Path Variables

Often the next section of code written in a geoprocessing script is one or more references to data paths or workspaces and provides access to data and to properties related to workspaces and other routines such as listing different kinds of data, describing data, etc. As discussed in Chapter 3, the formulation of strings to data paths and workspaces can be challenging. Choosing a standard method for defining data paths, workspaces, and references to data will help design readable code and help troubleshoot problems. Refer to Chapter 3 and other sources for examples on writing proper syntax for data paths and workspaces.

Alternative Method for Workspaces Definitions and Data Paths

In addition to the methods describe in Chapter 3 code developers may find the following "Pythonic" syntax useful when defining data paths and workspaces. The `os` module must be imported to use this syntax.

```
import arcpy, os

aFolder = r"c:\PythonPrimer"

fileGD = os.path.join(aFolder, "file_geodatabase.gdb")

fc = os.path.join(fileGD, "a_featureclass")

# alternatively, this syntax can be used

fc2 = os.path.join(aFolder, "file_geodatabase.gdb",
"a_featureclass")
```

The script yields the following results:

```
fileGD - c:\PythonPrimer\file_geodatabase.gdb
fc - c:\PythonPrimer\file_geodatabase.gdb\a_featureclass
fc2 - c:\PythonPrimer\file_geodatabase.gdb\a_featureclass
```

The `os.path.join` routine allows different "path" components (folders, geodatabases, feature datasets, feature classes, images, etc) to be "joined" to create a single path while producing cleaner code and helping to reduce "typo errors" by not having to make sure the "\\" is used properly for folder, geodatabase, workspace, and data locations.

The geodatabase above could then be assigned to a workspace and then used in other coding processes.

```
arcpy.env.workspace = fileGD
gd_workspace = arcpy.env.workspace

gd_info = arcpy.Describe(gd_workspace)

print "GD Type: " + gd_info.workspaceType
print "GD Version: " + gd_info.release
```

See the **alternative_workspaces_data_paths.py** script in the **\PythonPrimer\Extra_Samples** folder for more details.

Define Variables

While developing a geoprocessing Python script, the programmer must review geoprocessing functions and determine the kinds of required variables. Typically, “hard coded” values used as parameters, such as specific file names, specific query strings, or function parameters make code less flexible and require the code to be changed with different data. Variables help make code more flexible and can be used with other data for the same purposes. Usually variables will be defined for data paths, workspaces, input and output data, query strings, and geoprocessing function parameters among others. Once a variable is defined, it can be used multiple times throughout a script.

Hard Coded Parameters

The Clip function below does not use variables, but rather has specific values “hard coded” for each parameter. The program must use the specific quoted values to process the Clip function. If the data path or data change, then changes to these parameters will also need to be changed. The code has been formatted to fit the page and does not correspond to correct Python syntax. See the **hard_coded.py** script in the **\PythonPrimer\Extra_Samples** folder for the correct syntax.

```
import arcpy

# this defines the current ArcGIS workspace, in this case a
# folder for shapefiles

arcpy.env.workspace = 'C:\\PythonPrimer\\extra_samples\\'

# checks to see if the output exists; if it does, then
# delete it

if arcpy.Exists('c:\\PythonPrimer\\extra_samples\\output\\
                 c_facil_clip.shp'):
    arcpy.Delete_management('c:\\PythonPrimer\\extra_samples\\
                             output\\c_facil_clip.shp')

# Hard coded parameters
arcpy.Clip_analysis('city_facilities.shp',\
                    'central_city_commplan.shp', \
   'c:\\PythonPrimer\\extra_samples\\output\\c_facil_clip.shp')

print 'Completed hard coded clip'
```

Parameters Using Variables

The same Clip function below uses variables instead of the hardcoded values shown above. Even though specific variables are "hard coded," only the variable assignments need to be changed (typically only one time) rather than each occurrence of a specific value throughout the script. The code is also easier to read and can reduce syntax errors when developing code. In later chapters the reader will discover how variables can be dynamically assigned by user input from a custom ArcGIS tool or when the geoprocessing script will be performed in an automated fashion. Using variables for the function parameters make this possible. See the **non_hard_coded.py** script in the **\PythonPrimer\Extra_Samples** folder.

```
import arcpy

# this defines the current ArcGIS workspace, in this case a
# folder for shapefiles

arcpy.env.workspace = 'C:\\PythonPrimer\\extra_samples\\'

infile = 'city_facilities.shp'
clipfile = 'central_city_commplan.shp'
outfile = 'c:\\PythonPrimer\\extra_samples\\output\\c_facil_clip.shp'

# checks to see if the output exists; if it does, then
# delete it

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

arcpy.Clip_analysis(infile, clipfile, outfile)

print 'Completed non-hard coded clip'
```

Add and Modify Geoprocessing Functions

Once an ArcGIS Python script has been started, other variables and geoprocessing functions can be added. The user should consult ArcGIS Help documentation or other sources to learn about the specific geoprocessing function requirements, parameters, and parameter formats. Additional research may be required to discover the function (or functions) and the pre-requisites for unfamiliar geoprocessing routines. Typically, this involves researching ArcGIS Help, Esri user forums, and Internet searches regarding Esri ArcGIS tasks and processes. Since this chapter introduces writing a basic geoprocessing script, the following section outlines some of the steps a code developer can use to learn about a geoprocessing function before adding it to a Python script.

Search ArcGIS Help

A common practice for code developers for ArcGIS is to refer to the ArcGIS Help documentation. This is often a first line of inquiry when developing code and process logic. All of the geoprocessing routines have Python examples and full explanations of each parameter. Make sure to refer to some of the additional documentation that may explain some of the theory and provide insight to the functionality of the algorithms and tools.

Once the documentation is reviewed, the programmer can copy or type the syntax to a Python editor for the given function as well as adding and defining any additional variables required for the function. To copy code from an ArcGIS Help document, the developer can highlight the line (or lines) of the example script, right-click and select **Copy** from the drop down list (or use Ctrl + C keys) and then Paste the selected code into an open Python script using the Python IDLE editor **Edit—Paste** operation (or Ctrl + V).

NOTE: The code developer will want to check the syntax being copied from ArcGIS Help, since sometimes the characters can change (for example, quotes and tabs).

After the syntax is pasted into the Python editor, specific changes can be made to the individual parameters such as using variables defined for the script being developed by the programmer.

ArcGIS Toolbox Aliases

ArcGIS tools (i.e. those found in the ArcToolbox) are organized into toolboxes that share related activities. For example, tools that are used for creating feature classes, working with attribute fields, and creating feature layers and table views can be found under the **Data Management Tools** toolbox. When using these tools in Python, it is good practice to use the alias toolbox name that is associated with the specific tool (i.e. the toolbox alias indicate the specific ArcGIS toolbox where the tool can be found). The general format of this syntax is:

```
arcpy.ArcGISTool_toolalias(<parameters for tool>)
```

For example, the Clip tool has the following syntax.

```
arcpy.Clip_analysis(<input feature class>, <clip feature class>,
<output feature class>, {tolerance}, {units})
```

Notice that the **Clip** tool can be found in the **Analysis** toolbox.

The specific toolbox aliases can be found in the ArcGIS Help topic **Geoprocessing—Managing toolboxes**. An alternative method for writing ArcGIS Tool syntax can be found here: **Geoprocessing—Geoprocessing with Python—Accessing tools—Using tools in Python** (see the ArcGIS section on Tool organization). Instead of using the toolbox alias, a code developer can use the module form.

```
arcpy.toolboxname.tool(<parameters>)
```

For the **Clip** tool shown above, the syntax would be

```
arcpy.analysis.Clip(<input feature class>, <clip feature class>,
<output feature class>, {tolerance}, {units})
```

Syntax for the tool must also be maintained (such as capitalization of characters; see the specific tool help for the exact syntax). The method used to access ArcGIS tools is determined by personal preference; however, one of the two methods must be used when accessing ArcGIS tools. The code developer should adopt one of the two methods and use them consistently in Python scripts.

Summary

Chapter 4 introduced some of the specific ArcGIS Python concepts that are required to begin writing useful Python code to perform geoprocessing. The reader was introduced to the use of workspaces, data paths, and variables to set up data and ArcGIS tool parameters versus hard coding these parameters. Doing so provides the fundamentals to write flexible and extended Python script for multiple tasks, many of which will be covered in later chapters.

Chapter 4 Demo Writing a Clip Features Script

This demonstration combines some of the concepts mentioned throughout this chapter. After completing this demo the reader should be able to understand the following.

ArcGIS Concepts

Import the `arcpy` module
Define a workspace
Refer to ArcGIS Help for the Clip geoprocessing routine
Successfully run the Clip routine

Python Concepts

Create a pseudo-code outline for the script
Create variables for use in the script
Set up `try` and `except` blocks
Add error handling text to the except block

The reader should attempt to write the code on their own to gain practice in developing programming skills. The **Chapter04\Demo4_Clip_Features.py** solution is provided and can be consulted. Exercise 4 will need to use the code developed in Demo 4.

STEP 1 - Create Pseudo-code

a. Start a new Python script window by opening IDLE
b. Name the script **Clip_Features.py** and save it to the **\MyData** folder.
c. Add the following pseudo-code information to the Python script editor. Add your name and the current date at the top of the script. The script up to this point should contain the following:

```
# Demo 4:  Clip Features Geoprocessing Script

# Created by:  <author>
#
# Created on:  <date>
# Updated on:  <date>

'''
This script will perform the ArcGIS Clip routine for a specified
set of feature classes.

'''

# 1. import arcpy, sys, and traceback modules

# 2. Create workspace

# 3. Create variable definitions for geoprocessing functions

# 4.  Add try: and except: blocks

# 5. Add Clip routine (see ArcGIS Help)

# 6. Add exception code within except block
```

STEP 2 **- Start building the script**

a. Add the following information.

Add additional commentary
Import `arcpy` module (and the `sys` and `traceback` modules, see below)

The `sys` and `traceback` modules are used in the exception block used for error handling.

Add a workspace

If needed, change the path for the workspace to the folder that contains the **\PythonPrimer\Chapter04\Data** folder. NOTE: The workspace path may need to be changed depending on where the reader placed the data for the demo/exercise. After adding the information above, the section of script should look similar to that shown below. The section below shows the code modifications.

```
# 1. import arcpy module

import arcpy, sys, traceback

# 2. Create workspace

arcpy.env.workspace = r"c:\pythonprimer\chapter04\data"
```

STEP 3 - Consult ArcGIS Tool Help for the Clip routine

Consult the ArcGIS Tool Help for the Clip geoprocessing function. ArcMap or ArcCatalog can be used to locate the tools. Clip is under **Analysis Tools—Extract—Clip**. Alternatively, use the Search Tab in ArcToolbox and type in each of the geoprocessing functions to locate the tool. Click on the Tool Help to review the specific documentation. Make sure to review the description of the function as well as the required and optional parameters. Scroll to the bottom of the help to see a Python script example using the tool.

The parameters below will be required for the geoprocessing functions in the demo script. Refer to the **\PythonPrimer\Chapter04\Data** folder to find the data. Use **\PythonPrimer\Chapter04\MyData** to store any output files. Load the data into ArcMap to review what the data looks like.

STEP 4 - Add Clip Parameter Variables

Create variables for each of the required parameters. The output feature class can be written to the workspace shown above or a separate variable can be created that point to the **Chapter04\MyData** folder.

Input features (**City_Facilities.shp** - City Facilities); **City_Facilities.lyr** exists that contains some standard symbology. The layer file is not required for the script to function, but can be used in ArcMap for viewing.

Clip features (**Central_City_CommPlan.shp** - Central City Community Plan Boundaries)

Output features – a feature class name, **\PythonPrimer\Chapter04\MyData\City_Facilities_Clip.shp**. The reader can change the path to match a location on their local system.

After creating variables for the **Clip** geoprocess, the script should look similar to the following for this section of code:

```
# 3. Create variable definitions for geoprocessing functions

# can be used to write output to a different
# location than the workspace above

outpath = r"c:\pythonprimer\Chapter04\MyData\\"

infile = "city_facilities.shp"
clipfile = "Central_City_CommPlan.shp"
outfile = outpath + "City_Facilities_Clip.shp"
```

STEP 5 - Add the Clip routine and additional ArcGIS code

a. Add the `try:` block and type in the following syntax for the **Clip** geoprocessing function (see the code below). Make sure to indent the code and use the toolbox alias name or the alternative method described above for the Clip routine. Add a couple of `print` statements that can print out to the Python Shell to monitor the scripts progress.

b. In addition, add the two lines to check to see if the output file exists (i.e. see the "if" statement with `arcpy.Exists` and `arcpy.Delete_management` functions). See below. This will be discussed in subsequent chapters. Notice that `arcpy.Exists` does not have an associated toolbox alias, since `Exists` is an ArcGIS function and not a tool.

Typically, the ArcGIS functions that are not associated with tools typically perform general tasks such as checking for the existence of data. For more information see ArcGIS Help: **Geoprocessing—Geoprocessing with Python—Accessing tools—Using functions in Python** and **Geoprocessing—The ArcPy site package—Functions—Alphabetical list of ArcPy functions**.

```
# 4.  Add try: and except: blocks

try:

    # 5. Add Clip routine (see ArcGIS Help)

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

    print "Starting Clip routine"

    arcpy.Clip_analysis(infile, clipfile, outfile)

    print "Finished Clip routine"
```

STEP 6 - Add Exception Code

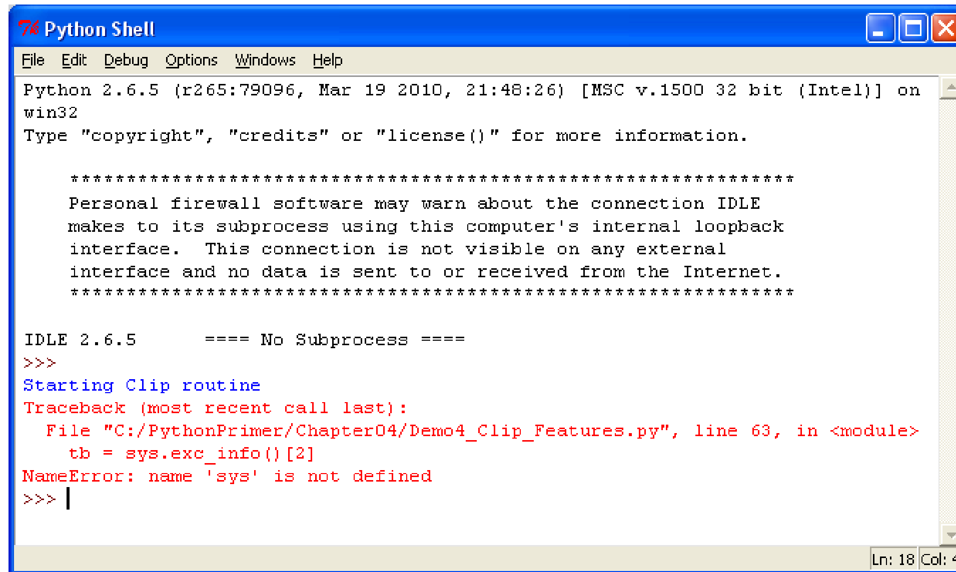
Add the following exception code to the script. Go to the **\PythonPrimer\Chapter04** folder and open the **Exception_code.py** script in Python IDLE. Select all of the text, copy, and then paste it into the Clip Python script being developed for this demo. Use the Python editor **Edit—Copy** (or Ctrl + C keys) and then the **Edit--Paste** (or Ctrl + V) from the Python script editor (of the Clip script being developed) to paste the code within the script. Make sure to place the cursor within the `except:` block before pasting the code. Make sure to indent the exception code as shown below. If the exception code is not indented properly by default, highlight all of the exception text inside the Python script editor, click the **Format—Indent Region** option within the Python script editor. This will indent the entire block of selected code. Note that some lines may extend beyond the window display. This is ok.

The section of script below shows a portion of the indented except code block. Note, that some code lines are "wrapped" to the next line to fit on the page. See the **Chapter04\Exception_code.py** script for the proper formatting.

```
# 6. Add exception code within except block

except:

    #
    http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#//002z00
    00000q000000

    # Notice the indent because of the except block
    tb = sys.exc_info()[2]
    tbinfo = traceback.format_tb(tb)[0]
    pymsg = "PYTHON ERRORS:\nTraceback Info:\n" + tbinfo +
    "\nError Info:\n     " +        str(sys.exc_type) + ": " +
    str(sys.exc_value) + "\n"

    msgs = "ARCPY ERRORS:\n" + arcpy.GetMessages(2) + "\n"

    # prints error messages in the Progress dialog box
    arcpy.AddError(msgs)
    arcpy.AddError(pymsg)

    # prints messages to the Python Shell
    print msgs
    print pymsg
```

Make sure the `sys` and `traceback` modules are in the "import" line at the top of the code. Without these modules the script will result in an error (as a "name" not defined because the

sys and traceback modules would not be imported) if an error did occur in the try block section of code. As you can see above, the sys and traceback routines are used in the exception block. The figure below shows an example of the error that occurs if the sys or traceback modules are not imported.



After completing the script, the file can be saved.

STEP 7 - Click File—Save

STEP 8 - Check the script for Python errors

Click **Run—Check Module** to check for any errors. If the script was written correctly, no error messages should pop up and the Python Shell prompt (>>>) will appear.

If errors appear, try and figure them out or consult the **Demo4_Clip_Features.py** script in the **\PythonPrimer\Chapter04** folder. Once typos and other syntax issues are resolved, click save. Click **Run—Check Module** to make sure the script does not have any errors.

STEP 9 - Run the script

Click **Run—Run Module**.

The script will run and the two print statements will appear in the Python Shell script. If errors occur, note them, and consult the **Demo4_Clip_Features.py** script to remedy any issues. If errors are found, fix them, save the script and re-run the script.

```
>>>
Demo 4 - Clip Features

Starting Clip routine
Finished Clip routine
>>>
```

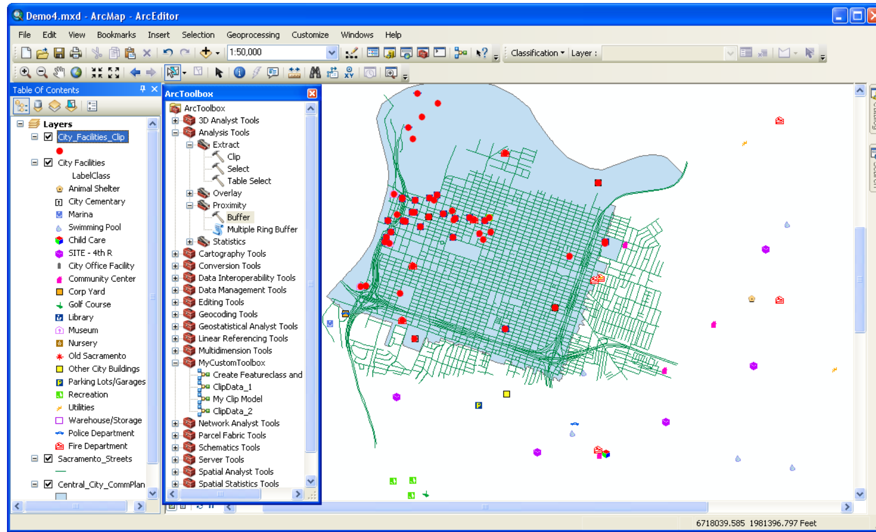
STEP 10 - Review results in ArcMap

Close the Python script and the Python Shell window. This will release any data locks on the files so they can be properly seen and used in ArcMap or ArcCatalog.

Open ArcMap and add the **City_Facilities.shp, Central_City_CommPlan.shp,** (if not already added) and the **City_Facilities_Clip.shp**.

Notice the **City Facilities** shapefile has been clipped (dots) with the **Central City Community Plan** boundary.

The results in the demo can be used in the Exercise below. The **Demo4_Clip_Features.py** script can be found in the **Chapter04** folder.


Demo4.mxd - ArcMap - ArcEditor
File Edit View Bookmarks Insert Selection Geoprocessing Customize Windows Help
1:50,000
Classification Layer :
Table Of Contents
Layers
City_Facilities_Clip
City Facilities
LabelClass
Animal Shelter
City Cementary
Marina
Swimming Pool
Child Care
SITE - 4th R
City Office Facility
Community Center
Corp Yard
Golf Course
Library
Museum
Nursery
Old Sacramento
Other City Buildings
Parking Lots/Garages
Recreation
Utilities
Warehouse/Storage
Police Department
Fire Department
Sacramento_Streets
Central_City_CommPlan
ArcToolbox
ArcToolbox
3D Analyst Tools
Analysis Tools
Extract
Clip
Select
Table Select
Overlay
Proximity
Buffer
Multiple Ring Buffer
Statistics
Cartography Tools
Conversion Tools
Data Interoperability Tools
Data Management Tools
Editing Tools
Geocoding Tools
Geostatistical Analyst Tools
Linear Referencing Tools
Multidimension Tools
MyCustomToolbox
Create Featureclass and
ClipData_1
My Clip Model
ClipData_2
Network Analyst Tools
Parcel Fabric Tools
Schematics Tools
Server Tools
Spatial Analyst Tools
Spatial Statistics Tools
Catalog
Search
6718039.585 1981396.797 Feet

Exercise 4 - Add the Buffer Routine to the Clip Features Script

Using the code developed in **Demo 4**, do the following to add the **Buffer** routine to the script after the **Clip** routine. Review the ArcGIS Help for the Buffer routine.

1. Add variables for the following required buffer routine variables:

   a. **Input features** for the buffer routine will use the variable created for the output clip feature class. (The `outfile` variable used in the Clip routine in Demo 4 is the "input features" for the Buffer routine in Exercise 4).
   b. **Output buffer features** – a feature class name that will represent the "buffered" features (i.e. **\PythonPrimer\Chapter04\MyData\City_Facilities_Clip_Buffer.shp**).
   c. **Buffer distance** – a variable representing a number (e.g. 10, 100, 1000, etc.).
   d. **Buffer units** – a variable representing the word "Feet" or "Meters". This will represent the unit portion of the buffer distance parameter. The buffer distance consists of two parts, a unit measure (10, 100, 1000, etc) and the unit type (feet, meters, miles, etc). The format for the parameter will look like this: "100 Feet", "1000 Meters", etc.

   NOTE: To use the Buffer distance and the Buffer units together, the code developer will need to "concatenate" the two variables. To obtain the format shown above for the buffer distance, the following syntax will be needed: **str(Buffer_distance) + " " + Buffer_units**. `str()` converts the number to a "string" type so that it can be "concatenated" with the word "Feet." Python cannot concatenate a number with a string. The double quotes have a space between each quote. This represents the space between the number and the unit type, thus if **Buffer_distance** is 100 and **Buffer_units** is "Feet", then the above syntax will represent "100 Feet".

   The optional parameters are not needed for the exercise and do not need to be added.

2. Add other print statements if desired.

3. Add an `if` statement similar to the example above that checks to see if the output buffer file exists; if it does, then delete it. This will be placed after the Clip routine and before the Buffer routine. *Remember that a different variable will be used for this if*

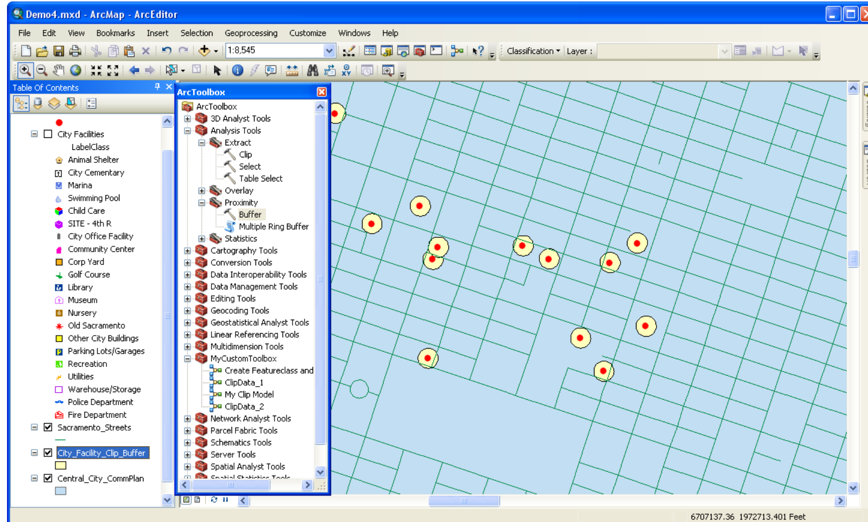
statement that points to the "buffer output features" feature class. (See Demo 4, STEP 5b in the text above).

4. Make sure to Save and "check" the module before running. Save the script to a new name **(\MyData\City_Facilities_Clip_Buffer.py**). Attempt to fix any syntax problems.

5. Run the script to see if the clip and buffer works.

6. After successfully running the Python script, close the Python script editor and the Python Shell and then review the results in ArcMap.

ArcMap should show the clipped and buffered data similar to the following:



Chapter 4 Questions

The following questions focus on the Chapter 4 material.

1. What is the benefit of writing "pseudo-code"?

2. Where in a Python script does the `import` modules need to exist?

3. What is the main difference between setting a variable to a workspace versus setting a variable to a data path?

4. What does "hard coded" refer to? Describe.

5. What is the benefit of using variables as parameters?

6. What is the suggested location for setting variables? Why does this make sense?

7. If a code developer does not know if an ArcGIS tool, function, or geoprocess exists, what are two options can one use to find the tool, function, or geoprocess and learn about its parameters?

The following questions focus on the script created in Exercise 4.

1. If the `outpath` variable (for the Clip Features script) was not used in the script, what would the `outfile` variable be assigned to? Write the syntax.

2. If the workspace environment was not set, would the script be able to function properly without it? Why or why not. (Hint: Try commenting out the `arcpy` environment line and re-running the Python script). Describe what happens and provide a screen shot of the results.

3. Describe what the `arcpy.Exists` and `arcpy.Delete_management` functions do in the script.

4. How do you know the `sys` and `traceback` modules are required for this script? Hint: Look through the entire code written for this exercise.

5. What happens if you take out the `sys` module and re-check the script? Provide a screen shot of the results in the Python Shell.

6. What happens if you take out the `traceback` module and re-run the script? Provide a screen shot of the results in the Python Shell.

7. For the buffer script, write the syntax used to set a variable to the output buffer file.

8. Why do the `buffer_distance` and `buffer_units` variables need to be combined in order to work properly in the Buffer routine?

Accessing Data, Demos, and Code

The reader can obtain the supplemental information for this book at the author's website using the following credentials (which are case sensitive):

http://pythonprimer.urbandalespatial.com/resources

Username: PythonPrimer
Password: PP4AGIS!

Additional information will be provided on this website with any updates, changes, etc.

References

Author's website – **www.urbandalespatial.com**
Jennings, Nathan. "Managing Street Sign Assets: An enterprise geospatial business systems integration solution." *ArcUser*, Winter 2009. Date Accessed: 11.09.2011
http://www.Esri.com/news/arcuser/0109/streetsigns.html

ArcGIS

ArcGIS Resource Center - **http://resources.arcgis.com/**
ArcGIS Web-based Help - **http://resources.arcgis.com/en/communities/**
ArcGIS Blog - **http://blogs.Esri.com**
ArcGIS Forums **- https://geonet.esri.com/community/developers/gis-developers/python**
Geoprocessing script examples and models -
http://resources.arcgis.com/en/communities/python/
Esri Training courses - **http://www.esri.com/training/main**
ArcUser - **http://www.Esri.com/news/arcuser**
ArcGIS Resource Center. Exception Code Snippet. Esri, 2011.
http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#//002z0000000q000000.

Python

Python website – **www.python.org**
Lutz, Mark. Learning Python, 4th Ed. Beijing: O'Reilly Media, Inc., 2009.

Organizations

Esri – **www.Esri.com**
American River College GIS Program - **http://wserver.arc.losrios.edu/~earthscience/**
Sierra College GIS Program - **http://www.sierracollege.edu/academics/divisions/science-math/geography.php**
UC Davis Extension **- https://extension.ucdavis.edu/subject-areas/geographic-information-systems**
Del Mar College **- http://www.delmar.edu/CIS_-_Geographical_Information_Systems.aspx**
City of Sacramento GIS – **www.cityofsacramento.org/gis**
County of Sacramento GIS – **www.sacgis.org**
Cal Atlas – **http://atlas.ca.gov**

INDEX

A

ArcToolbox, 8, 23, 24, 30, 33, 34, 36, 38, 41, 42, 43, 45, 48, 95, 100, 105

B

backslash, 96

C

Check Module, 31, 32, 66, 83, 85, 88, 91, 108

D

data path, 57, 96, 98, 113

E

error handling, 24, 31, 33, 34, 51, 77, 80, 82, 103

F

feature class, 13, 28, 57, 59, 64, 68, 74, 82, 101, 105, 111, 112
feature classes, 49, 65, 68, 74, 76, 95, 100
forward slash, 71, 96

G

geoprocessing, 7, 21, 25, 27, 28, 29, 30, 31, 34, 35, 38, 45, 51, 68, 70, 76, 82, 84, 86, 93, 94, 95, 98, 99, 101, 103, 105, 106

I

if statement, 65, 75, 88, 111

L

Loops
- for loop, 65

M

ModelBuilder, 8, 21, 27, 35, 36, 38, 41, 45, 49, 51, 93

P

Python
- IDLE, 21, 25, 26, 27, 28, 57, 61, 62, 66, 83, 84, 87, 91, 100, 103, 107
- Python Shell, 25, 27, 28, 29, 34, 50, 61, 62, 63, 64, 84, 85, 89, 91, 106, 108, 109, 112, 114

Python Constructs
- capitalization, 32, 64, 101
- comments, 66, 68, 82
- indentation, 65

Python Modules
- arcpy, 13, 28, 51, 68, 74, 76, 77, 95, 96, 100, 101, 103, 104, 106, 113
- os, 76, 77, 87, 95
- sys, 76, 77, 95, 104, 107, 113, 114
- traceback, 31, 33, 34, 76, 77, 87, 104, 107, 113, 114

Q

Quotes
- double quotes, 70, 78, 79, 100
- single quotes, 78
- triple double quotes, 11, 78

S

strings, 65, 68, 70, 78, 79, 98

V

variables, 32, 34, 59, 63, 64, 68, 70, 77, 82, 84, 87, 91, 93, 98, 99, 100, 101, 103, 105, 111, 113, 114

W

workspaces, 36, 38, 69, 70, 71, 74, 93, 96, 97, 98, 101