

A Python Primer for ArcGIS®

Workbook III

Nathan Jennings

Copyright © 2016 Nathan Jennings
All rights reserved.
ISBN: 1533287341
ISBN-13: 978-1533287342

ACKNOWLEDGEMENTS	5
-------------------------------	----------

INTRODUCTION	7
---------------------------	----------

<i>Objectives and Goals</i>	<i>7</i>
<i>Structure of the Workbooks</i>	<i>9</i>
<i>Data and Demos</i>	<i>11</i>
<i>Accessing the Data, Demos, and Code</i>	<i>12</i>
<i>Required Software</i>	<i>12</i>
<i>Older Versions of ArcGIS and Python</i>	<i>13</i>
<i>Reporting Errata</i>	<i>13</i>
<i>Prerequisite Knowledge and Skill</i>	<i>14</i>
<i>Problem Solving</i>	<i>15</i>
<i>Developing Geoprocessing Workflows</i>	<i>16</i>

WORKBOOK III: INTEGRATING AND AUTOMATING PYTHON SCRIPTS FOR ARCGIS	17
---	-----------

Chapter 10 Custom ArcGIS Script Tools.....	19
---	-----------

<i>Overview</i>	<i>19</i>
<i>Python Functions</i>	<i>19</i>
<i>Custom ArcToolboxes.....</i>	<i>23</i>
<i>Toolbox Parameters and Python Script Relationships</i>	<i>25</i>
<i>Considerations to Develop a Custom Toolbox</i>	<i>25</i>
<i>Creating a Custom ArcToolbox</i>	<i>26</i>
<i>Associating a Python Script to the Custom Toolbox</i>	<i>28</i>
<i>Defining Parameters for the Script Tool.....</i>	<i>30</i>
<i>Parameter Display Name</i>	<i>33</i>
<i>Parameter Data Types.....</i>	<i>33</i>
<i>Parameter Properties.....</i>	<i>34</i>
<i>Customizing the Script Tool Interface</i>	<i>40</i>
<i>Modifying the Python Script</i>	<i>41</i>
<i>Adding Python Script Parameter Arguments</i>	<i>41</i>
<i>Adding ArcGIS Messages to the Python Script.....</i>	<i>44</i>
<i>Executing the Script Tool.....</i>	<i>47</i>
<i>Writing Tool Documentation</i>	<i>49</i>
<i>Python Toolboxes.....</i>	<i>53</i>
<i>Creating a Python Toolbox.....</i>	<i>53</i>
<i>Defining Parameters for the Python Script Tool</i>	<i>57</i>
<i>Defining the Execute Function</i>	<i>59</i>
<i>Summary.....</i>	<i>60</i>

Chapter 10 Demos.....	61
------------------------------	-----------

Demo 10a: Custom Script Tool - Clip and Buffer Tool.....	63
---	-----------

Demo 10b: Tasseled Cap using a Python Toolbox	71
--	-----------

Exercise 10 - Custom Script Tool – Develop Your Own Tool 79

Chapter 10 Questions 81

Chapter 11 Python Add-ins 83

Python Add-in Prerequisites 83

Creating a Python Add-in Project 84

Creating a Custom Toolbar 85

Adding an Interface to the Toolbar 87

Adding Functionality to the Python Add-in 89

Installing the Python Add-in 91

Using the Python Add-in 94

Modifying a Python Add-in 96

Debugging a Python Add-in 96

Sharing a Python Add-in 97

Summary 97

Chapter 11 Demo Adding Address Units using a Python Add-in..... 99

Chapter 11 Questions 117

Chapter 12 Automating Geoprocessing Scripts 119

Overview 119

Prerequisites 120

The Batch File 122

Running a Python Script at a Command Prompt 122

Creating a Python Batch File 126

Scheduling the Batch File to Automatically Run the Geoprocessing Script 128

Summary 133

Chapter 12 Demo Using a Batch File to Auto-run a Python Script..... 135

Chapter 12 Questions 137

ACCESSING DATA, DEMOS, AND CODE 139

REFERENCES 140

INDEX..... 141

Acknowledgements

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

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

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

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

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

Introduction

A Python Primer for ArcGIS Workbook III expands the Python and ArcGIS concepts introduced in *Workbooks I* and *II* by introducing Python functions, custom ArcGIS script tools, and Python Add-ins. These are considered advanced topics once the programmer has mastered the fundamentals of both Python and ArcGIS. Python functions are the key to providing clean user interfaces to custom script tools and toolbars. Python functions are “reusable” code since they are called to perform common and repetitive tasks, thus reducing the lines of code one has to write. *Workbook III* concludes with fully automating geoprocessing scripts by implementing the Windows Task Scheduler to run at specific days, times, and frequencies. If the reader is new to GIS and to Python, it is recommended to check out *Workbook I* to gain an understanding and grounding in the fundamentals. For those who have some GIS background and programming experience *Workbook II* can be the launch point to dive into some of the core concepts of ArcGIS Python programming. The objectives and goals as well as the sections on problem solving and developing geoprocessing workflows are repeated here for those skipping *Workbook I* since these concepts are at the heart of the *Workbook* series.

Objectives and Goals

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

The workbooks accomplish the following objectives:

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

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

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

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

Structure of the Workbooks

A *Python Primer for ArcGIS* is divided into three separate workbooks so the newcomer to Python and ArcGIS can begin with *Workbook I* and work through all of the material and obtain a firm grounding in Python programming as well as become more familiar with the ArcGIS geoprocessing structure. Those that already have a fundamental understanding of Python and ArcGIS can begin with *Workbook II* to gain more insight into common geoprocessing tasks that many GIS professionals encounter. *Workbook III* takes the fundamentals and the common geoprocessing tasks a step further and provides some guidance to relate a Python script to a custom ArcToolbox 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 good user habits such 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. A custom-built algorithm implemented using Python syntax is described as well as using the Spatial Analyst extension and `sa arcpy` 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 programmer 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 shows how coders 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 Task Scheduler 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 in the *Accessing the Data, Demos, and Code* section at the back of the book. The supplemental material is organized as follows: **\PythonPrimer\ChapterXX**. Within each chapter the **Data** folder contains the data files required for the demo and/or exercise. Data files can be shapefiles, file geodatabase feature classes or tables, or standalone tables (e.g. dBase format), or TIF or ERDAS (.img) images. ArcMap documents (.MXD) can be used as referenced or renamed for readers to modify and save their own work. A **MyData** folder is also provided so that readers can save their own work for demos and exercises. All of the data and ArcMap documents will be in ArcGIS 10 or later format. NOTE: The ArcMap documents reference the **\PythonPrimer\ChapterXX** structure above. If the reader changes this folder structure, the ArcMap documents provided by the author may need to have the source files in the Table of Contents revised to the new location. The scripts have been tested on Windows 7 32-bit and 64-bit operating systems. The reader may need to make some additional adjustments to data paths on 64-bit Windows systems.

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

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

County of Sacramento – parcel and street subsets

CalAtlas – Landsat Thematic Mapper (TM) satellite imagery subset

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

Accessing the Data, Demos, and Code

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

Required Software

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

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

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

Older Versions of ArcGIS and Python

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

Reporting Errata

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

Prerequisite Knowledge and Skill

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

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

Problem Solving

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

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

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

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

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

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

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

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

Developing Geoprocessing Workflows

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

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

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

Workbook III: Integrating and Automating Python Scripts for ArcGIS

Workbook III expands the capabilities already seen in *Workbooks I* and *II* by associating Python scripts with custom ArcGIS and Python tools, creating Python Add-ins, and automatically executing scripts by scheduling Windows tasks and batch processes.

Chapter 10 introduces the construction and use of Python functions to develop custom ArcGIS or Python tools. Creating help documentation for script tools is also discussed.

Chapter 11 reviews the process of building and using Python Add-ins to provide unique functionality to custom toolbar elements.

Chapter 12 illustrates the development of the batch routine which can be used in the Windows Task Scheduler to automatically execute a Python script at specific dates, times, and frequencies (e.g. daily, weekly, monthly, etc.).

Completing all three workbooks provides the reader with a broad skillset of Python scripting to implement geoprocessing routines using a variety of methods for the ArcGIS platform.

Chapter 10 Custom ArcGIS Script Tools

Overview

Up to this point scripts have been written to run (or started) from within the Python IDLE environment. Since *A Python Primer for ArcGIS Workbooks* focus on developing and using Python scripts for geoprocessing tasks, one might ask if Python scripts can be run or initiated from within the ArcGIS application. The short answer is yes! The programmer can create a custom ArcToolbox or a Python script toolbox to store a custom graphical user interface (GUI) that points to a Python script. Behind these interfaces is the implementation of Python functions which are keys to connect the Python script to the script tool. Chapter 10 begins with a brief overview of the Python function construction and use and then illustrates how the Python function can be used to relate scripts to custom tools or Python Add-ins. Custom ArcGIS script Tools and Python script tools are discussed and the reader can view the demos to see how each of these tools is implemented and then has an opportunity to create a custom script tool on his/her own.

Python Functions

Most of *A Python Primer for ArcGIS Workbooks* discussions and examples focuses on using fundamental Python programming structures. Python functions can be considered a “next step” in programming development since functions help “modularize” and recycle blocks of code that may be implemented many times within a Python script or used with separate Python scripts that require similar functionality. In addition, since the developer may wish to create some custom behavior on the tool interface, it is helpful to have an idea of how Python functions work before putting them to use.

Python functions are chunks of code that sit at the top of a Python script and are “called” within the body of a Python script. The Python function can help reduce the number of coding lines to write when implementing common tasks (for example implementing a custom print routine or a series of calculations or operations where a result needs to be used in subsequent lines of code).

The following shows the basic construct of a Python function typically found at the beginning of a Python script. See the **python_function.py** script in the Chapter10 folder.

```
def FunctionName({optional arguments}):  
    # body of function
```

To implement the Python function code, it must be called from within the main body of a script.

```
import arcpy, os, sys, traceback  
  
# function definition  
  
def CalcValue(anInteger):  
    resultValue = anInteger + 1  
  
    return resultValue  
  
try:  
    print "Computing a value..."  
  
    x = 0  
  
    print "Start value is: " + str(x)  
  
    # function call, passing an argument  
    # and accepting a returned value  
  
    returnedValue = CalcValue(x)  
  
    print "Returned Value is: " + str(returnedValue)  
  
except:
```

Immediately after the `import` line, the Python function is defined. Within the `try` block the `CalcValue` function is called passing the argument (`x`), an integer, to the function. Next, the line of code within the `CalcValue` function is processed and then the `resultValue` is returned to the main body of the code. The `returnedValue` variable within the main body is assigned the value of `resultValue` and then is used in a print statement. Notice that in the function definition, the variable `anInteger` is used instead of `x`. The variable name used in the Python function is often different than the variable used in the function call because the function call may include many different variable names when the function is called multiple times within the code body. In this particular case, all of the variables passed to the function must represent integers because the function is expecting an integer so that the code in the function body can be processed correctly.

In this example, an argument was passed to the function. In some cases functions may not require arguments as shown below.

```
def aFunction():  
    # body of function
```

In other cases, there may be many arguments and so the individual arguments are separated by commas. Note when a function has several arguments, the proper argument order and type of value must be adhered to.

```
def aFunction(arg1, arg2, arg3):  
    # function body
```

Here is another portion of the script the implements the `PrintThis` function where two arguments are passed and has no value returned to the main body of the script.

```
import arcpy, os, sys, traceback  
  
# other functions go here  
  
def CalcValue(anInteger):  
    resultValue = anInteger + 1  
  
    return resultValue  
  
def PrintThis (aNumber, anotherNumber):  
  
    print "The print function prints: " + str(aNumber) + \  
        " and " + str(anotherNumber)  
  
try:  
  
    # other code goes here  
  
    # function call, passing two arguments  
    # and no value returned  
  
    PrintThis(x, returnedValue)  
  
except:
```

In this script two Python functions are provided so that if any calculation needs computing or a standard print statement is required, the respective function calls can be made without having to rewrite the same lines of code. These are two simple examples of the Python function. In

reality, the code within a Python function may have many more lines of code and have more complex syntax. The construction of a Python function depends on its use. In the cases of the custom ArcTools or Python tools, functions are often used to set initial values or tool parameter functionality within the graphical user interface.

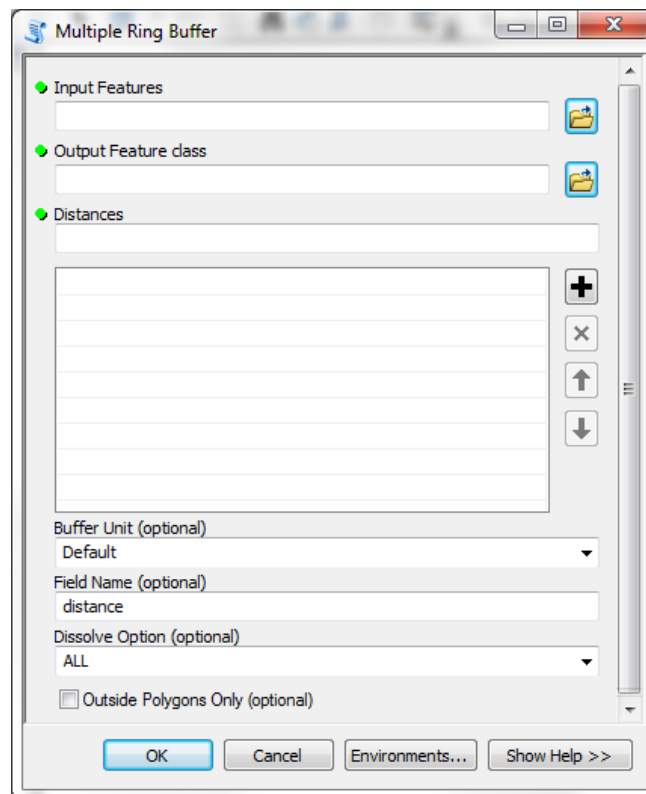
Script Tools

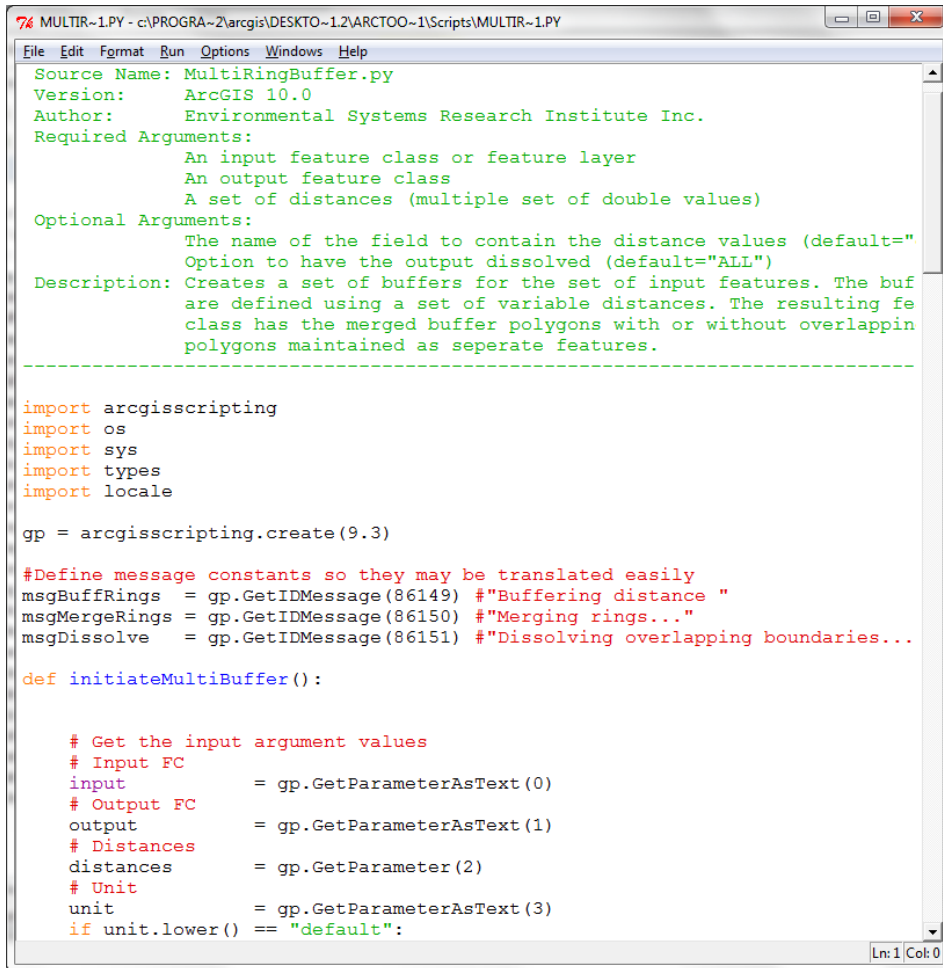
Script tools are ArcTools (created by Esri or custom built by an ArcGIS user) and may include one or more existing ArcTools with custom Python coding or use `arcpy` libraries and routines to implement custom processing. Essentially, script tools are implemented by an ArcGIS user by clicking on the tool, filling in the parameters, and clicking the OK button. Prior to ArcGIS 10.1, only custom ArcToolboxes could implement script tools. With ArcGIS 10.1+ Python Toolboxes can be used to create, modify, and organize the Python code required to implement the script tool. The following table compares the primary differences between custom ArcToolboxes and Python toolboxes to process custom Python code. The table is adopted from the ArcGIS Help comparing custom ArcToolboxes with Python ArcToolboxes.

	Custom ArcToolbox	Python ArcToolbox
	(9.0+)	(10.1+)
Code/Tool Management	Core Python script, GUI, and validation code in separate files	Python script, GUI, and validation code in single Python script (.pyt) file
Code Changes	Modify separate Python files	Modify a single source Python file
Support Other Tools	Can use ModelBuilder Tools and other code tools (e.g. .NET)	Does not support ModelBuilder Tools or other code tools; can call other toolboxes
License Management	Not available	Can use a license routine to check for proper ArcGIS licenses, if required

Custom ArcToolboxes

An ArcToolbox can run an Esri function (often developed using C++ or similar programming language), a model developed from within ModelBuilder, and/or a custom Python script. Some of the existing ArcToolbox tools are developed in Python where the source code is exposed to the ArcGIS user. A common example referred to by Esri that uses Python is the **Multiple Ring Buffer** routine found within the **Analysis Toolbar—Proximity** Toolset. The tool interface and an excerpt from the script are shown below. The Python script for the **Multiple Ring Buffer** tool can be viewed by right-clicking on the tool and choosing Edit. Note the script uses older 9.3 ArcGIS and Python syntax which is still supported in later versions of ArcGIS.





```

File Edit Format Run Options Windows Help
Source Name: MultiRingBuffer.py
Version: ArcGIS 10.0
Author: Environmental Systems Research Institute Inc.
Required Arguments:
    An input feature class or feature layer
    An output feature class
    A set of distances (multiple set of double values)
Optional Arguments:
    The name of the field to contain the distance values (default="")
    Option to have the output dissolved (default="ALL")
Description: Creates a set of buffers for the set of input features. The buffers
are defined using a set of variable distances. The resulting feature
class has the merged buffer polygons with or without overlapping
polygons maintained as separate features.

-----

import arcgisscripting
import os
import sys
import types
import locale

gp = arcgisscripting.create(9.3)

#Define message constants so they may be translated easily
msgBuffRings = gp.GetIDMessage(86149) #"Buffering distance "
msgMergeRings = gp.GetIDMessage(86150) #"Merging rings..."
msgDissolve = gp.GetIDMessage(86151) #"Dissolving overlapping boundaries..."

def initiateMultiBuffer():

    # Get the input argument values
    # Input FC
    input = gp.GetParameterAsText(0)
    # Output FC
    output = gp.GetParameterAsText(1)
    # Distances
    distances = gp.GetParameter(2)
    # Unit
    unit = gp.GetParameterAsText(3)
    if unit.lower() == "default":

```

Source: Esri, 2011. MultiRingBuffer Tool.

Providing a custom tool to an ArcGIS user provides a convenient method to “deploy” a script since the end user may not understand how a Python script functions. A custom tool is created from within ArcCatalog or ArcMap and is stored within a folder. A user opens the custom tool and fills in the parameters in the same way as any other geoprocessing tool found in the ArcToolboxes. Python scripts that have been configured to accept tool parameter input (i.e. the values that are entered into tools through a tool interface) can be associated to the custom tools to perform custom geoprocessing tasks.

Toolbox Parameters and Python Script Relationships

In the MultiRingBuffer tool shown above the individual inputs correspond to script elements as “Get Parameters.” The `GetParameterAsText` routine connects the tool input parameters to the Python script. (Although an “older” method of using `GetParameterAsText` is used above (i.e. the former `argisscripting` Python module that uses the `geoprocessing (gp)` object), the same syntax holds true with the `arcpy` module—

`arcpy.GetParameterAsText()` when using ArcGIS 10.0+). When a user enters in values to the tool parameters and clicks OK, the parameter values are passed to Python and stored as variables where they will then be referenced by subsequent Python code. Instead of assigning variables to specific values within the Python script itself (for example, with a standalone script), the parameter values are assigned to the `GetParameterAsText(<index>)` routine and a specific index number is used to index the value it represents for the specific tool parameter. The parameter indices start with “0” to index the first tool parameter. Sequential indices (0, 1, 2, etc) are then used for tools that contain multiple parameters.

Considerations to Develop a Custom Toolbox

Developing a custom toolbox that is associated with a Python script is often a “second phase” of code development since considerable code development and testing has already been spent to create a functioning standalone script. If the code developer needs to develop a “front end” (i.e. a GUI) for the script, then some development and testing on the tool GUI and parameters are required. Like developing code, developing the tool GUI and parameters is often an iterative and time consuming process.

Before getting started, the code developer should consider which aspects of the existing code will be used as tool parameters. Typical values will include workspaces or data paths for input and output, feature class or table names, numeric values, field names, and possibly options that can be used for “Yes/No” (True/False) items. Keep in mind that not every unique feature class, table, or other value that is implemented in the script needs to have a parameter in the tool. In many cases various datasets are only used as intermediates and may not need to be included in the tool parameters. The code developer will also need to consider the data type of each parameter (feature classes, tables, images, workspaces, unique lists of values, numbers, etc). These items will be used to define the tool parameters within the tool interface.

NOTE: All parameters in a tool get assigned to “strings” when they are passed to a Python script using the `GetParameterAsText` routine.

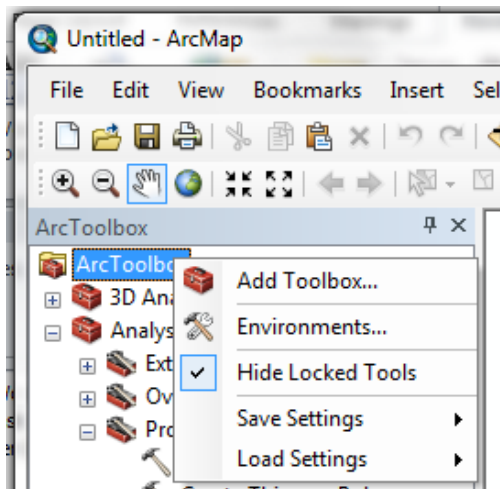
Creating a Custom ArcToolbox

The basic steps involved to create a custom tool that uses a pre-existing and properly configured Python script are:

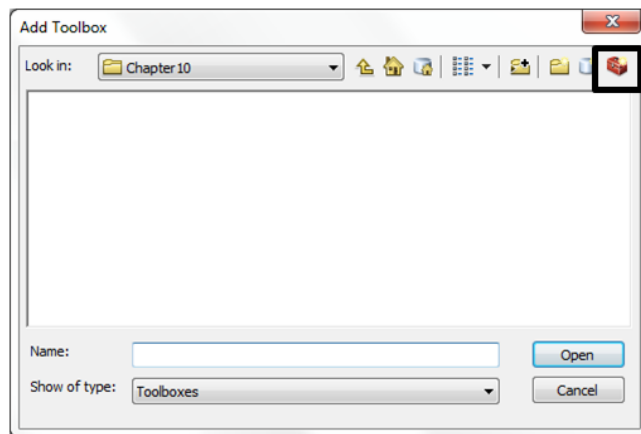
1. Create a custom toolbox and “Add” a script tool
2. Associate the Python script to the tool
3. Add and configure the tool parameters the script will need to execute
4. Modify the Python script as required (e.g. use the `GetParameterAsText` routine to accept and use tool parameters and other script changes)
5. Create help documentation for the tool

The reader should refer to this section of the chapter when reviewing the demos and exercise.

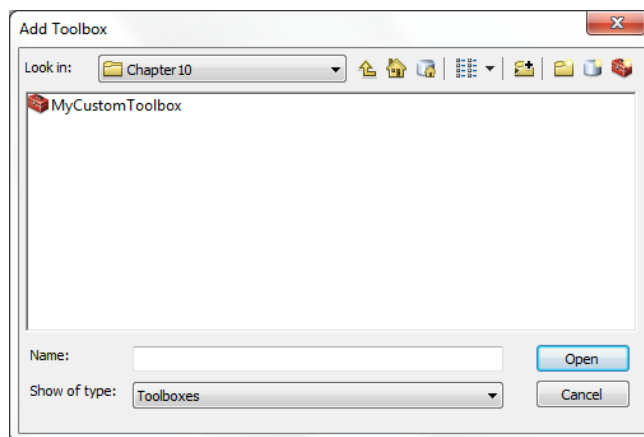
Creating a custom ArcToolbox can be performed in ArcMap by right clicking on the ArcToolbox. Make sure the ArcToolbox is showing. Alternatively, a new toolbox can be created in ArcCatalog by navigating to a desired folder, then right clicking and selecting **New—Toolbox**.



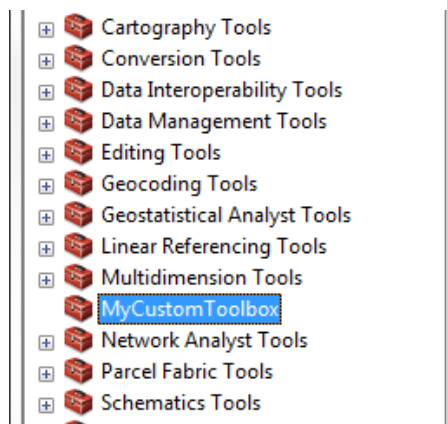
Browse to a desired folder to create the new toolbox.



Click the “New Toolbox” button in the upper right corner of the window and change the name.

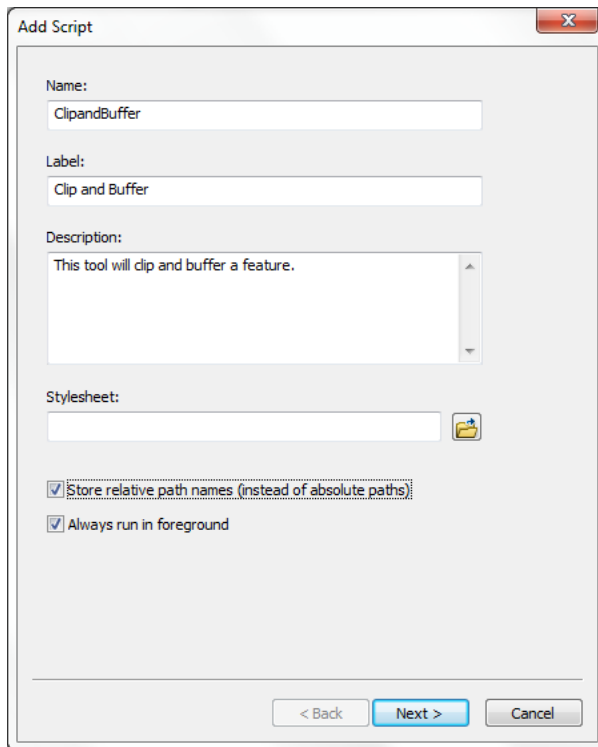


The new toolbox is added to the ArcToolbox.



Associating a Python Script to the Custom Toolbox

After the custom toolbox is created, the Python script can be added to it by right-clicking on the toolbox and choosing **Add** from the menu. A tool wizard, similar to the figure below, will appear. A series of steps are required to associate the script to the tool. Refer to the Chapter 10 material provided for this chapter and the Demo 10a **ClipandBuffer.py** script.



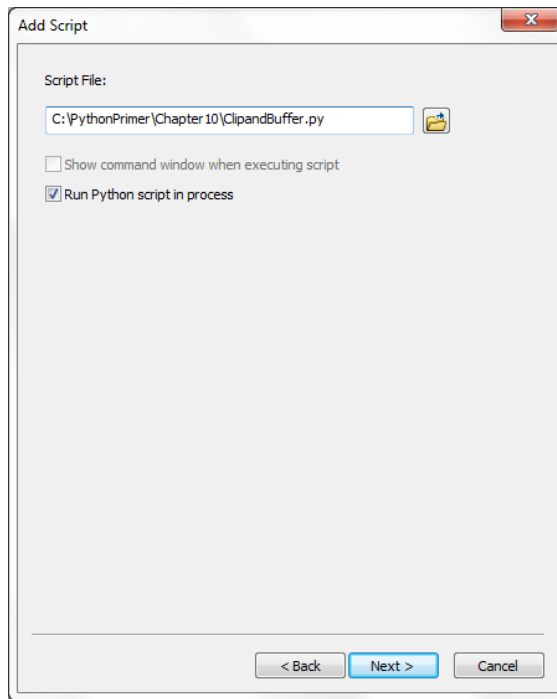
The screenshot shows the 'Add Script' dialog box with the following details:

- Name:** ClipandBuffer
- Label:** Clip and Buffer
- Description:** This tool will clip and buffer a feature.
- Stylesheet:** (Empty field with a browse icon)
- ☒ Store relative path names (instead of absolute paths)
- ☒ Always run in foreground
- Buttons:** < Back, Next >, Cancel

The initial dialog box asks for a script name and label. The script name represents a descriptive name of the tool and “cannot” contain spaces, dashes, or underscores. The label is the name that will show up in the custom toolbox (and hence becomes the “script tool”) and can have spaces. The user can add a brief useful description about the functionality of the script. This description shows up in the tool help for the specific script tool. It is recommended to always check the box to store relative path names. This helps alleviate problems with moving scripts and toolboxes to different folder locations.

Custom tools are also always run in the foreground by default. Tools that use foreground processing will execute and continue until the tool completes. Foreground processing will prevent the user from using the application (such as ArcMap) until the tool completes. In addition, a progress dialog box appears showing the progress of the tool. The reader is

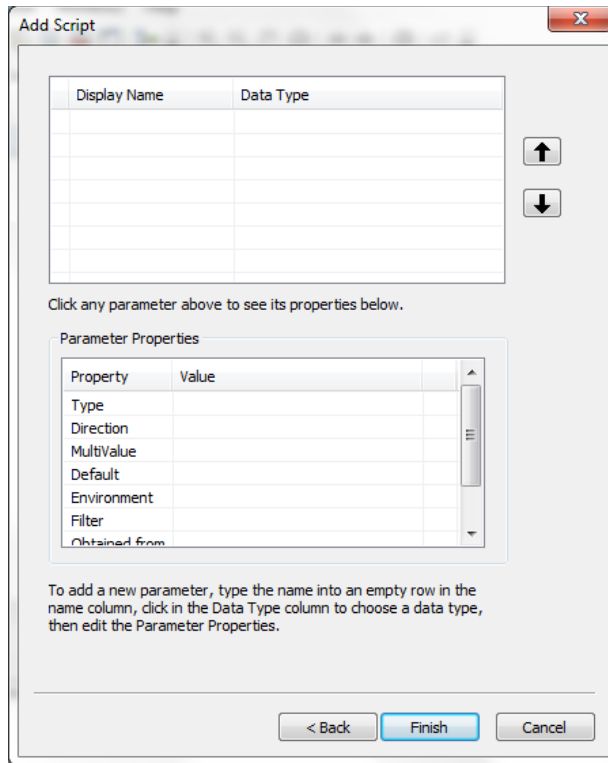
encouraged to consult the ArcGIS Help for more information. For *A Python Primer for ArcGIS*, this property will remain “checked” and the tool will be executed using foreground processing. After clicking **Next** on the wizard, the Python script file becomes associated with the script tool. Browse for the appropriate Python script file. The “Run Python script in process” should be left “checked.” This option helps the scripts run more efficiently.



It is recommended that the script is placed in some logical folder so that it can be easily found if additional script modifications are required.

NOTE: If the user moves the custom toolbox to a different location, the Python script does not move with it. The Python script will remain in its location until the user changes this dialog box to associate the tool to the new location. This is one of those “drawbacks” with custom script toolboxes.

Clicking **Next** in the wizard provides the opportunity to define “parameters” for the script tool.



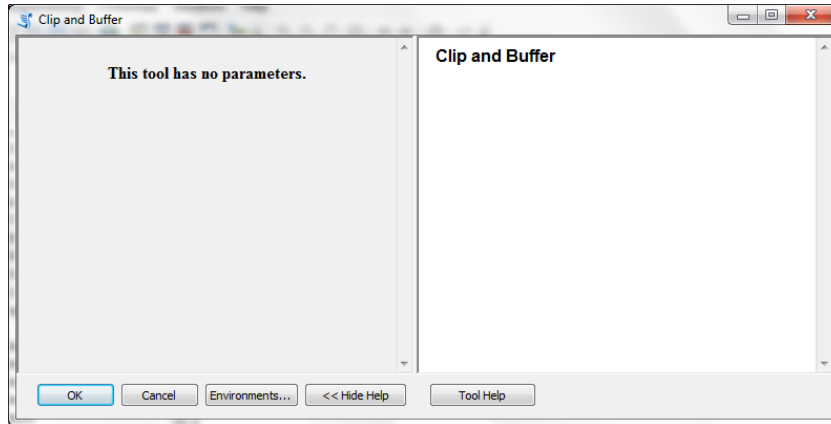
Defining Parameters for the Script Tool

Script tool parameters represent the values a geoprocessing tool requires to function (i.e. the values that are passed to the Python script which is linked to the tool interface). The parameters need to have a descriptive name and the proper data type defined. Not only will these definitions insure the proper data is passed to the Python script, but they can help limit the kinds of values a user can enter (or choose) in the tool interface. For example, if a code developer created a string data type for the buffer units, but did not limit the choices to a specific list of values that contain ‘Feet’, ‘Meters’, or ‘Miles’, the tool’s user might type in the value ‘FT’ instead of an appropriate value from the list. The value ‘FT’ may not be recognizable for the specific ArcGIS tool that requires it, so the program will not function properly.

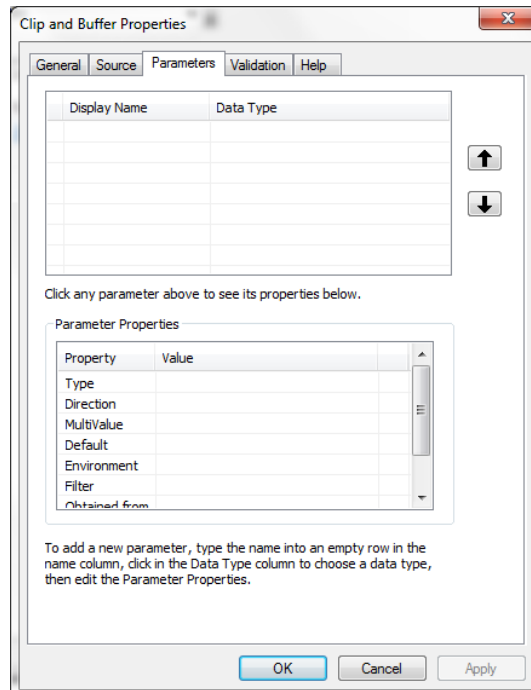
To create a well-designed tool interface, the following process is required:

1. Create a Display Name (the parameter heading in the tool interface)
2. Set the proper Data Type for the parameter
3. Set Parameter Properties as required

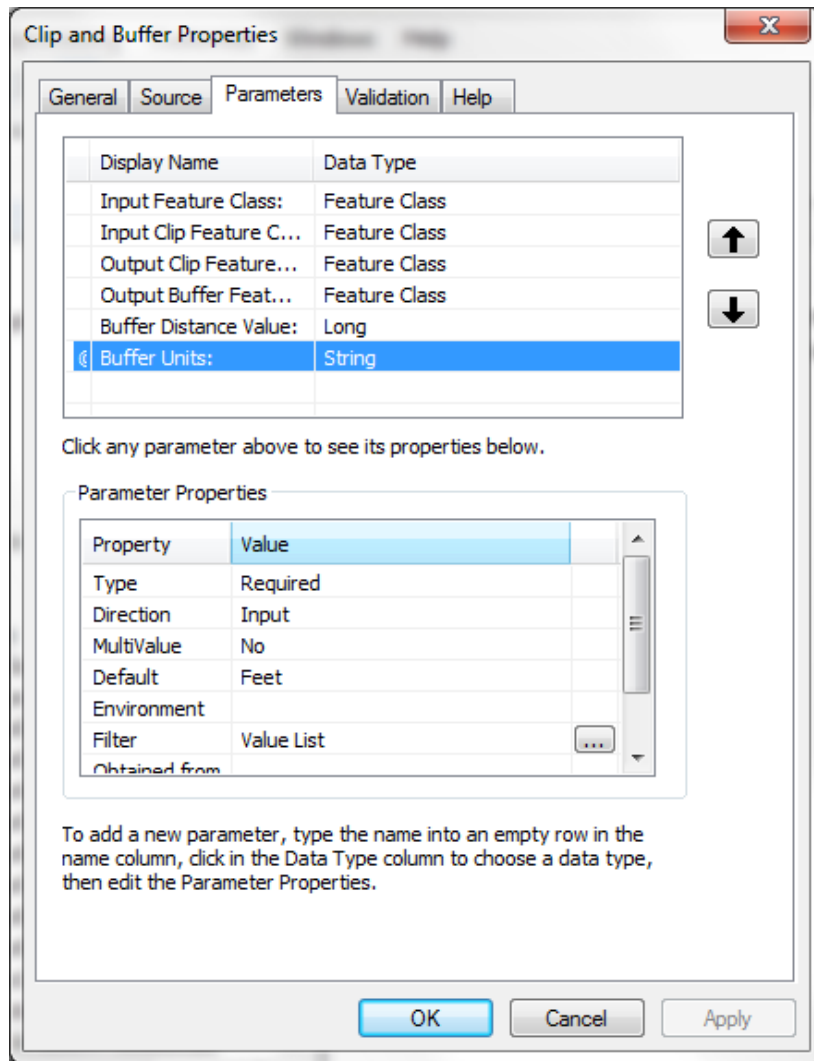
Note: The tool properties can be viewed and modified by right-clicking on the tool name and selecting Properties after the script has been associated with the respective toolbox. If a script does not have any defined parameters, a blank tool interface is loaded and looks like the following figure.



In addition, the Parameter properties will also be blank.



The following figure shows the parameters created for the custom **Clip and Buffer** tool when the Parameters Tab is displayed and the appropriate tool parameters have been filled in.



Notice the Display Name and Data Type columns at the top of the dialog box. The Display Name is the value that shows up as a parameter heading. The Data Type shows the type of value that the parameter represents. Defining the data types can be challenging depending on how “flexible” the code developer decides to make the script. For example, an initial development of a tool user interface may only limit user input to shapefiles, so a shapefile data type will likely be used for some of the parameters. If, on the other hand, the code developer wants the user to be able to choose a shapefile or a geodatabase feature class, then the feature class data type might be used. Also notice the Parameter Properties in the middle section of the dialog box includes some additional parameter setting that may be modified to limit how

the parameters are used and can enforce the selection of appropriate values for the parameters. See the ArcGIS Help document under **Geoprocessing—Creating Tools—Creating script tools with Python scripts—Setting script tool parameters**.

Each of the primary Parameter tab options is explained below.

Parameter Display Name

The display name is simply a short text phrase that represents the parameter's heading and indicates the kind of information required for the parameter. For example, a Parameter Display Name might be assigned to *"Input Polygon Feature Class:"* to indicate to the user that a polygon feature class should be chosen. The actual Data Type and parameters set up by the code developer will limit the specific kinds of information, formats, or values a user can use or choose as the parameter.

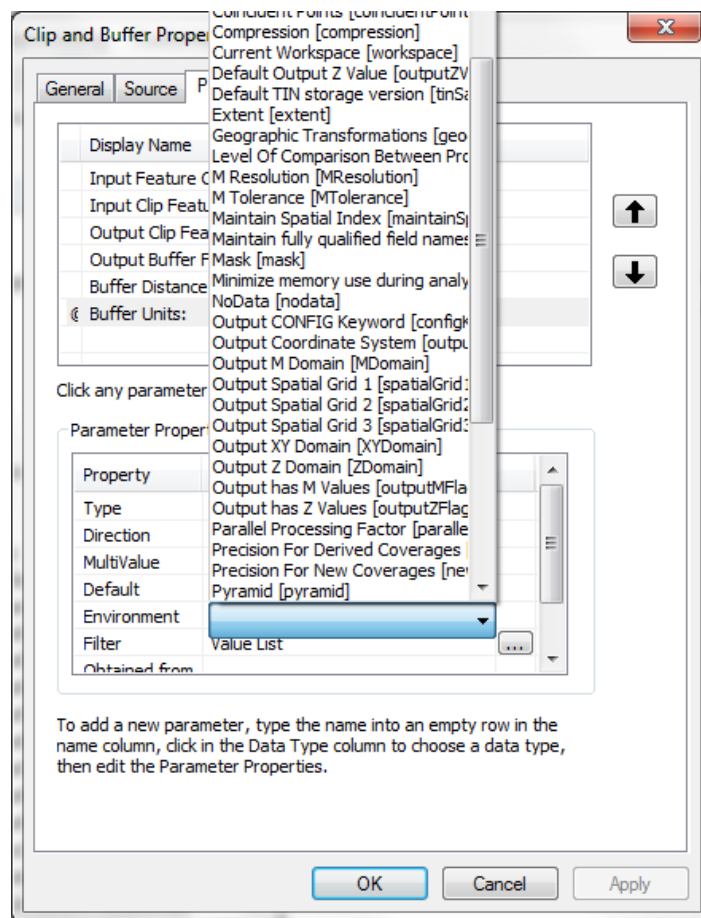
Parameter Data Types

ArcGIS includes a large number of data types a code developer can choose. For a complete list the reader is directed to the ArcGIS document found in the in the ArcGIS Help topic **Geoprocessing—Geoprocessing tool reference—Geoprocessing tools supplementary tool parameters**. In most cases typical data types used in custom tools will be strings, integers, feature classes, tables, folders, and workspaces.

Parameter Properties

Parameter Properties help limit the values filled into the parameter so the proper data types, formats, values, etc. can be chosen by the tool's user. The Parameter Properties include the following:

- a. **Type – (Required, Optional, Derived)**. Indicates if this parameter is *required* for the tool to function, *Optional* where the user can choose to set it or not, or *Derived*, which indicates if the parameter will be derived from another parameter and will NOT be shown in the tool's dialog box. For example, an intermediate feature class that is created as part of the process may be derived and will not be visible to the user interacting with the tool. Derived data types are always *Outputs*. (See Direction below).
- b. **Direction – (Input or Output)**. Inputs are datasets required to process the tool routine. Outputs are datasets used as intermediate or as final datasets that result when a tool routine completes executing (such as a feature class, table, or image).
- c. **Multivalue – (Yes or No)**. If the parameter has a dropdown list or list of check boxes for values for the user to choose, then *multivalue* is set to Yes. An example of a multivalue may include choosing multiple values from a list (through a check box and a Value List) or adding multiple feature classes to a list. See the ArcGIS help document for the Spatial Join routine for more details and an example of using a multivalue (e.g. **Data Management Tools—General—Append** or **Analysis Tools—Overlay—Spatial Join**).
- d. **Default** – If a code developer wants to have a default value show up in a tool parameter, then this property will be assigned a value. If the value is in a *Value List*, consistent name syntax is recommended. For example, if a *Value List* contains the value 'FEET', then 'FEET' will be used in the default property and not 'Feet'. (This property is also good for setting specific file types such as '*.dbf').
- e. **Environment** – If the parameter references an environment setting, this value is populated such as an Extent, Cell size, Pyramid Layers, etc. The screenshot below shows some of the possible options for the *Environment* property.

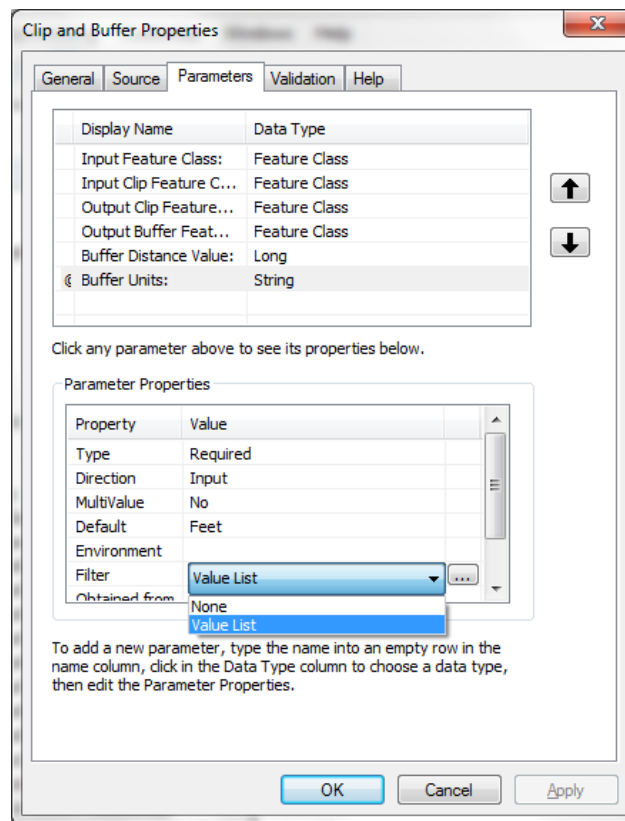


- f. **Filter – (None, Value List, Range, Feature Class, File, Field, Workspace).** A filter is a condition that can be placed on the parameter to limit its options. For example, if a parameter *Type* is a feature class, then the *Filter* property can be set to '*Feature Class*' to only allow feature classes to be used for this parameter. This can help the end user choose appropriate values for parameters. After the *Filter Type* is chosen a space to add specific values or check boxes for specific types can be used. Click on the '...' (ellipse) when it appears in the *Filter* option. The following table summarizes the different *Filter Types* and possible values that can be used for each type.

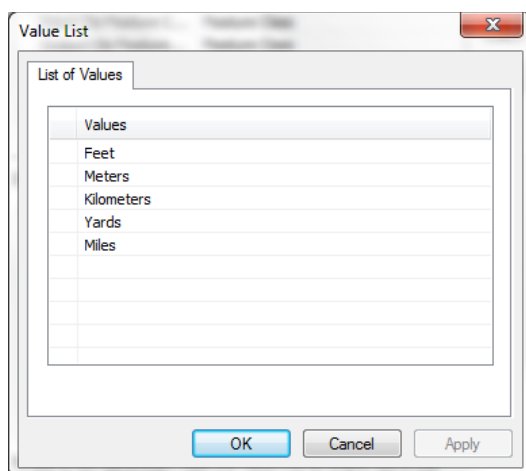
Filter Type	Values
Value List	Typically strings or numbers, for example, ('Feet', 'Meters', 'Yards', etc.) or (1,2,3,4,5, etc).
Range	Minimum and Maximum ranges for numbers. For example, 0-10.
Feature Class	Can be any supported feature type. Possible options that show up as check boxes include: Point, Multipoint, Polygon, Polyline, Annotation, Dimension, Sphere, Multipatch. More than one type can be chosen.
File	A list of semicolon separated file suffixes (TIF, IMG, TXT, CSV, SHP, etc.) and does not include the '.' (dot), or example: <i>tif; img; txt; csv; shp</i> , etc.
Field	A list of allowable field types such as Short, Long, Blob, Raster, GUID, etc. More than one value can be chosen.
Workspace	A list of allowable workspace types, such as the file system (shapefiles, ArcInfo Coverages), local databases (file or personal geodatabase, or enterprise databases (ArcSDE) geodatabases). More than one type can be chosen.

- g. *Obtained from* – This property sets a dependency on another parameter. For example, a *Field* parameter may have the *Obtained from* property set to a *feature class* or *table* parameter to indicate that the fields come from the respective feature class or table. Fields represent the attribute table in a feature class or table.
- h. *Symbology* – used only with output parameters and points to the location of a (.lyr) file and is used for displaying the output in ArcMap

Referring to the **Clip and Buffer Tool** Properties dialog box below and some of the properties mentioned above, the Parameter Properties for the **Buffer Units** parameter (*string* data type) is limited to a list of specific values (*Filter* property is set to a *Value List*) and the *Default* value is 'Feet'.



The Value List contains a unique list of unit names the user can choose from:

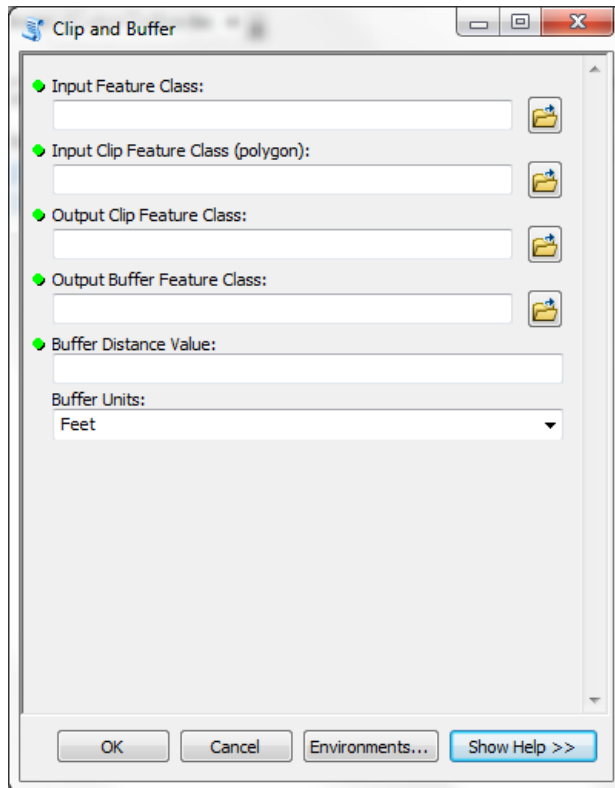


The table below summarizes the parameters for the **Clip and Buffer** script tool discussed in this chapter.

Parameter Table for the Clip and Buffer Tool

Display Name (Parameter)	Data Type	Parameter Property	Value	Additional Settings
Input Feature Class:	Feature Class (Required)	Use defaults		
Input Clip Feature Class (polygon):	Feature Class (Required)	Filter	Feature Class	Check only Polygon from list
Output Clip Feature Class:	Feature Class (Required)	Use defaults except Direction	Direction = Output	
Output Buffer Feature Class:	Feature Class (Required)	Use defaults except Direction	Direction = Output	
Buffer Distance Value:	Long (Integer type), (Required)	Use defaults	(optional) set Default to a value (such as 100)	*This value might be useful, depending on the kind of buffer.
	String	Filter	Value List containing the values (Feet, Meters, Kilometers, Yards, Miles)	Click the Value List name or '...' to add the specific values to the list

The **Clip and Buffer** tool interface looks like the following after all of the parameter properties are defined. The “green” dot indicates a tool parameter is required (the default setting for the Type in the parameter properties).



Customizing the Script Tool Interface

The discussion above illustrates the primary parts of a geoprocessing script tool interface that can be created and changed based on the desired functionality the programmer wants to provide to the end user. The specific modifications to the custom **Clip and Buffer** tool example shown here can be found in **Demo 10a**. The next section discusses how an existing Python script can be augmented to accept values entered into the custom script tool interface.

Modifying the Python Script

To run a Python script from a custom tool interface, the Python script needs some minor modifications compared to a script that is intended to run as a “stand alone” script. Remember, a standalone script is one that can be executed independently of interacting with the ArcGIS interface (such as within ArcMap or ArcCatalog). Up to this point in *A Python Primer for ArcGIS*, the code developer has been writing and testing code by setting specific workspaces, data paths, feature class names, and other parameters so that the script can be run from the Python IDLE or command line (i.e. “stand alone”). If a script is ready to be deployed to an end user in ArcGIS using a custom tool interface, some changes need to be made to some of the variables used in the Python script.

To accept values from an ArcGIS custom tool, the script must have variables defined to translate and accept user input instead of the program developer specifically defining variables. Values from custom script tool parameters are assigned to Python variables using the `GetParameterAsText()` routine. The `GetParameterAsText()` routine is an ArcGIS construct that uses an index to identify each tool parameter and assign it to an appropriate variable in the script. For additional information, see **Understanding script tool parameters** in the ArcGIS Help.

Adding Python Script Parameter Arguments

Python can accept script parameter arguments so that the parameters can be typed or passed from an application (in this case ArcGIS) to the script. The script parameter argument takes the form

```
variable_name = arcpy.GetParameterAsText(<index>)
```

where `index` is the parameter number in the custom ArcGIS tool beginning with zero (0). For example the first parameter in the Tool user interface is set to `arcpy.GetParameterAsText(0)` and are sequentially ordered from top to bottom in the tool interface. Parameters that are expected to be assigned variables in the Python script will use their respective parameter index.

For the **Clip and Buffer** tool example, the input feature class is the first parameter and thus will have its respective index referenced in the `GetParameterAsText(0)` routine when assigning the parameter value to a variable in the associated Python script.

```
# input feature class
# used in the stand alone script, commented out here
# GetParameterAsText used below to accept tool input
# infile = 'City_Facilities.shp'

# infile is assigned the "first" parameter from the ArcGIS Tool

infile = arcpy.GetParameterAsText(0)
```

The syntax above allows any valid feature class to be entered by the user which will then be assigned to the `infile` variable. A feature class is assigned because the tool parameters were defined to accept a feature class data type. Refer to the above discussion. Note the “commented” `infile` line that shows a feature class name `'City_Facilities.shp'` which was likely used in the stand alone script before a custom ArcTool was created and associated with the Python file.

The next parameter in the **Clip and Buffer** tool is the input clip polygon feature class that will be used to clip the input feature class. Since this is the second parameter, it is referenced as `arcpy.GetParameterAsText(1)` which is assigned to the `clipfile` variable.

```
# clip file
# clipfile = 'Central_City_CommPlan.shp'

# clipfile is assigned the "second" parameter
# from the ArcGIS Tool

clipfile = arcpy.GetParameterAsText(1)
```

The rest of the parameters can be changed in a similar manner. The code developer should note the name of the variable in the script and the corresponding tool parameter value in the custom tool. The code developer may choose to re-order the variables so they correspond to a similar order in the ArcGIS toolbox. The code below shows the rest of the variables and their respective script parameter arguments.

```
outfile = arcpy.GetParameterAsText(2)

# output buffer feature class
# outbuffer = outpath + 'City_Facilities_Buffer.shp'

outbuffer = arcpy.GetParameterAsText(3)

# buffer distance
# buff_val = 100

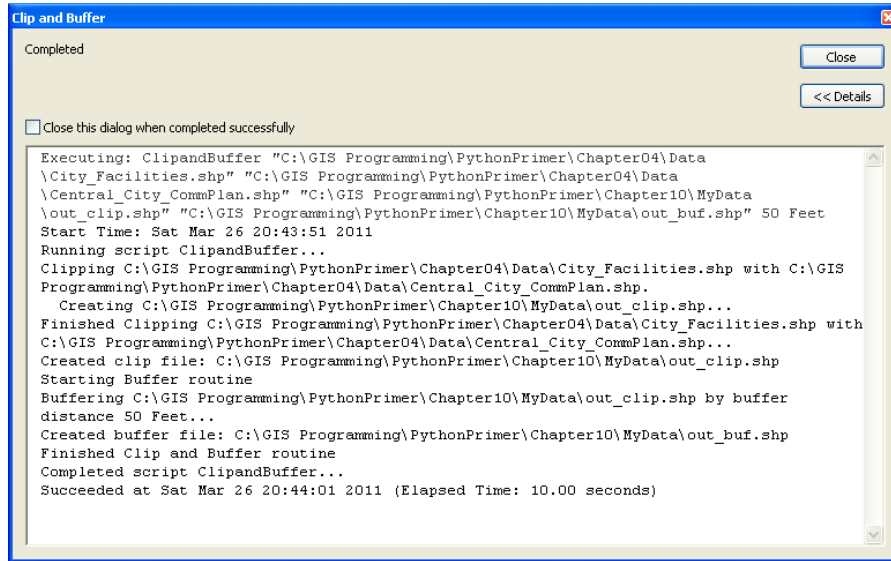
buff_val = arcpy.GetParameterAsText(4)

# buffer units
# buff_units = 'Feet'

buff_units= arcpy.GetParameterAsText(5)
```

Adding ArcGIS Messages to the Python Script

Before completing this section about creating a custom tool to run a script, it is helpful if some ArcGIS-type messages (a.k.a. the **print** statements) can be reported back to the tool's progress dialog box so the tool user can see its execution progress.



Up to this point, the stand alone scripts have used `print` statements to print messages only to the Python Shell.

ArcGIS messages can be easily added to a Python script by using the following syntax.

```
arcpy.AddMessage(' <text string>')
```

For the **Clip and Buffer** script tool, the following messages were added to the script. See the script below. The script has been formatted to fit the page. The **ClipandBuffer.py** file in the **Chapter10** folder contains the properly formatted code.

```
print 'Starting Clip routine'

arcpy.AddMessage('Clipping ' + infile + ' with ' + clipfile + '.\n
Creating ' + outfile + '...')

arcpy.Clip_analysis(infile, clipfile, outfile)

print 'Finished Clip routine'

arcpy.AddMessage('Finished Clipping ' + infile + ' with ' + clipfile
+ '...')

arcpy.AddMessage('Created clip file: ' + outfile)

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

print 'Starting Buffer routine'

arcpy.AddMessage('Starting Buffer routine')

arcpy.AddMessage('Buffering ' + outfile + ' by buffer distance ' +
buff_dist + '...')

arcpy.Buffer_analysis(outfile, outbuffer, buff_dist)

    print 'Created buffer file'

arcpy.AddMessage('Created buffer file: ' + outbuffer)

arcpy.AddMessage('Finished Clip and Buffer routine')
```

The above script shows both the `print` and the `AddMessage` statements. Only the `AddMessage` statements are processed and reported back to the process dialog box when the ArcGIS user uses the tool's graphical user interface. The `print` statements are provided as the programmer initially developed the code. These can be left in and used if the programmer needs to further develop the scripts before “re-deploying” the custom tool to ArcMap (or ArcCatalog). They do not need to be commented out.

The `AddMessage` function prints text statements to the Progress dialog box as processes are completed. The text statements can also include variable names and values reported from other routines such as the number of selected features or a count of objects.

In addition to the `AddMessage`, two other kinds of messages can be reported back to the Progress dialog box:

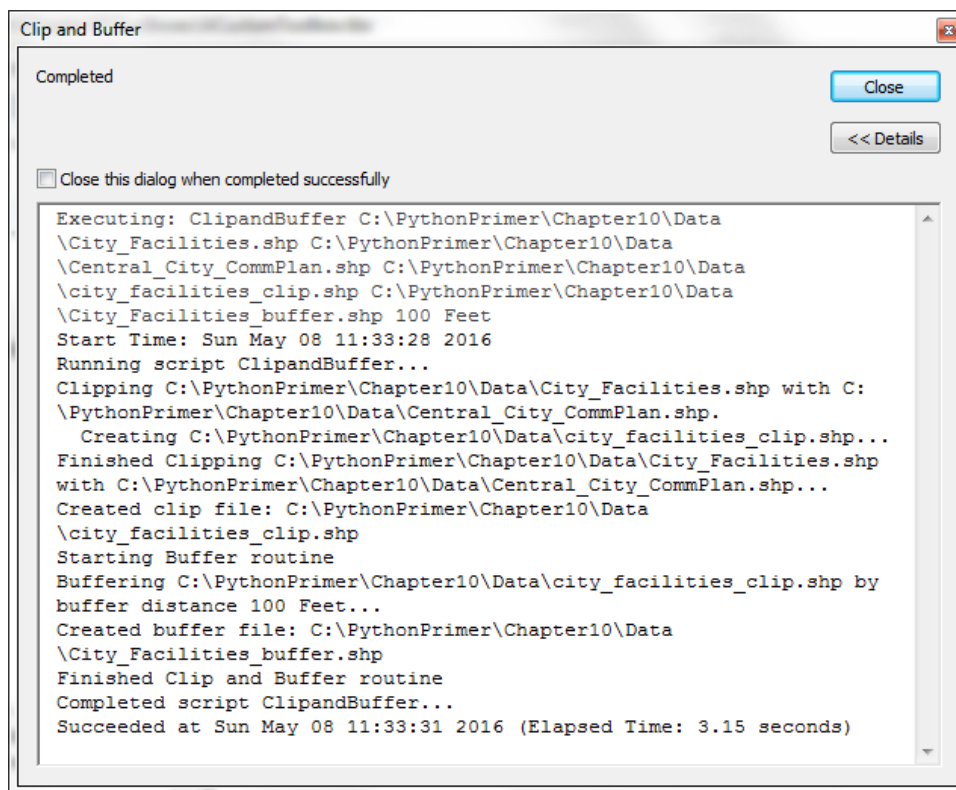
1. `AddWarning`
2. `AddError`

`AddWarning` represents a message associated with a potential problem with the process, but does not necessarily indicate that the process will fail. Examples of warnings may include a select layer routine that does not include any selected features or a feature class that does not include a defined spatial reference. `AddError` is typically associated with a process that will cause a script or tool to fail. Examples of errors might include a looping structure that loops beyond a valid set of values or the feature class cannot be found or does not include the correct type of field values (e.g. text, numbers, etc).

`AddWarning` and `AddError` messages can be added and customized by the programmer to provide meaningful feedback to the end user of the tool. The **Exception.py** script referenced in the book and in the example code uses the `AddError` message statement to report back a custom message when an ArcGIS error is encountered.

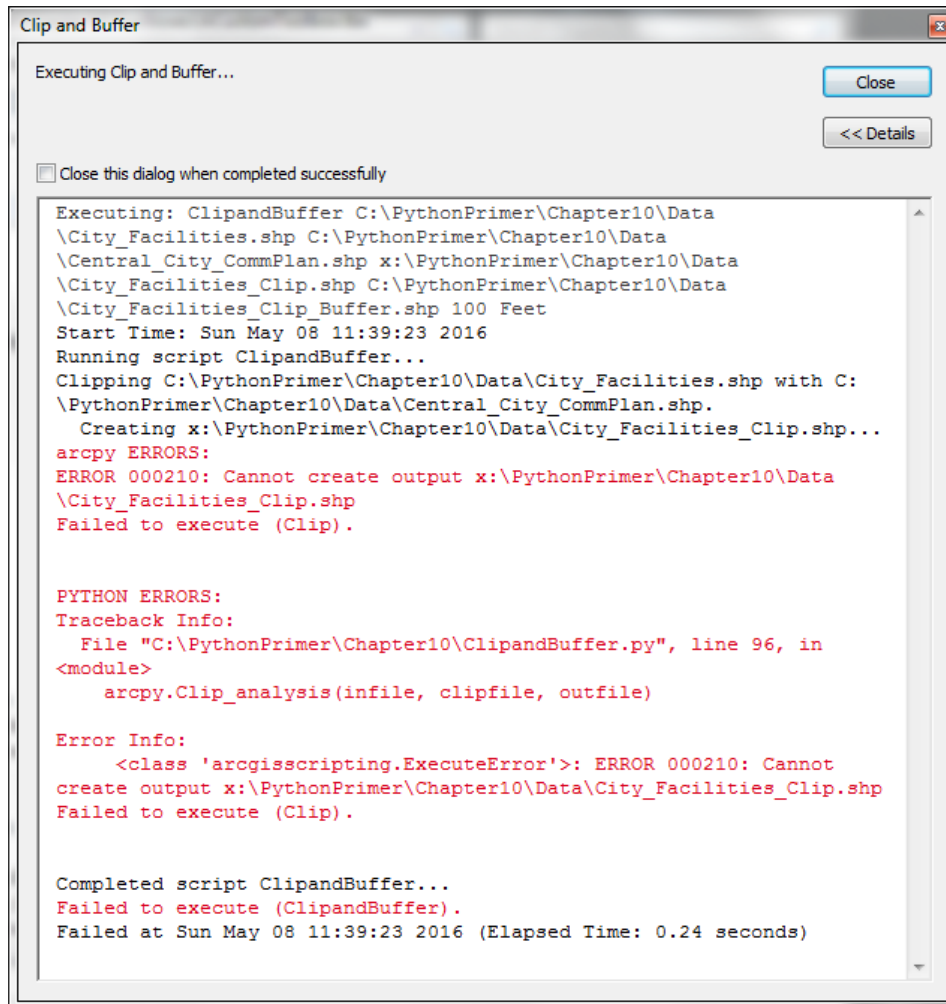
Executing the Script Tool

Once all of the parameters have been set and the Python script has been modified to include the `GetParameterAsText` and the `AddMessage` routines, the script tool created in the ArcToolbox can be executed. The user fills in the parameters and clicks OK. If the script is written correctly with the proper syntax for the parameters and the messages, a progress dialog box appears that reports the input values as well as any custom messages added to the script. The following dialog box shows the parameters and messages that are used and written for the **Clip and Buffer** tool referenced in this chapter.



Any errors encountered will also show up in the progress dialog box because of the `traceback` module and message statements found in the `except` block of the script. The reader should note the use of the `AddMessage` and `AddError` routines within the script.

The following figure shows an example of some error messages that were produced as a result of problems encountered when the **Clip and Buffer** routine was executed. Note the `except` block was implemented and specific error messages were “printed” back to the progress dialog box. The error messages are shown in “red” and the regular messages are shown in “black.”

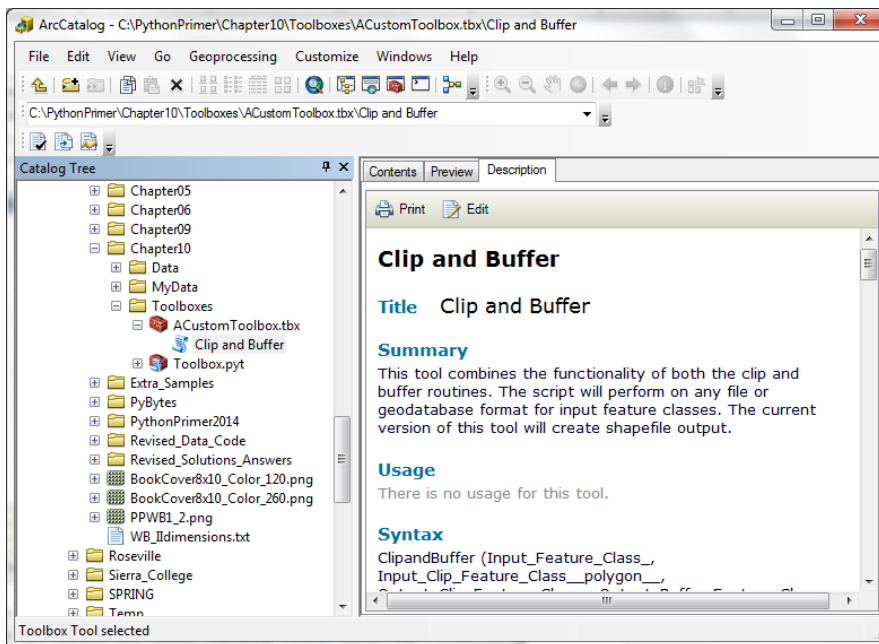


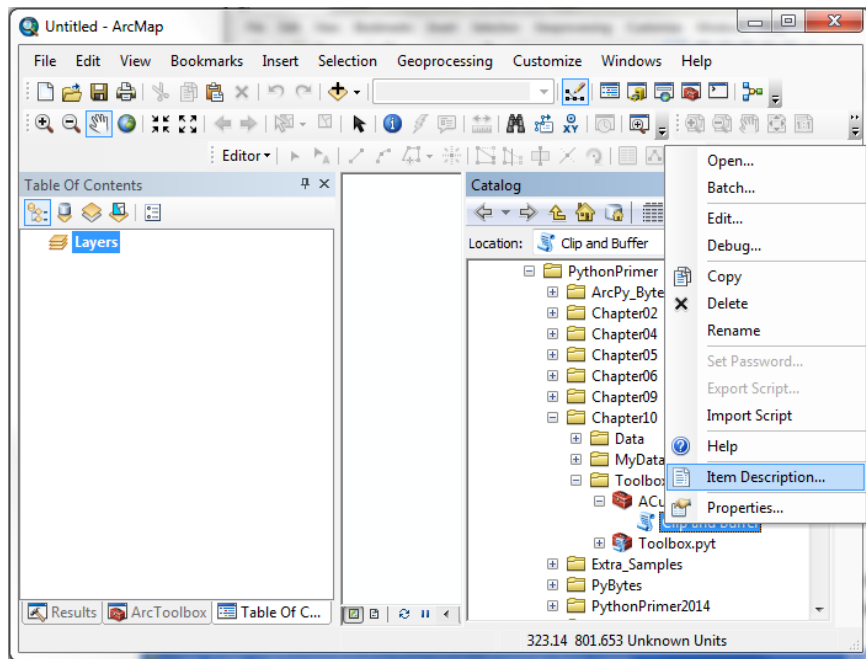
Writing Tool Documentation

Tool documentation is an important part of deploying a custom tool to an ArcGIS user. The documentation can provide some useful information and context that the end user can refer to for additional assistance when using the tool. Tool documentation can include brief descriptions of tool functionality, definitions and syntax of parameters, script examples and syntax, and illustrations. This section provides a brief introduction to develop tool help for a custom tool. The reader is encouraged to refer to the ArcGIS help documentation for more information **Geoprocessing—Creating tools—Documenting tools**.

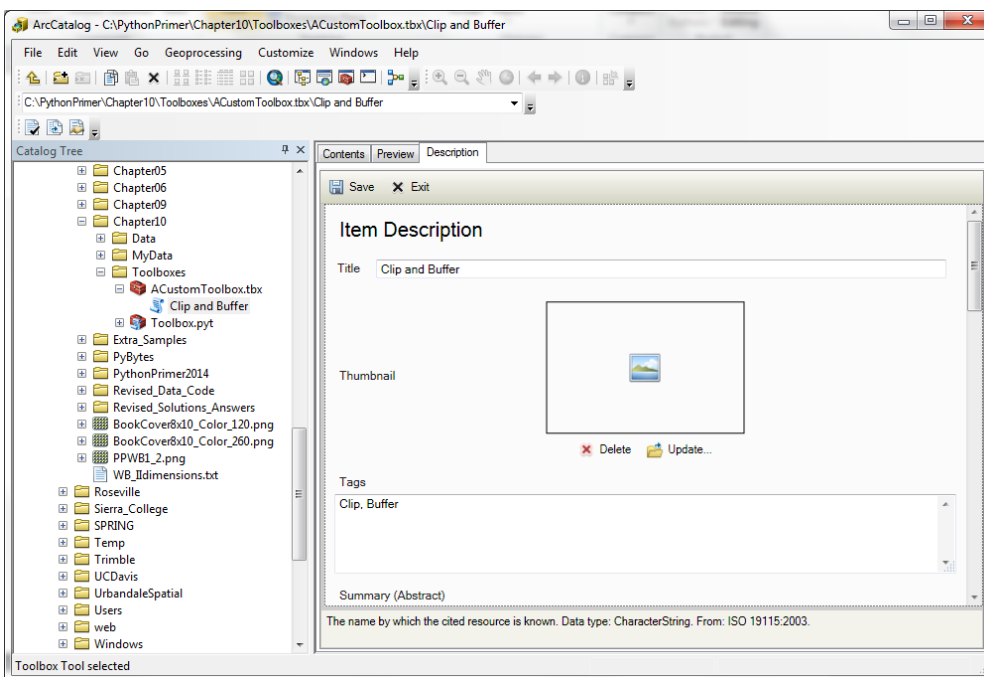
NOTE: It is highly recommended that any modifications to the custom tool help are performed within ArcCatalog or the Catalog Window (in ArcMap). Either of these locations will provide the ability to edit all aspects of the custom tool help documentation (such as summary descriptions, descriptions of the individual parameters, as well as adding image illustrations, etc.). **Warning: DO NOT access the Item Description through the ArcMap Toolbox.** **Not all features are available and ArcMap may lock up.** Any tools that have been developed with older versions of ArcGIS used different methods of editing tool help documentation. The existing documentation should appear, but some of the documentation may need to be rewritten or modified to work with ArcGIS 10+.

The code developer can create custom help documentation by locating the script tool in ArcCatalog and then click the **Description** tab or by right-clicking on the script tool in the Catalog Window and selecting **Item Description**. Both methods are shown below for the **Clip and Buffer** tool discussed in this chapter.





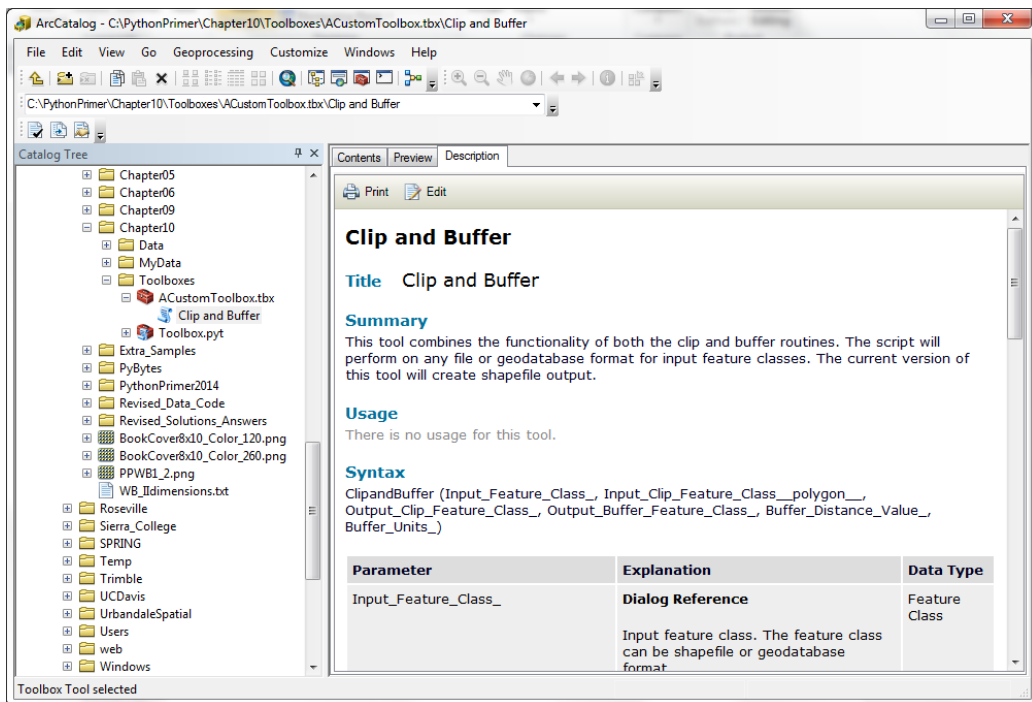
To edit the help document click on the **Edit** button at the top of the **Description** or **Item Description** tab. The following view appears showing a variety of tool help options that can be edited.



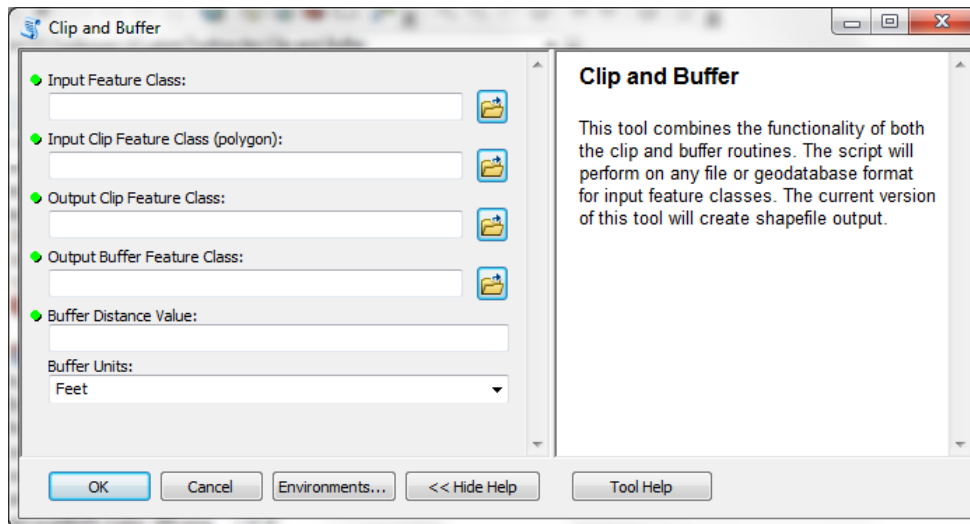
The title is already populated from the tool name provided when the tool was originally created. The help document has the following sections.

- Title* – title of the tool
- Tags* – comma separated keyword names that can be used by search tools
- Summary* – brief summary of the tool functionality
- Usage* – a brief overview of how the tool can be used. Refer to other tool help for ideas.
- Syntax* – specific help documentation for each parameter in the tool.
- Code Samples* – excerpts of code samples; similar to those found in other ArcGIS geoprocessing tool help documentation
- Credits* – Author of the script and/or tool
- Use Limitations* – brief description of any limitations of use for the script or tool

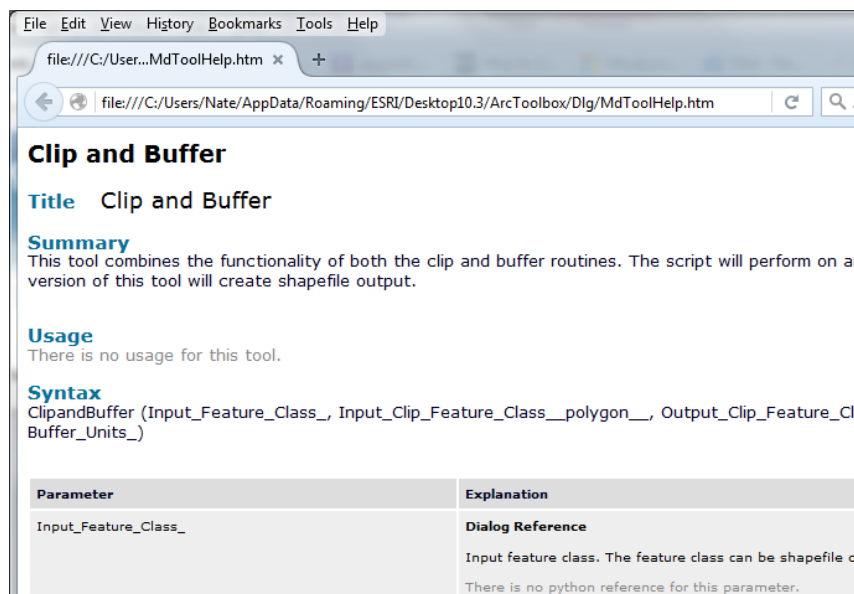
The documentation is easy to update by simply typing in descriptive information into each section. When the documentation is complete, it can be saved by clicking on the **Save** button. A preview can be reviewed in ArcCatalog or the Catalog Window in ArcMap.



When the custom script tool is opened the **Tool Help** shows the updated help information.



Click on the **Tool Help** button to show the completed help document for the script tool.



Note the location of the help document in the browser address field. The documentation is an HTML document that is written to the local system's ArcGIS toolbox under the specific user using the following path structure. Different operating systems may use a different path to store the tool help documents.

file:///C:/Users/<user>/AppData/Roaming/ESRI/Desktop<version>/ArcToolbox/Dlg/MdToolHelp.htm

The code developer can change this location if the tool and documentation need to be placed in a centralized location.

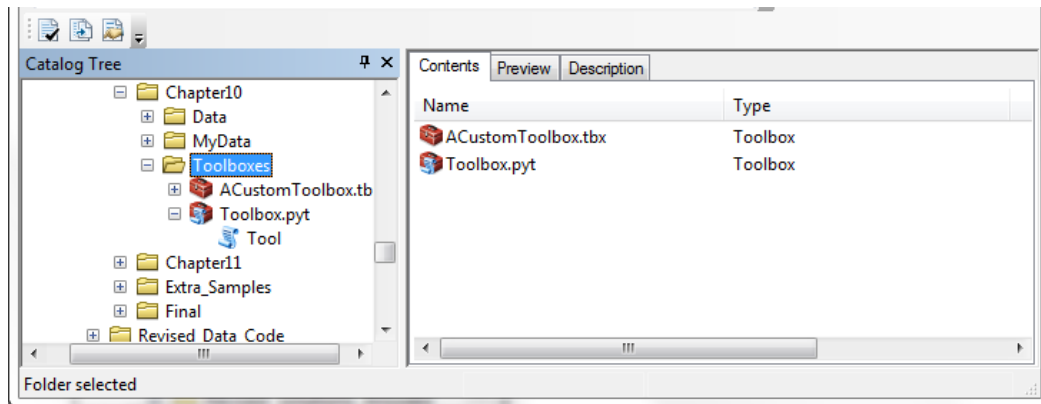
Python Toolboxes

As mentioned earlier in the chapter, Python Toolboxes differ slightly from Custom Toolboxes.

1. Custom toolboxes can contain models or script tools
2. The Python toolbox can only contain Python script tools
3. The code associated with the Python script tool is more tightly integrated with the toolbox

Creating a Python Toolbox

The Python Toolbox is created in the same manner as a Custom Toolbox in ArcCatalog. By default a Python Toolbox contains the extension `.pyt` and a default “Tool” script.



The `.pyt` file can be opened and edited by right clicking on the toolbox. The following is a sample of this file.

NOTE: The `.pyt` file tends to open in a default text editor (such as Notepad) when a user makes changes to a Python script tool. This is expected because Python IDLE doesn't recognize the `.pyt` extension by default. The author recommends making changes to the `.pyt` file in the default editor since changes are directly saved to the `.pyt` file. There is no need to save the `.pyt` to a different folder location.

```
import arcpy

class Toolbox(object):
    def __init__(self):
        """Define the toolbox (the name of the toolbox is the name of
        the.pyt file)."""

        self.label = "Toolbox"
        self.alias = ""

        # List of tool classes associated with this toolbox

        self.tools = [Tool]

class Tool(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""

        self.label = "Tool"
        self.description = ""
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definitions"""

        params = None
        return params

    def isLicensed(self):
        """Set whether tool is licensed to execute."""

        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before
        Internal validation is performed. This method is called
        whenever a parameter has been changed."""

        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for
        each tool parameter. This method is called after internal
        validation."""

        return

    def execute(self, parameters, messages):
        """The source code of the tool."""

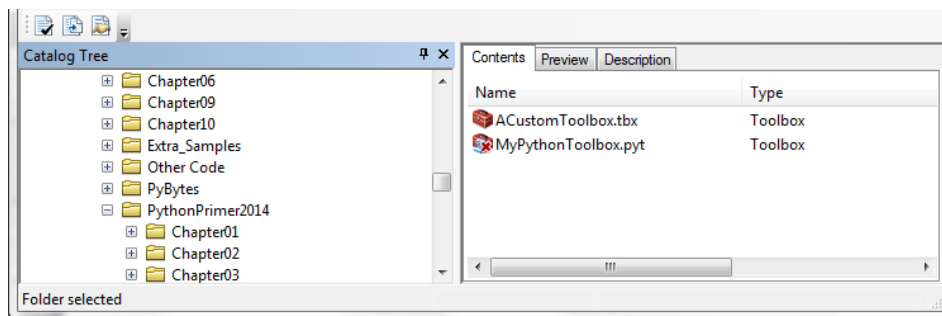
        return
```

The toolbox script (.pyt file) contains two “classes” of objects; one for the **toolbox** and one for the specific **tool** in the toolbox. A **class** is a programming construct that functions as a “template” for other objects. In the above script, the “toolbox” class contains a function that defines the toolbox name and alias. Likewise, the “tool” class provides a series of functions that help define tool and tool parameters, licensing, messaging, and tool execution. These “classes” serve as the “template” for any custom toolbox or tool that the programmer creates. All toolboxes will have names and all tools will have labels, descriptions, and have the ability to run in the background or foreground irrespective of the specific functionality of the toolboxes or tools. The .pyt script provides an easy way for a code developer to make a few changes in the template script so the Python toolbox and tools can function properly and keep this section separate from the Python script that provides the bulk of the primary geoprocessing functionality.

To customize the Python Toolbox once it has been created, the following can be performed.

1. Change the name of the Python toolbox (right click on the name in ArcCatalog and change the name)

NOTE: Make sure “NOT” to change the name of the “Toolbox” class in the Python code once the toolbox is named. If the class “Toolbox” is changed, the Python Toolbox will not be functional and the individual “tools” will not be recognized by ArcGIS. The illustration below shows an example where the Python toolbox contains a problem (note the red “x” on the toolbox icon).



2. Open the .pyt script (right click—Edit) and make changes to the toolbox label and/or alias
3. Change the default **Tool** class and label
4. Add one or more additional tool classes and provide unique names and labels
5. Add the unique tool names to the `self.tools` Python list in the **Toolbox** class
6. Add additional tool parameters to each tool (see below for more details)

Make sure to save the changes in the default text editor that appears (normally, Notepad in Windows). Just “Save” is required. It is not necessary to save the *.pyt* file to a different folder location.

If ArcCatalog is open by default, the toolbox and tools are refreshed automatically. One can also use the “F5” key or right-click on the folder or toolbox to refresh the toolbox.

The following script shows a *.pyt* file after some basic changes have been made. The bold lines indicate the changes from the original Python toolbox. The programmer added the Toolbox name (i.e. label), the Python list of unique tools in the `self.tools` list, and the two new tool classes.

```
import arcpy

class Toolbox(object):
    def __init__(self):
        self.label = "MyPythonToolbox"
        self.alias = ""

        # List of tool classes associated with this toolbox
        self.tools = [MyPythonTool1, MyPythonTool2]

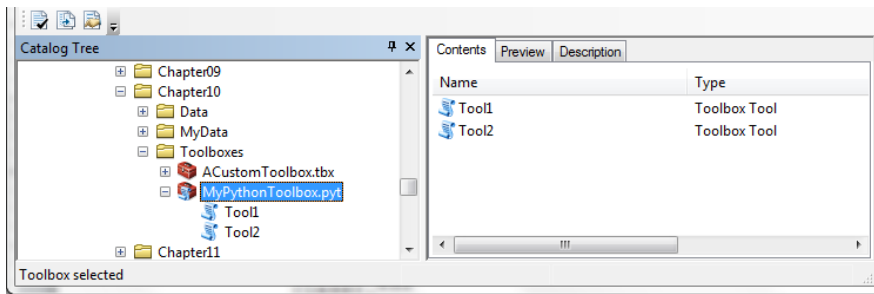
class MyPythonTool1(object):
    def __init__(self):
        """Define the tool (tool name is the name of the
class)."""
        self.label = "Tool1"
        self.description = ""
        self.canRunInBackground = False

    def execute(self, parameters, messages):
        """The source code of the tool."""
        return

class MyPythonTool2(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Tool2"
        self.description = ""
        self.canRunInBackground = False

    def execute(self, parameters, messages):
        """The source code of the tool."""
        return
```


The Python toolbox appears like this after the respective changes have been made.



Defining Parameters for the Python Script Tool

Two functions are required for a Python script tool to function within a Python Toolbox:

1. **Initialization function** – this is the `def __init__(self)` routine, a common Python function which provides some basic details of the tool, such as the label, the description, if the tool can run in the background or not, among others.
2. **Execution function** – this is the `def execute(self, parameters, messages)` routine. This routine implements the primary code to perform the geoprocessing functions. Where the Custom Toolbox required a specific Python (.py) file, the Python Toolbox contains all of the code required in the .pyt file to perform geoprocessing.

The other functions within the .pyt file are not required but most of them will likely be used. For example, when the user needs to interact with a tool interface, the `getParameterInfo(self)` routine is used to define and set up the proper data types. If a special license is required to process the tool (such as Spatial Analyst or 3D Analyst), the `isLicensed(self)` routine is used to check to see if the user has access to the appropriate license.

Defining parameters for a Python Toolbox script tool begins with writing code for the `getParameterInfo(self)` routine (function). Many parameter objects exist, but most will include those found in this code snippet.

```
def getParameterInfo(self):
    inFeature = arcpy.Parameter(
        displayName = "Input Feature Class",
        name = "in_features",
        datatype = "GPFeatureLayer",
        parameterType = "Required",
        direction = "Input")

    params = [inFeature]

    return params
```

def getParameterInfo(self): - the function that compiles all of the parameter information for individual inputs and outputs of a Python script tool. Each tool in a Python toolbox will have a *getParameterInfo* function.

arcpy.Parameter() – the specific ArcGIS routine that contains the specific parameter characteristics for a given parameter (e.g. an Input Feature) for a specific tool. A Python script tool can have one or more *arcpy.Parameter()* routines; one for each parameter in the tool.

inFeature – a variable that contains the list of defined parameter properties that is returned to the toolbox and used by the *execute* routine. The variable name is specified by the programmer.

displayName – the text string that is displayed in the tool user interface

name – the parameter name that is shown in the tool's syntax section in the tool help

datatype – one of many data types and depends on the parameter required for the geoprocess to function properly. Search the ArcGIS Help to access a full list of the various data types used in ArcGIS (see Data types for geoprocessing tool parameters).

parameterType – determines if the parameter is required, optional, or derived

direction – indicates if the parameter is an input or an output

params – a variable that contains the full list (Python list) of parameters for the tool

return params – syntax that returns the parameters to the Python script tool and to the *execute* routine to process the geoprocessing code.

Defining the Execute Function

The other required task to perform geoprocessing from a Python Toolbox is to write the code for the `execute` function. The values from the parameters in the `getParameterInfo` routine are passed to the `execute` function and used in geoprocessing tasks. Parameters in the `execute` routine use an index that begins with zero and correspond to the parameters in the tool. Like the custom ArcGIS tools, all parameters are interpreted as text strings (hence the use of `valueAsText`).

The code snippet below shows a simple example of the `execute` function. The function receives the input feature class from the `getParameterInfo` routine and then uses it to implement a `Describe` routine on the feature class to obtain the type of feature class. The function completes with a simple message printed to the tool's process dialog box.

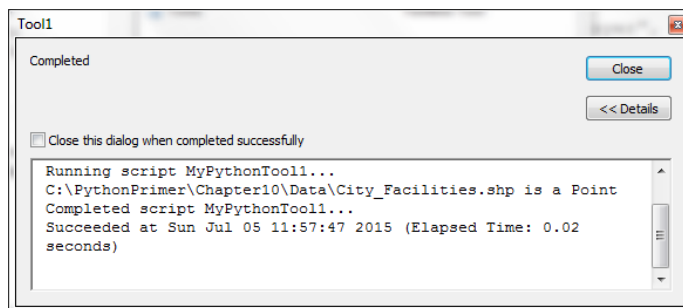
```
def execute(self, parameters, messages):
    """The source code of the tool."""

    # comes from the getParameterInfo function
    # parameters[0] is the first parameter in the tool
    inFC = parameters[0].valueAsText

    geomType = arcpy.Describe(inFC).shapeType

    if geomType == "Polygon":
        arcpy.AddMessage(inFC + " is a " + geomType)
    elif geomType == "Polyline":
        arcpy.AddMessage(inFC + " is a " + geomType)
    elif geomType == "Point":
        arcpy.AddMessage(inFC + " is a " + geomType)
    else:
        arcpy.AddMessage(inFC + " is a different geometry type")

    return
```



Normally, the `execute` function will be more complicated than the above example. See **Demo 10b** for a more realistic Python toolbox script tool.

Summary

This chapter focused on creating and using Python functions for custom ArcGIS Tools as well as review the two primary methods to create custom tools and assigning routines to them:

1. Custom ArcGIS Toolbox
2. Custom Python Script Toolbox

Having a useful user interface for an end user can be helpful to deliver enhanced functionality beyond the existing geoprocessing tools within ArcToolbox. Using the methods in this chapter can help achieve this goal. In addition to writing and modifying the Python script for the custom tools, providing some useful “help” documentation is also a good idea and required if one expects users to “use” and “understand” the functionality of the custom tool.

To recap, the key elements to create a functioning tool interface are:

- a. Determine the required and optional parameters
- b. Set the correct data types, value lists, and default values
- c. Set parameters as input or output
- d. Set any dependencies between parameters
- e. Modifying code within the Python script or the Python tool script (.pyt) as needed
- f. Create any informative, warning, or error messages
- g. Create useful help documentation

Developing each of these elements can be challenging above the challenges of writing the actual Python script. It is highly recommended that a Python script is thoroughly tested, error handling is added, and various conditional statements are added to check for any specific issues that might occur (e.g. does a spatial reference exist, is a feature class a polygon type, etc.) before deploying a script tool. Even after these kinds of quality checks are implemented other issues may be discovered when designing and testing a tool graphical user interface (GUI) with the Python script. **Demo 10a, 10b**, and the **Exercise** provide the reader a good starting point to learn and experiment with the concepts in this chapter.

Chapter 10 Demos

The Chapter 10 demos focus on creating and configuring custom toolboxes that use Python scripts. **Demo 10a** walks through the process of creating a custom ArcToolbox that uses an associated Python script. **Demo 10b** works through a similar process of creating a custom “Python” toolbox where the programmer writes code within a Python script tool (.pyt) to provide the configuration and custom geoprocessing capabilities. **Demo 10b** introduces the programmer with coding functions and using “classes.” After working through the demos, the reader should have a good understanding of the two key methods for relating a Python script to a custom ArcGIS Toolbox. The author recommends making a copy of the Toolbox folder to a different folder before working through the demos. The toolboxes in the copy can be referred to as needed.

The concepts illustrated in the chapter and demos are:

ArcGIS Concepts

- Custom ArcToolboxes
- Custom Python Toolboxes
- Spatial Analyst Extension
- Tasseled Cap image processing algorithm
- Rasters
 - GetParameterAsText
 - getParameterInfo
 - valueAsText
- AddMessage
- AddError

Python Concepts

- class
- def (Python functions)
- Python Lists

Demo 10a: Custom Script Tool - Clip and Buffer Tool

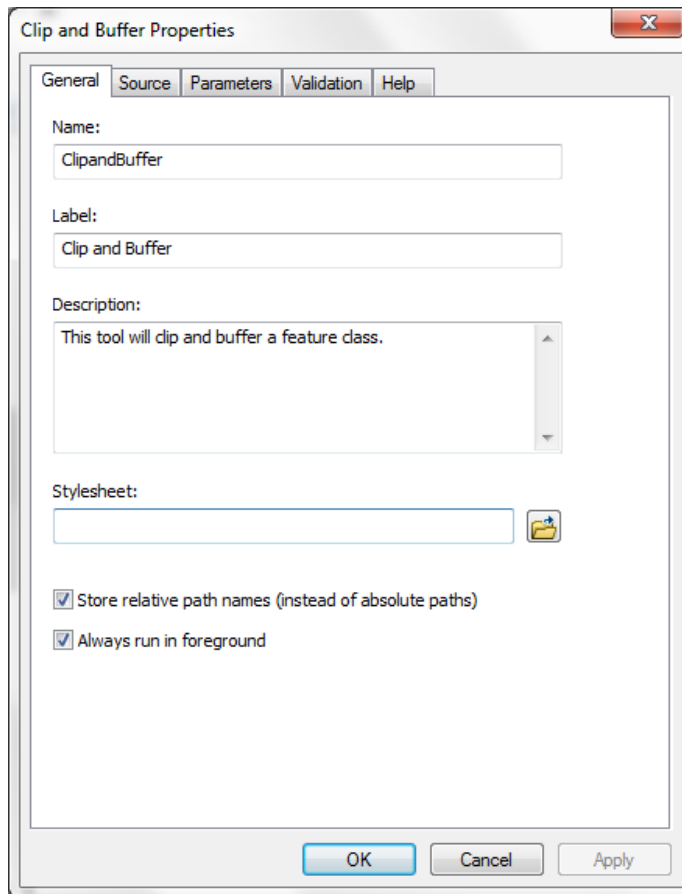
This demonstration uses the **ClipandBuffer.py** script that has already been developed and discussed in the chapter. The steps below can be followed to develop the graphical user interface (GUI) and parameter properties for the **Clip and Buffer** tool described in the chapter. Refer to the **Creating a Custom Toolbox** and subsequent sections and the **Parameter Table** found in Chapter 10 under the **Parameter Properties** section to construct and define the parameters and properties for the tool interface. The reader can refer to the chapter to study and practice developing a custom script tool and the associated changes to the script as well as develop the help documentation. The script, toolbox (**ACustomToolbox.tbx**), data, and help document (**Dlg.zip**) can be found under **\PythonPrimer\Chapter10**.

The demo script (**ClipandBuffer.py**) contains the `GetParameterAsText` and `AddMessage` routines to pass values from the tool interface to the Python script and to write messages back to the progress dialog box. Consult the script for more details.

The **Dlg.zip** file contains the help documentation created by the author for this tool. The reader is encouraged to generate their own tool help by using the methods described in this chapter to gain additional experience. The help document in the zip file can be referred to or viewed, if desired.

STEP 1 - Create the Custom Toolbox

1. Before developing the tool interface, make sure to create a custom toolbox. Use the directions in the **Creating a Custom Toolbox** section of Chapter 10 to create the toolbox and then right-click on the new toolbox and choose **Add--Script** to bring up the default dialog box for the tool properties.
2. Assign the tool a name **Clip and Buffer**. Make sure not to use spaces in the *Name* property. The *Label* can contain spaces. Provide a brief description if desired.



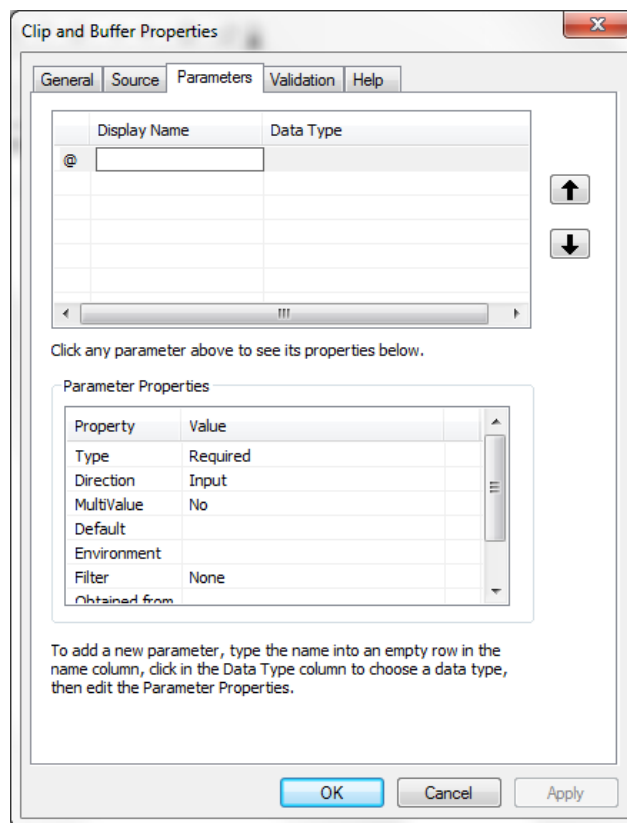
STEP 2 - Assign the Python Script to the Tool

Click on the **Source** tab (or click the **Next** button, if this is the first time to create the tool) and browse to the folder location of the **ClipandBuffer.py** script.

STEP 3 - Customize the Script Tool Interface Parameters

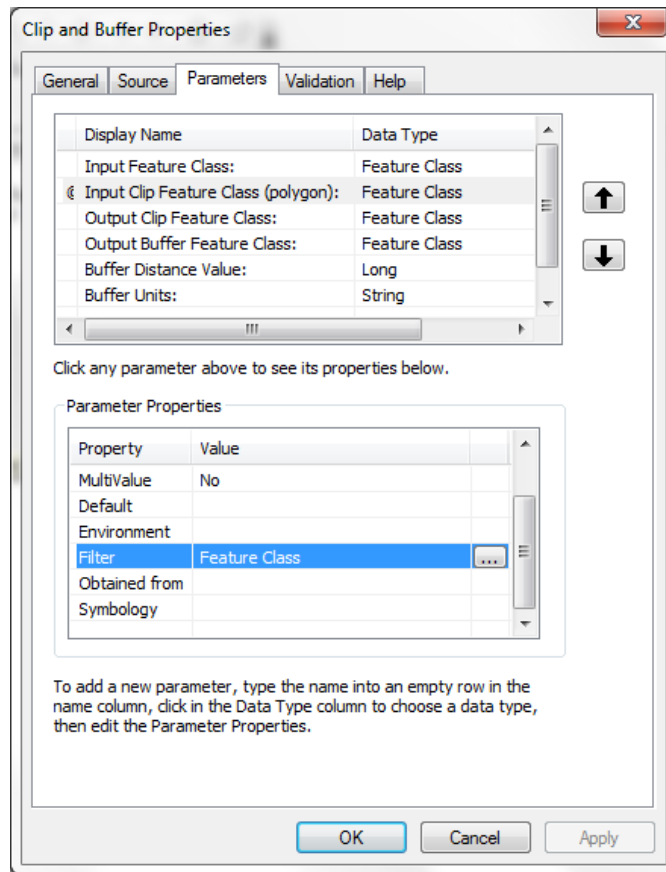
To create the script tool interface, the *Display Name*, *Data Type*, and *Parameter Properties* must be set. This section illustrates the steps to set up the **Clip and Buffer** script tool interface parameters. Refer to the values in the above **Parameter Table** in the **Parameter Properties** section of the chapter to create the tool interface.

1. Bring up the script tool Properties (right click and choose **Properties**) and click on the **Parameters** Tab. If this is the first time to create the toolbox and associate the script, the **Next** button is used to access the Parameters tab. If the code developer is modifying a script tool that is already associated with a custom toolbox, but does not contain any parameters, right click on the script tool and choose Properties and then click on the Parameters Tab.



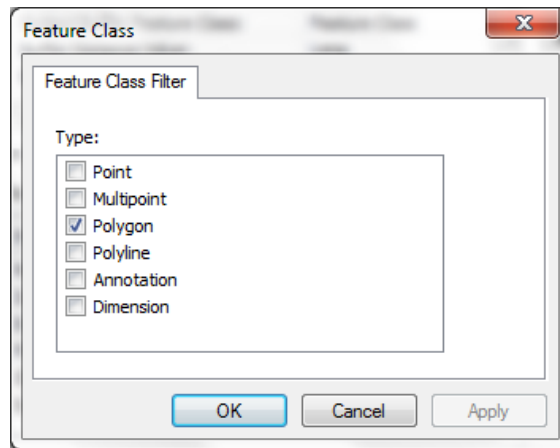
2. In the **Display Name** field enter the first Display Name from the information found in the **Parameter Table** in Chapter 10 (e.g. **Input Feature Class**). The "@" symbol indicates which Parameter is being created or modified.

3. Click in the **Data Type** and choose *Feature Class* from the list. Keep the default values for all of the other Parameter Properties for this parameter.
4. When the code developer enters the **Input Clip Feature Class** parameter, the *Filter* Parameter Property is used. Enter the Display Name and Data Type as describe above. Next, in the *Filter* Parameter Property select *Feature Class* from the **Value** column.

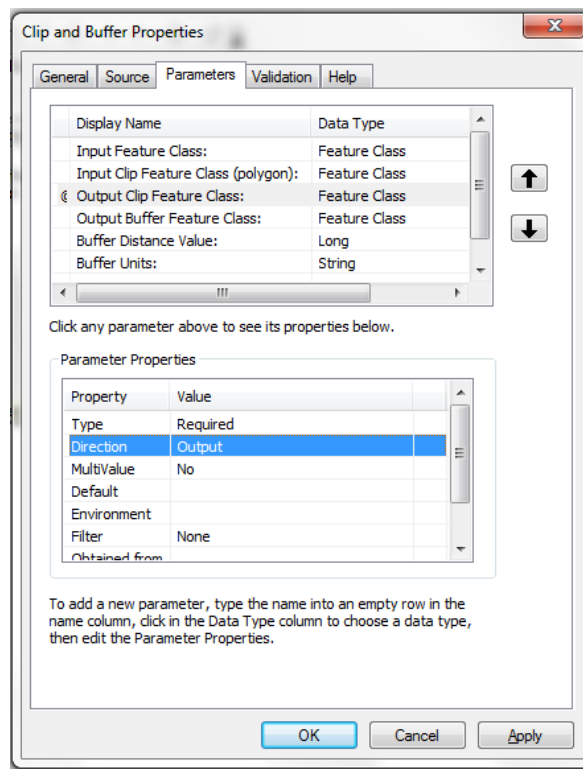


When the Feature Class value is clicked and selected from the list, the following dialog box appears. Check only the *Polygon* check box since clip feature must be a polygon feature type.

NOTE: If the user needs to modify this value again, the Feature Class value can be clicked in the Value column or the "..." (ellipse) can be clicked to bring up this dialog box.



5. Other parameters can be added using the **Parameter Table** in Chapter 10. For both the **Output Clip Feature Class** and **Output Buffer Feature Class** change the **Direction** to *Output*, since these feature classes will be saved as outputs from the tool.

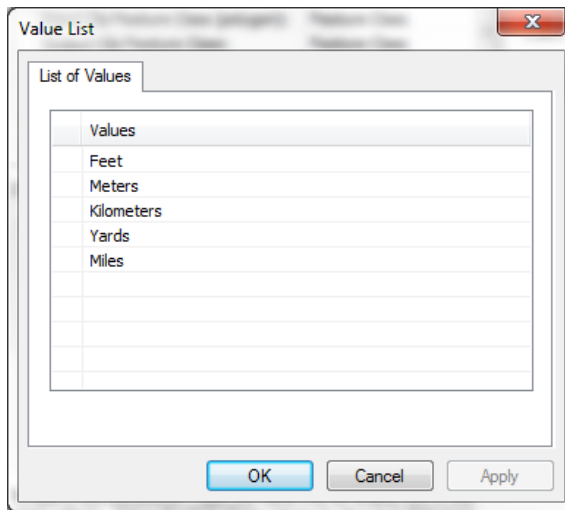


6. The **Buffer Distance Value** should be entered as a **long**, since the tool should only accept number values for this parameter. This will prevent the tool from accepting text characters. The **Buffer Units** parameter will be entered as a **String** data type and use a

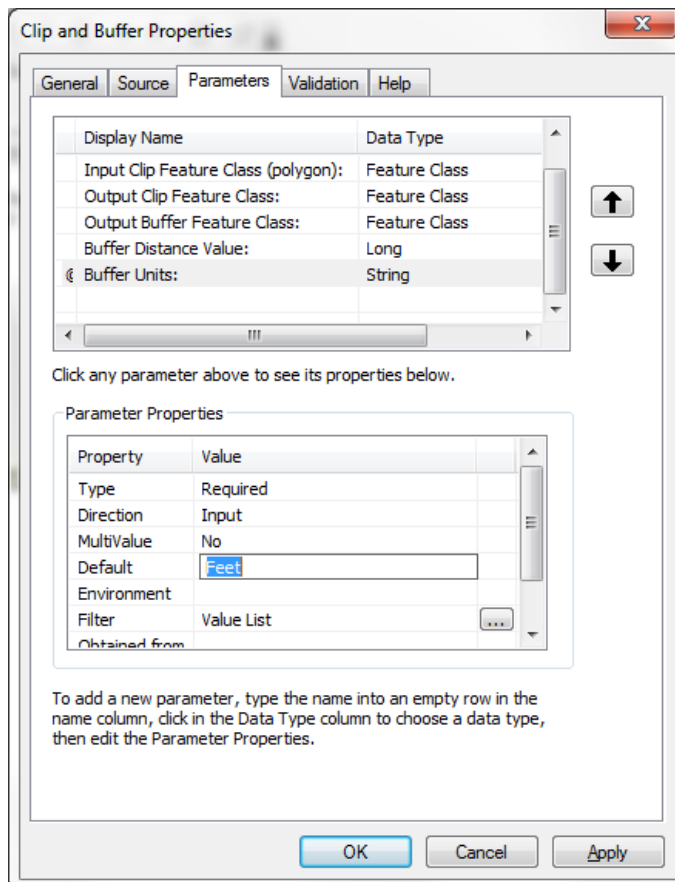
Value List for the **Filter**. When the code developer clicks on **Filter—Value List** a dialog box pops up so that the specific values for the list can be entered. The specific values for the unit types entered are:

Feet
Meters
Kilometers
Yards
Miles

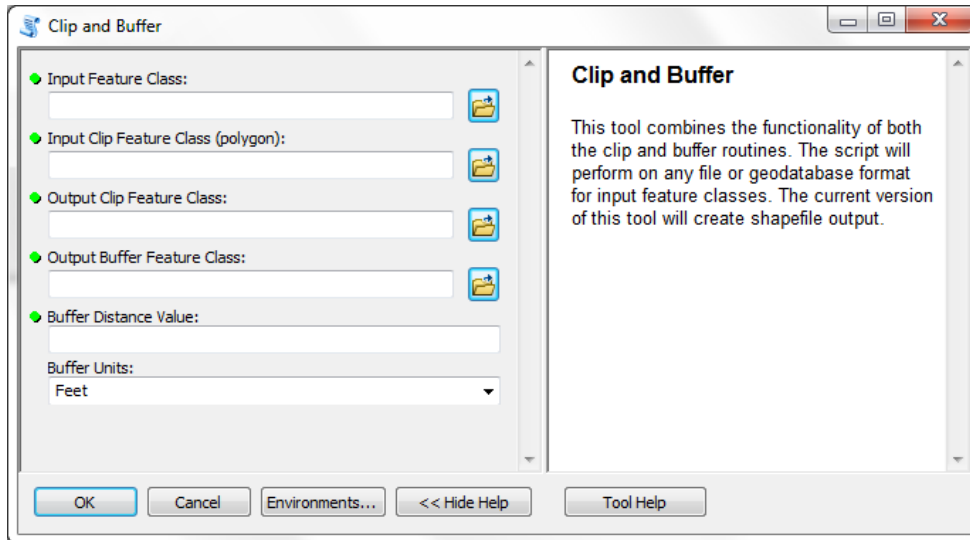
The resulting value list will look like this.



After clicking OK, the code developer can set the **Default** Parameter Property to **'Feet'**. This will provide **'Feet'** as the default value in the tool interface.



7. Now that all of the parameters have been set up, the code developer can click OK. The **Clip and Buffer** script tool can be opened by double clicking on the tool to see the new tool interface.



Notice each of the parameter's headings and see that all of the parameters are required (i.e. the "green dot" appears next to each parameter), and that the **Buffer Units** parameter already has the default value loaded. A dropdown list exists for the user to choose a different units option.

The reader can review the tool help developed for the **Clip and Buffer** tool and can experiment with editing the documentation. Refer to Chapter 10 for an overview of how to develop and modify the custom help documentation.

STEP 4 – Test the Clip and Buffer Tool

Use the following data in the **Chapter10\Data** folder to test the tool. The results should look similar to those in the **Chapter10\MyData** folder. If they do not, review the demo and the material in Chapter 10.

1. **Input Features** – *City_Facilities.shp*
2. **Input Clip Feature Class** – *Central_City_CommPlan.shp*
3. **Output Clip Feature Class** – user entered folder location and feature class
4. **Output Buffer Feature Class** - user entered folder location and feature class
5. **Buffer Distance** – user specified
6. **Buffer Units** – user specified from dropdown list

Demo 10b: Tasseled Cap using a Python Toolbox

Tasseled cap is a common image processing routine used with Landsat TM satellite imagery to derive greenness, wetness, and brightness image bands. This demo illustrates the use of the Python Toolbox to provide an interface for the user of the tool as well as implement the script to perform the tasseled cap routine using the Spatial Analyst extension. If the reader does not have a Spatial Analyst extension, the demo cannot run; however, the script and process can be reviewed.

STEP 1 – Create the Python Toolbox

In **ArcCatalog** within the **Chapter10\Toolboxes** folder right click and choose **New—Python Toolbox**.

Rename the toolbox to **Image Processing**.

STEP 2 – Change the default Tool name

On the **Image Processing.pyt** toolbox, right click and choose **Edit** to edit the Python template.

Note that the default text editor of the operating system will open versus Python IDLE or other editor. This is normal and the script will not show up with color coded text. The author recommends not attempting to open *.pyt* files directly in Python IDLE or other script editor. References to make such modifications can be found on the Internet and the reader does so at their own risk.

Make the following changes in the script (shown in bold below).

```
class Toolbox(object):
    def __init__(self):
        self.label = "Image Processing"
        self.alias = ""

        self.tools = [TasseledCap]

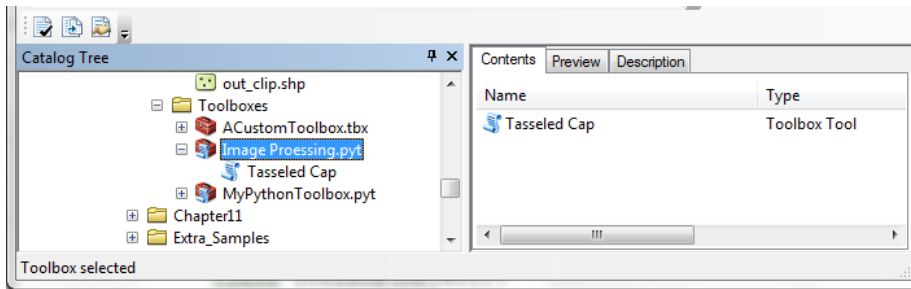
class TasseledCap(object):
    def __init__(self):

        self.label = "Tasseled Cap"
        self.description = ""
        self.canRunInBackground = False
```

Click **Save** (not Save As). **Save** saves the changes made above to the existing **Image Processing.pyt** file (i.e. the Python Toolbox).

Refresh the toolbox within ArcCatalog by *right clicking View—Refresh* on the toolbox or use the “F5” key.

The **Image Processing** toolbox should look like this:



STEP 3 – Code the Tool Parameters

The Tasseled Cap routine requires an input image file and an output image file. These two parameters are added to the `getParameterInfo` function.

Right click on the **Image Processing.pyt** file and select **Edit**.

In the `getParameterInfo` routine add the following code. NOTE: `DERasterDataset` is the data type for an image file. The parameters (`params`) are “returned” so they can be used in other functions within the tool (such as to check for licensing or for parameter validation).

```
def getParameterInfo(self):

    inRaster = arcpy.Parameter(
        displayName = "Input Landsat Image",
        name = "in_raster",
        datatype = "DERasterDataset",
        parameterType = "Required",
        direction = "Input")

    outRaster = arcpy.Parameter(
        displayName = "Output Tasseled Cap Image",
        name = "out_raster",
        datatype = "DERasterDataset",
        parameterType = "Required",
        direction = "Output")

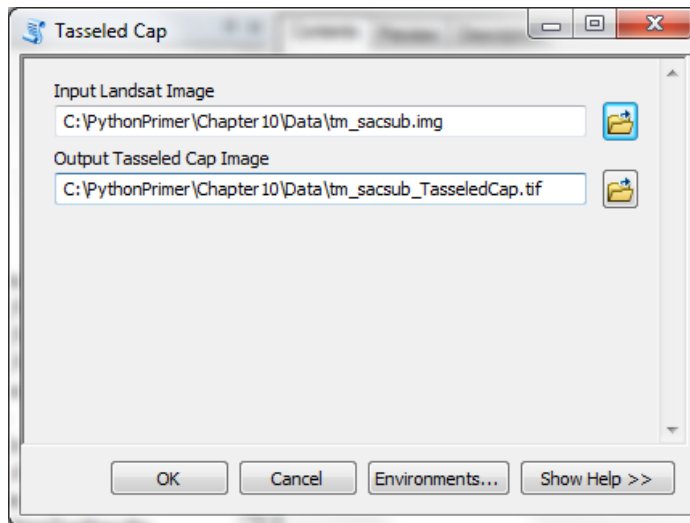
    params = [inRaster, outRaster]
    return params
```

Save the changes to the **Image Processing.pyt** file and then refresh the **Image Processing** toolbox.

STEP 4 – Check the Tool Interface

Open the **Tasseled Cap** tool in ArcCatalog. Browse for the **tm_sacsub.img** for the *Input Landsat Image* in **Chapter10\Data**. Notice that the output image file is auto generated in the input folder (i.e. **Data**) and contains the name of the input image and the name of the tool as a default file name. The user can change this, if desired (such as using **Chapter10\MyData**).

Do not run the tool at this point. This is just a check to make sure the code written above is written correctly. Sometimes the tool will show a red “x” if Python syntax is written incorrectly.



STEP 5 – Validate Parameter Values and Licensing

Depending on the kind of operation, specific parameters should be checked to make sure they are valid before the tool is implemented. In the same manner extension licensing can be checked to make sure the appropriate ArcGIS extension exists and is available for use.

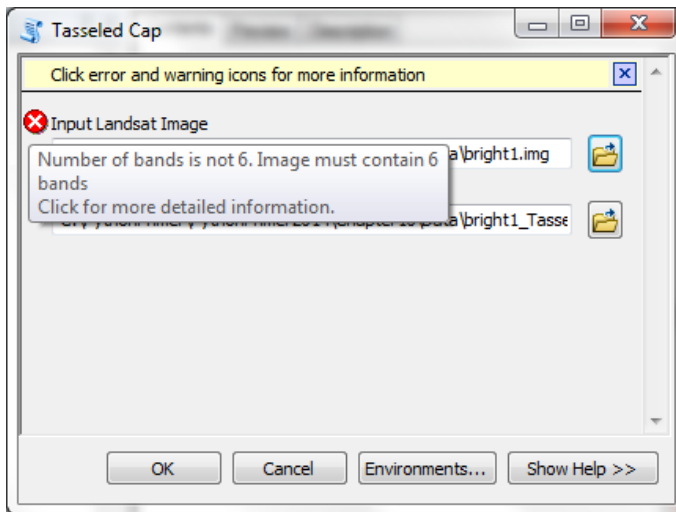
Add the following code to the `isLicensed` and `updateMessages` functions. The `isLicensed` routine checks to see if the Spatial Analyst extension is available; the `updateMessages` routine checks to see if the input image (`parameter[0]`) contains six bands (since the tasseled cap routine requires a six band image, in this example). If the input image file contained information about the sensor (e.g. Landsat TM), an additional check could be added to check the input image is from the Landsat sensor. The tasseled cap routine contains specific factors that relate to specific sensor types, so specific kinds of images can only be used to perform the tasseled cap routine.

```
def isLicensed(self):  
    try:  
        if arcpy.CheckExtension("Spatial") != "Available":  
            raise Exception  
    except Exception:  
        arcpy.AddError("Spatial Analyst Extension not available")  
        return False  
  
    return True  
  
def updateMessages(self, parameters):  
    if parameter[0].altered:  
        numbands =  
            arcpy.Describe(parameter[0].valueAsText).bandCount  
  
        if numbands <> 6:  
            parameters[0].setErrorMessage("Number of bands is not  
            6. Image must contain 6 bands")  
  
    return
```

STEP 6 – Test the Changes

The tool can be tested to see the effect of these changes.

Use the **Chapter10\Data\bright1.img** file as the input. This is a single band image. When this file is added as an input, the Tasseled Cap tool should show a red “x” icon. When the user hovers over the icon, the message should show the message added in the `updateMessages` routine. If the user does not have access to Spatial Analyst, the Spatial Analyst check will likely appear during the actual tasseled cap processing tool indicating that the Spatial Analyst extension is required.



Click the **Cancel** button to close the dialog box.

STEP 7 – Add the Execution Code

Copy the execute function code from the **TasseledCap_Execution_Function.py** file in the **Chapter 10** folder into the execution function for the **Image Processing** tool. Make sure the `def execute()` lines follow the same indentation as the rest of the code in the tool, otherwise, the script may not execute correctly.

In addition, modify the top portion of the **Image Processing.pyt** script with the following changes. The `os` module is required for some of the intermediate files that are created during the process. The `from arcpy.sa import *` imports the required Spatial Analyst functionality so that the raster processing can execute.

```
import arcpy, os
from arcpy.sa import *
```

The *execute* routine for the tasseled cap process sets several variables before the actual tasseled cap algorithm is performed.

1. Set the overwrite output environment variable to True

```
arcpy.env.overwriteOutput = True
```

This allows the output to be automatically overwritten.

2. Set an output path for the intermediate image files (bright, green, and wet)

```
outpath, img = os.path.split(parameters[1].valueAsText)
```

The Python syntax shows two different variables being assigned: one for the path (*outpath*) and one for the image file name (*img*).

3. Check out a Spatial Analyst License

```
arcpy.CheckOutExtension("spatial")
```

In addition to checking to see if a license is available for Spatial Analyst, the extension must be “checked out” to use the functionality within Spatial Analyst. After the image processing has occurred, the “check in” routine is used to check the extension back in so other users can use Spatial Analyst.

4. A numbers of variables are set to point to the output of the individual components of the tasseled cap routine (bright, green, and wet images).

```
brightbnd = os.path.join(outpath, 'bright1.img')  
greenbnd = os.path.join(outpath, 'green1.img')  
wetbnd = os.path.join(outpath, 'wet1.img')
```

5. Variables are set to point to the input image and set up the final tasseled cap output

```
in_image = parameters[0].valueAsText + "\\
```

The “\\” is added so that the proper data structure can be set up to access a specific image band from the input image. The output image does not require the “\\”.

6. Set a variable for the output tasseled cap image.

```
tasseled_cap = parameters[1].valueAsText
```

Once all of the changes are made, save the **Image Processing.pyt** file and close the text editor.

STEP 8 – Implement the Tasseled Cap Image Processing Routine

1. In ArcCatalog double click on the Tassel Cap tool to open the tool interface.
2. Fill in the following parameters.

Input Landsat Image: Browse to the location of the **tm_sacsub.img** in the **Chapter 10\Data** folder

Output Tasseled Cap Image: Browse to a folder (i.e. **Chapter10\MyData**) and set an Output file name (e.g. **tasseled_cap.img**).

3. Click OK. The input image takes only a few seconds to process.

The output tasseled cap image should look similar to the image below.



The resultant image is often “not” viewed, but is typically an input to other image processing routines. The individual components (bright, green, and wet) can also be viewed individually and may provide a more intuitive perspective on the output. Individual image bands will appear in gray scale. For example, “greener” objects in the natural world will appear as brighter pixels in the greenness image (**green1.img**), areas with higher water content (standing or water contained within vegetation) will appear as brighter pixels in the wet image (**wet1.img**), and bright or highly reflective materials (buildings, dry bare soil, etc) will appear as brighter (more white) pixels in the bright image (**bright1.img**). The user can review the individual tasseled cap images that compose the resultant tasseled cap image (**tasseled_cap.img**).

Optionally, the user can add help documentation, if desired.

Exercise 10 - Custom Script Tool – Develop Your Own Tool

Exercise 10 is an open exercise where the reader is free to develop any kind of script, set up a tool (preferably a Python Script Tool) graphical user interface (GUI), and write the documentation for the tool. The Chapter 7 script for batch clipping images can be a good starting point with a script that already functions as a “stand alone” script. At the completion of this exercise, the reader should have a custom tool interface that successfully mimics the same functionality as the “stand alone” script that is run through the Python IDLE editor.

Chapter 10 Questions

1. What are some of the benefits of using a “function” to write Python code?
2. What key elements are required to successfully create and implement a custom script tool?
3. What are the major differences between a Custom ArcTool and a Python Script Tool?
4. Can a custom tool without any user input parameters be used to process a Python script? Why or why not?
5. Why are tool parameters useful in a custom script (Python or non-Python) tool?
6. What elements are required to create a specific tool parameter?
7. Briefly describe what Parameter Properties are? How are they used to modify or limit a specific tool parameter?
8. Briefly describe the following and how they are used with a tool interface.
 - a. `GetParameterAsText()`
 - b. `AddMessage()`
 - c. `AddWarning()`
 - d. `AddError()`
9. What are the two locations in ArcGIS where a code developer can create and edit help documentation?
10. When the tool help is updated, how does a tool user access or see the tool help?

Chapter 11 Python Add-ins

The Python Add-in, introduced in ArcGIS 10.1, is a framework to provide custom tools and buttons on the ArcGIS interface. Only a limited number of options are available with a Python Add-in, so do not expect to develop an involved GUI similar to what used to be available with Visual Basic for Applications (VBA). If the user needs to run a process after the user interacts with map interface or input to a custom toolbar tool, menu, or drop down list, then creating a Python Add-in is a good option. The following interfaces can be created and customized as part of a Python Add-in.

- Buttons
- Tools
- Combo Boxes
- Menus
- Toolbars
- Tool Palettes
- Application Extensions

Users will not see a full custom Graphical User Interface (GUI) with a number of different objects in it, but rather a custom toolbar with one or more interfaces from the above list where each interface can have custom functionality.

Python Add-in Prerequisites

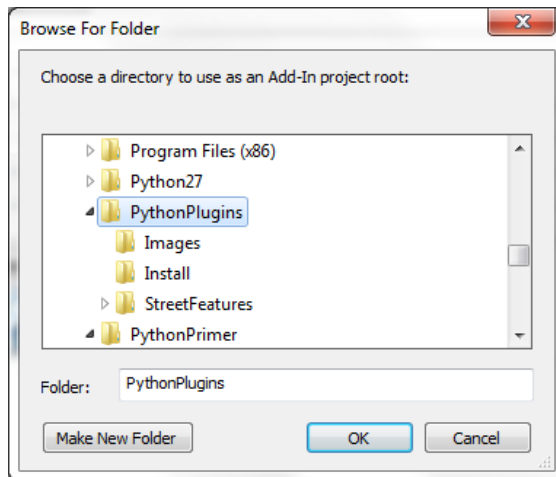
Before working with and customizing Python Add-in functionality, the Python Add-in Wizard must be downloaded from ArcGIS Online (www.arcgis.com) or perform a web search for “ArcGIS and Python Add-in Wizard.” The **addin_assistant.zip** file is downloaded. The user extracts the contents to a location (user specified) on the local computer. ESRI recommends the user create a desktop shortcut to the **addin_assistant.exe** file so it can be easily found and used when developing Python Add-ins.

The following steps can be used to build a Python Add-in.

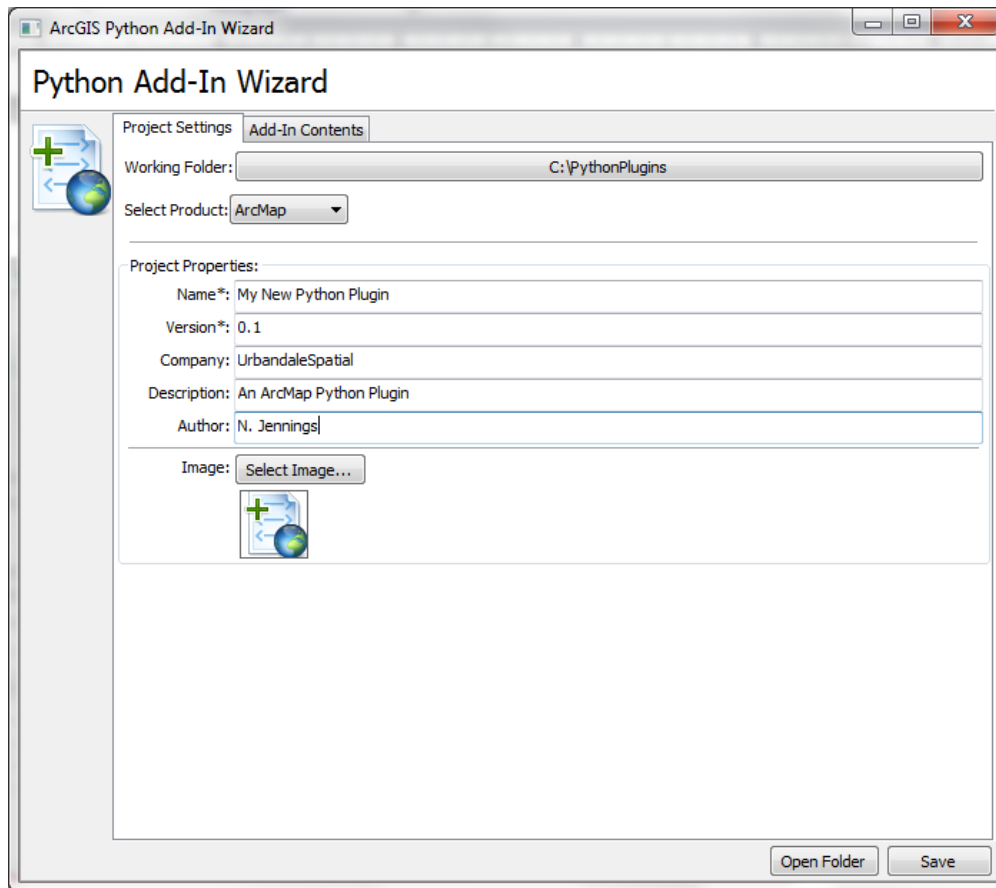
1. Create a Python Add-in Project
2. Create a Custom Toolbar
3. Add an Interface to the Toolbar
4. Add Functionality to the Python Add-in
5. Install the Python Add-in
6. Use the Python Add-in

Creating a Python Add-in Project

The first step to create a Python Add-in is to create a Python Add-in Project. The user creates a project by running the **addin_assistant.exe** program. A program dialog box appears to prompt the user to browse for a folder or create a new folder for the “Add-In” project root folder. The folder can be any name chosen by the user.



Information about the new add-in can be entered. Only the **add-in name** (Name) and **version** are required.



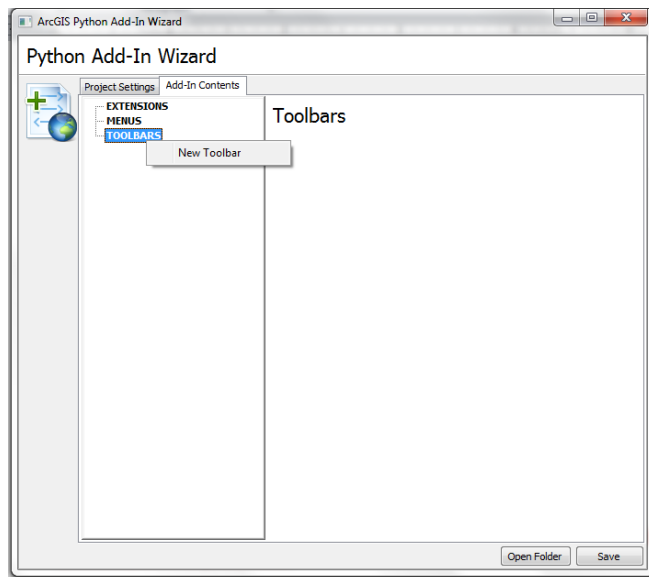
For many new add-ins, a custom toolbar is used. The **Add-In Contents** tab is used to create specific Python Add-in options.

Creating a Custom Toolbar

Three options appear on the **Add-Ins Content** Tab:

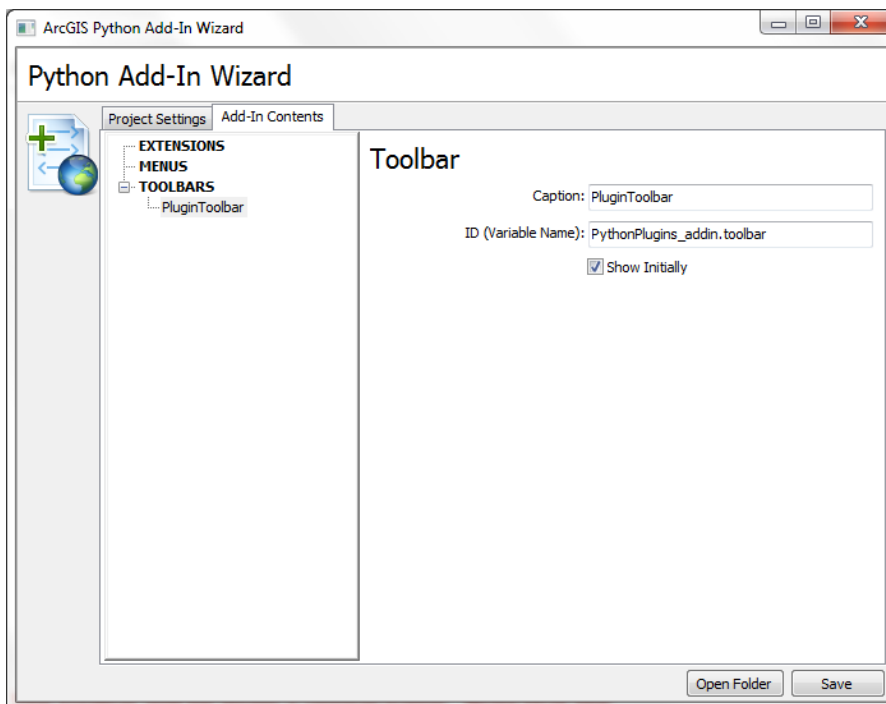
1. *Extensions* – create an Add-In extension
2. *Menus* – create menus
3. *Toolbars* – create a toolbar

To create a new toolbar, right click on **Toolbars** and select **New Toolbar**.



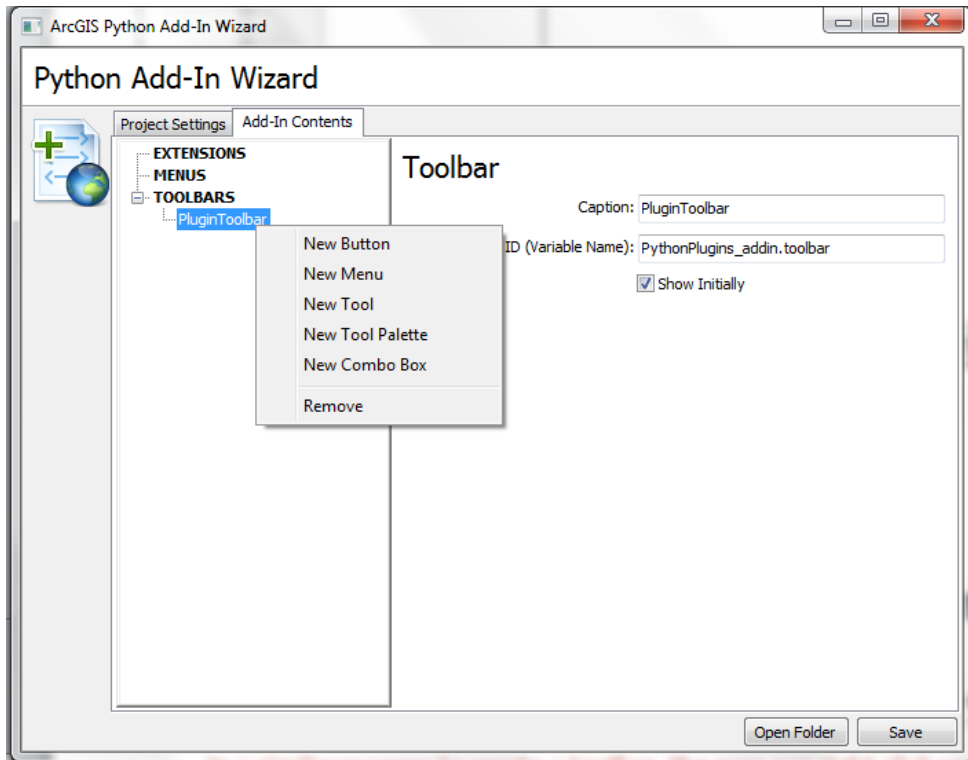
The toolbar will be given a unique name. Note that the ID uses the name of the Python Folder created above. The general format for the ID is

`<name of the folder>.<type of object>` (i.e. a toolbar in this case).

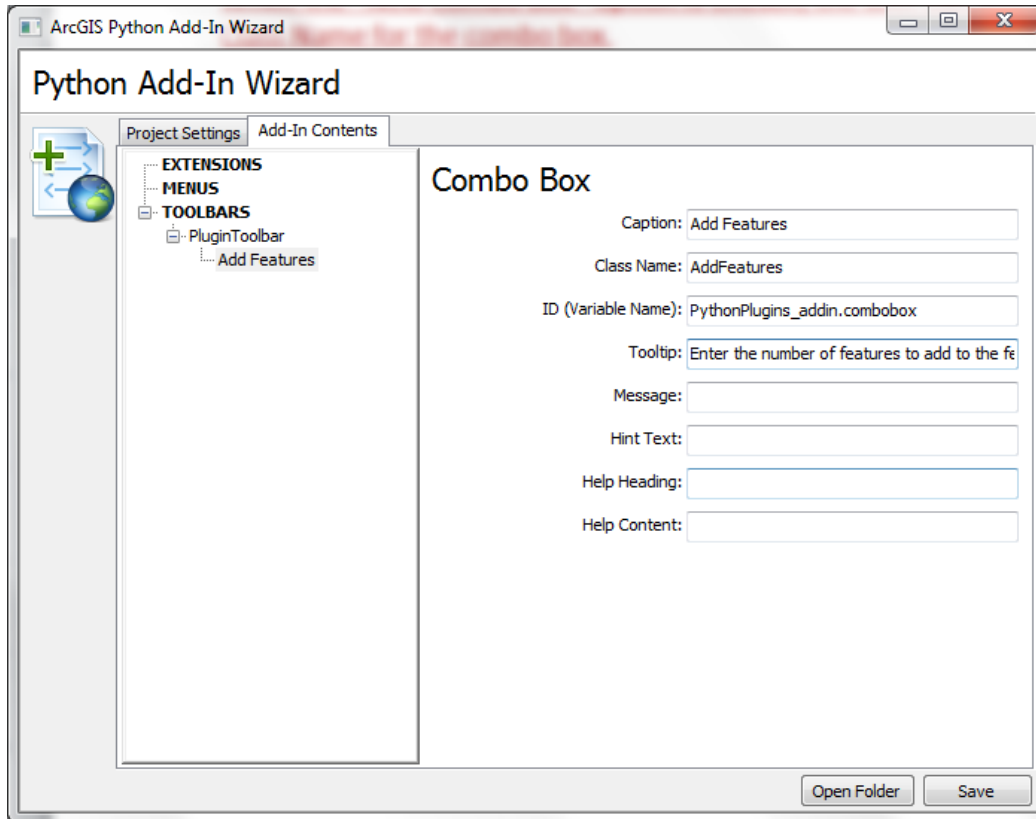


Adding an Interface to the Toolbar

Interface options can be added to a toolbar by right-clicking on the toolbar and choosing an option from the list.

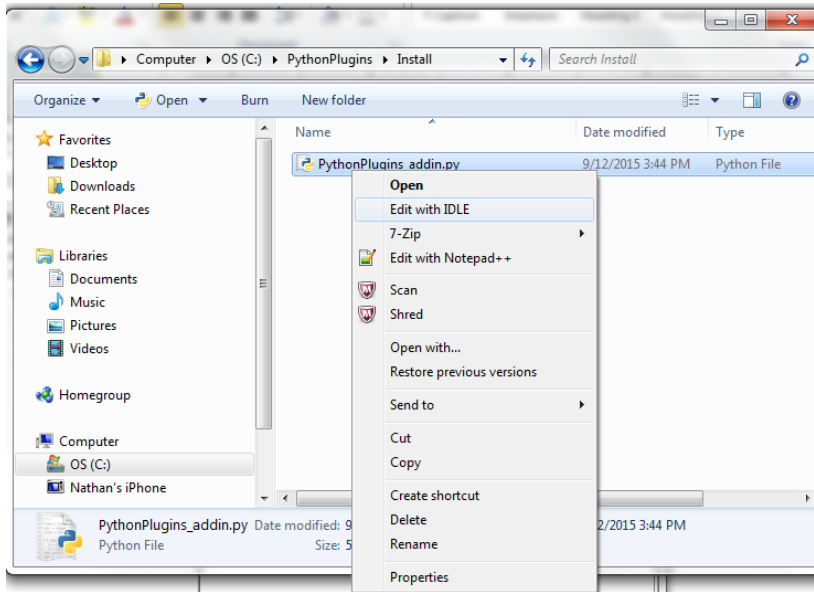


Different interface options have different properties. For example, the “New Combo Box” option has several properties for a caption, a class name, tool tips and tool help. A **class name** (e.g. **AddFeatures**) is required since this will be the name of the Python class used in the code to provide the interface functionality. The **caption** provides a useful name to the user. Spaces cannot be included in the class name. Once changes are made to the interface, **Save** is used to save the changes.



Adding Functionality to the Python Add-in

The next step is to modify the Python stump code to provide functionality to the interface. Click the **Open Folder** button in the Python Add-in Wizard. The folder created above appears. Double click the **Install** folder and then right click to open and edit the Python script file for the specific add-in project.



The user is now able to write custom code to add functionality to the interface just created.

The Python script below is automatically created when the Python Add-in is created and saved. Note the **AddFeatures** class appears which contains a number of Python functions. The name of the Python class is derived from the name of the add-in object created in the previous steps. Depending on the type of interface added to the toolbar, different functions may appear. For the combo box, unique functionality can be coded for the combo box when it initially appears in ArcMap. These are the available functions that can be coded for the combo box.

```

__init__(self) – sets the initial state of the combo box
def onSelChange – implemented when a value from the combo
    box drop down list has been chosen
def onEditChange – implemented when a new character has been entered in the
    combo box
def onFocus – implemented when combo box is clicked on by the user
def onEnter – implemented when the user taps the “Enter” keyboard button
def refresh – implemented after a value has been entered or chosen from a drop
    down list into the combo box. Refresh makes sure the value appears in the
    combo box.

import arcpy
import pythonaddins

class AddFeatures(object):
    """Implementation for PythonPlugins_addin.combobox
    (ComboBox)"""
    def __init__(self):
        self.items = ["item1", "item2"]
        self.editable = True
        self.enabled = True
        self.dropdownWidth = 'WWWWWWW'
        self.width = 'WWWWWWW'
    def onSelChange(self, selection):
        pass
    def onEditChange(self, text):
        pass
    def onFocus(self, focused):
        pass
    def onEnter(self):
        pass
    def refresh(self):
        pass

```

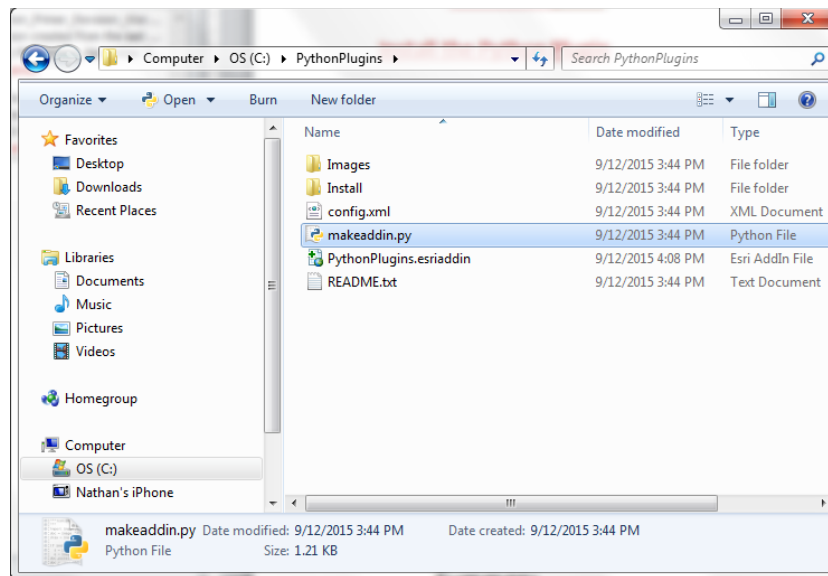
Installing the Python Add-in

For now the above code is left unchanged, but normally, a programmer would add and test Python code within one or more of the functions. **Demo 11** shows step by step instructions for creating, modifying, installing, and using a Python Add-in.

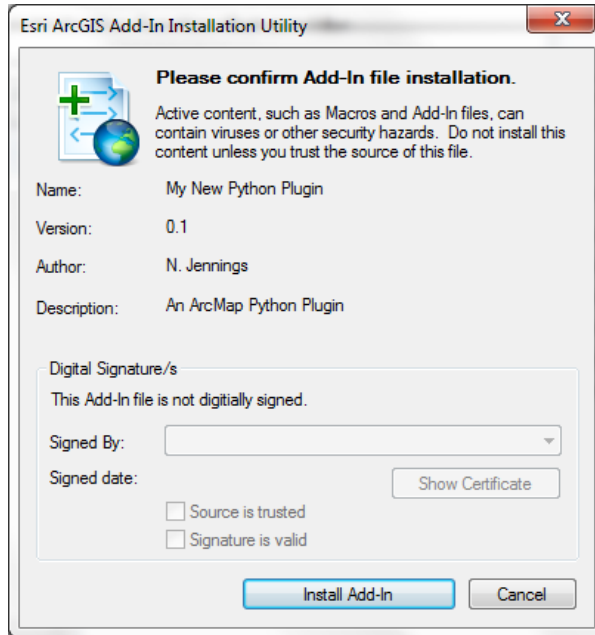
To install a Python Add-in, a two-step process is involved. Before implementing these steps, make sure to close out of all instances of ArcMap and ArcCatalog.

1. Run (double click) the **makeaddin.py** file – this script is found in the Add-in folder created above and is automatically added for each Add-in. The **makeaddin.py** creates an install package for the Add-in in the same folder and has the general format.

`<Add-in name>.esriaddin`



2. Run (double click) the **<Add-in_name>.esriaddin** file (the ESRI Data Add-in File) – this allows ArcGIS to add (install) and use the Python Add-in within ArcMap or ArcCatalog. When the user double clicks the ESRI Data Add-in file, the following dialog box appears. Click the **Install Add-in** button to perform the installation of the Python Add-in.



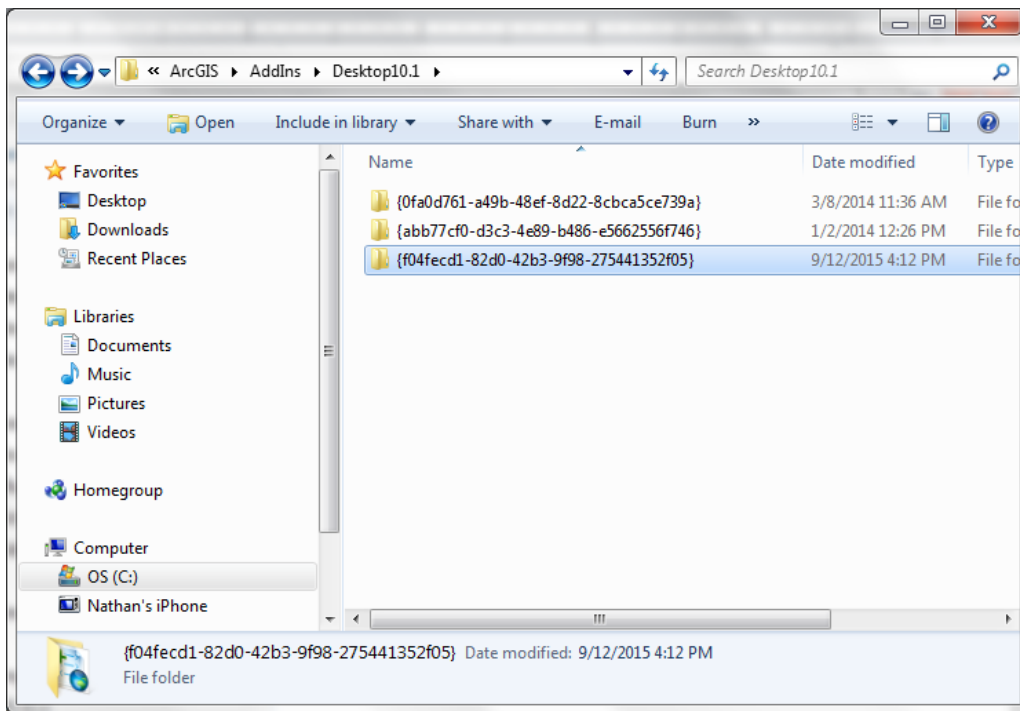
An *"Installation Succeeded"* message appears. The add-in is actually installed in the "users" default folder. Different operating systems may have a different path.

C:\users\<username>\My Documents\ArcGIS\AddIns\Desktop(version)

Note that if ArcGIS has been “updated” since ArcGIS 10.1, the new add-in may appear in the Desktop folder of a previous version.

A unique folder named with a globally unique identifier (GUID) is automatically created when the add-in is installed. Add-ins can be installed on newer releases of ArcGIS after they are created. They may appear by default in ArcMap (or ArcCatalog) if a user has been using the add-in and has made the toolbar visible. ESRI recommends using the **Add-in Installation Utility** (vs manually copying) to install Add-ins for Desktop applications.

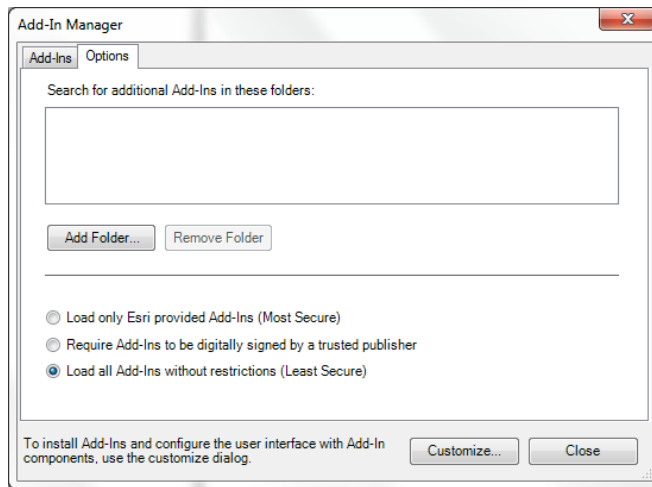
Note below that the add-in was added to the Desktop 10.1 folder even though the author has a newer version of ArcGIS installed.



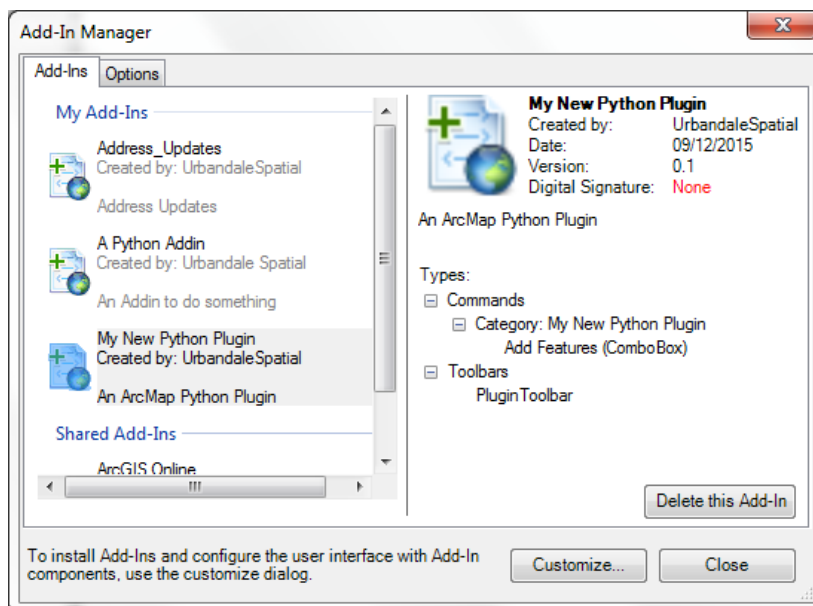
Using the Python Add-in

The new toolbar and tools appear in ArcMap or ArcCatalog after the Python Add-in is installed. If the new add-in does not appear, the following options can be reviewed.

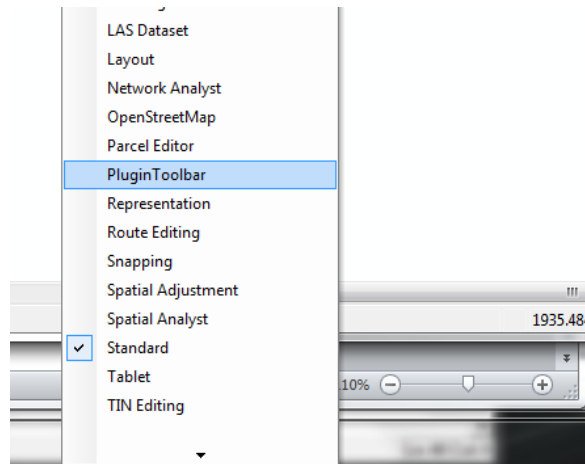
Open ArcMap (or ArcCatalog). Click **Customize—Add-ins Manager**. The add-in may appear if the Add-ins Manager Options is set to “*Load all Add-Ins without restrictions (Least Secure)*.” This may need to be changed if the add-in does not appear in the list of Add-ins.



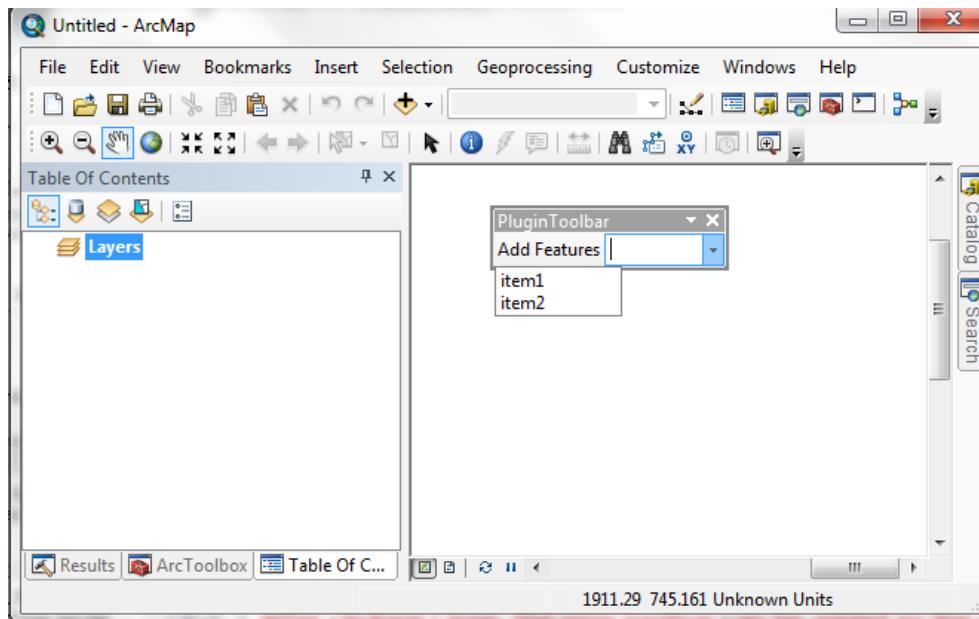
The user can go back to the Add-ins list and select the Add-in to load in ArcMap. Click Close. The Add-in will appear in the list of Toolbars to Add to the ArcMap or ArcCatalog.



After clicking Close, the new toolbar can be added to the ArcMap document (or ArcCatalog) (**Customize—Toolbars** or right click in the toolbar area and find the custom toolbar).



The new toolbar appears with the combo box interface. In this example since this interface does not have any functional coding behind it, only the interface appears in ArcMap.



Notice the “*item1*” and “*item2*” menu options. These values were prepopulated in the initialization function when the Python script was automatically created. If these options are not needed, they can be removed, otherwise, the programmer can add unique values to prepopulate a dropdown list of values for the combo box.

Modifying a Python Add-in

Before making any changes, make sure to back up existing code in a separate folder as a precaution. If interface changes are required (such as to the toolbar), the **Addins_assistant** executable (i.e. the Python Add-in Wizard) should be used to make modifications. Do not make changes in the back up. When saving new interface changes, a message may appear indicating that a backup of the existing script is saved. This is fine. Depending on the kinds of changes, some code from the “back up” script may need to be added back in to the newly modified add-in interface. Make sure these are added in the proper location within the script.

In addition, the Python script found in the **Install** folder can be opened directly and modified as required. Only modify the script directly if no other interface options are added or changed. If changes to the interface are required use the previous method. Make sure to “NOT” change the class name. These class names are referenced in accompanying configuration files when the add-in is installed. Making changes to the interface can cause the “Add-in” not to function properly. Once script changes are made:

1. Run (double click) the **makeaddin.py** routine to “remake” the add-in.
2. Run (double click) the **<Add-in_name>.esriaddin** routine to update the add-in for ArcGIS.

Debugging a Python Add-in

As previously recommended (and as a common best practice), using *print* statements throughout the code development process can be helpful. With respect to a Python Add-in, *print* statement results will appear in the Python Window when testing the add-in functionality within ArcMap or ArcCatalog.

On occasion, a “missing” object may appear in the toolbar (i.e. the toolbar will show a tool with the word “Missing” and a red “X”). This is often caused by incorrect syntax within the Python script. Use the “Check” option within the Python editor to make sure the script syntax is properly written.

Sharing a Python Add-in

Python add-ins can be shared on a network. The custom add-in folder (such as that created above) and its contents can be copied to a common network location. In the **Add-In Manager** in ArcMap or ArcCatalog the UNC (server/network share) path can be added in the **Options** tab. Users will be able to obtain the latest version of the add-in from a central location. New add-ins can be discovered using this mechanism and users can add the toolbar or functionality as described above.

Summary

Chapter 11 showed how Python and `arcpy` can be used to create some simple interfaces to ArcGIS. Even with a limited set of user interfaces, Python provides the ability for powerful processing of GIS data. The Python Add-in provides an alternative for the code developer so that the end user can interact with custom functionality that is not “out of the box” for ArcGIS. In addition, the Add-in allows for the user to directly interact with the ArcMap data view and possibly make changes to GIS data. The Python Add-in is typically a “second phase” of programming and depends on the needs and requirements of the end user and his/her GIS skill.

Chapter 11 Demo Adding Address Units using a Python Add-in

The following example demonstrates the use of a custom Python Add-in to automatically add additional point features (addresses) that represent dwelling units for a “base” address (e.g. 123 Main St may have units 1, 2, 3, and 4). Instead of a user interactively clicking the map multiple times to add unique unit point locations, the user can select a specific existing “base” address point feature and create additional point features that contain the exact same information as the “base” address, but have a unit field automatically updated with the specific unit number.

Prerequisites

The Python Add-in Wizard (**addin_assistant.zip**) must be downloaded and installed (i.e. the **addin_assistant.exe** file exists within a user specified folder). Go to **arcgis.com** and search for “Python Add-in Wizard” or do an Internet search and search for “ArcGIS and Python Add-in Wizard.” (Refer to the chapter for more details). It is recommended to make a short cut on the Desktop to the **addin_assistant.exe** file so it can be easily accessed.

Overview

The user can create a new address point in the address feature class and add the proper address fields for the address and as well as enter the first unit number in the UNIT field or the user can select an existing address (presumably with an existing unit). The user will then use the **Address Unit Update Add-in** to enter the number of units to add to the selected address. When the user taps the enter key, the number of new address points will be added to the feature class and all of the attribute information for the new addresses will be updated appropriately. The UNIT value for the new address locations will be updated based on the existing UNIT value in the “selected” address.

The concepts illustrated in the chapter and demos are:

ArcGIS Concepts

Add-ins
addin_assistant
esriaddin
cursors

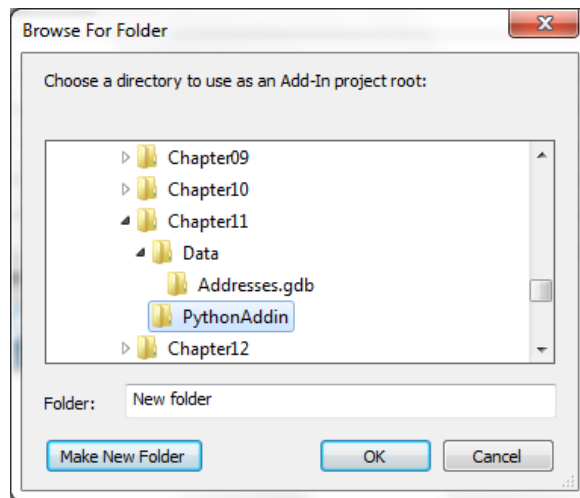
Python Concepts

functions

STEP 1 - Create a New Python Add-in Project

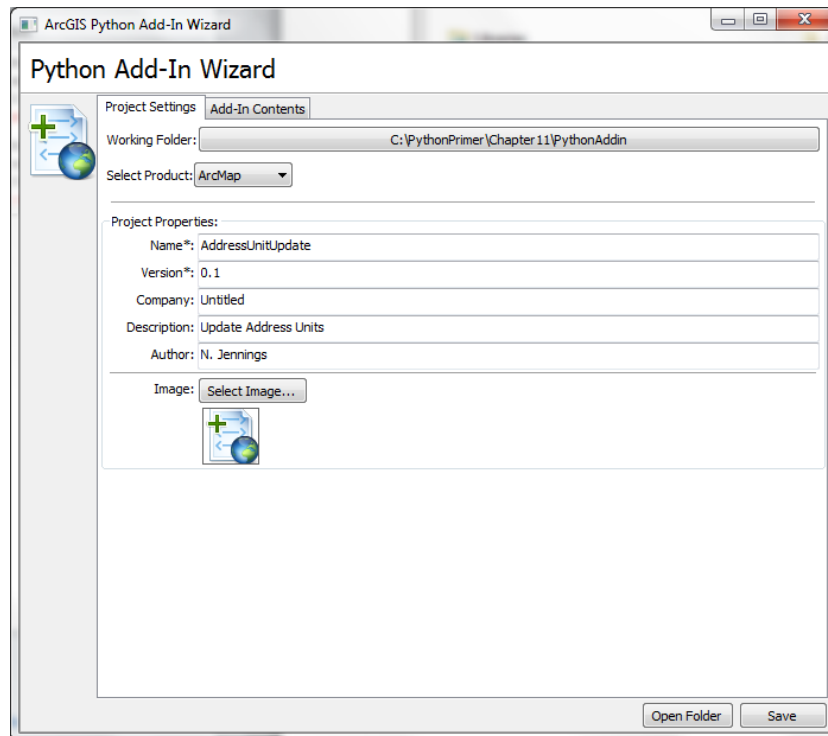
The working Python Add-in, map document, and data are provided so the reader can review these in the steps below. It is recommended that the reader create a different folder for the Python Add-in as well as create a new map document and add the address and city parcels data to it to perform the steps in the demo. Make sure to create a different folder, add-in project name, add-in toolbar, and add-in while working through the demo.

1. Double click on the **addin_assistant.exe** file.
2. Browse to a folder of your choice (recommended to use the **\PythonPrimer\Chapter11** folder) to create a new folder called **PythonAddin**.

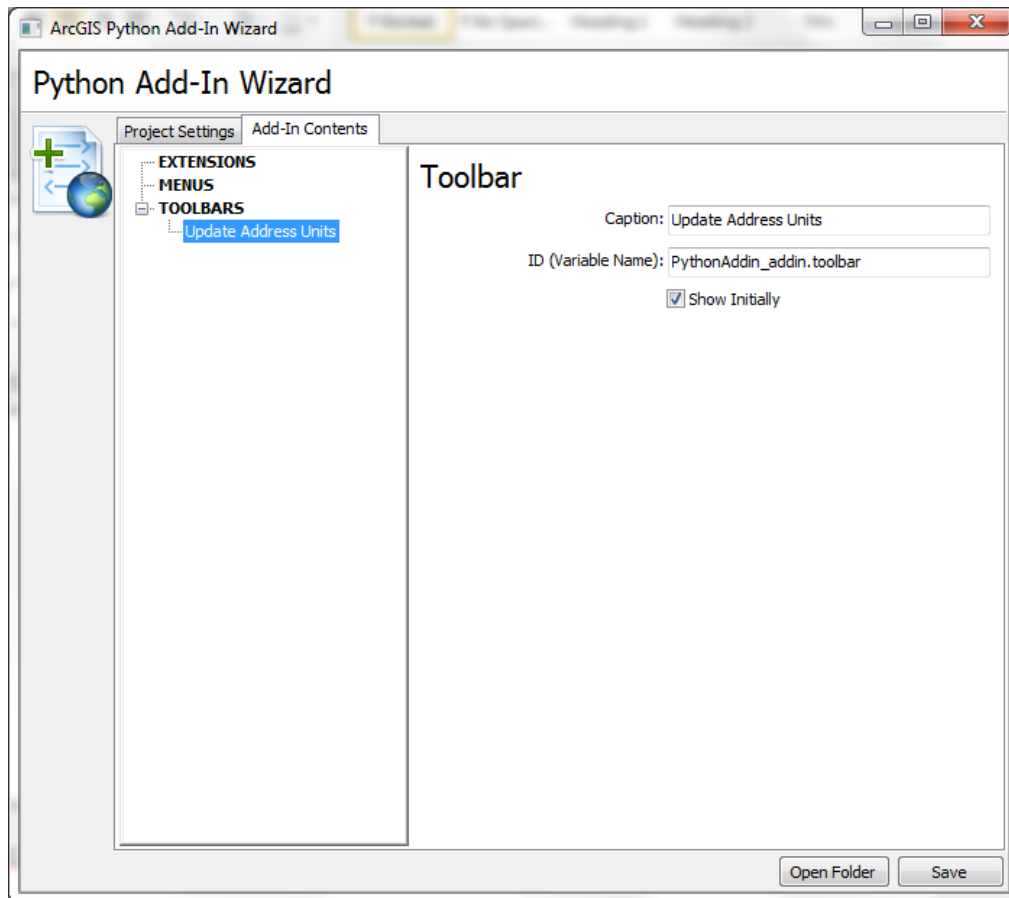


3. Click OK.

4. Modify the Add-in name to **AddressUnitUpdate** (no spaces) and fill in the other blanks if desired.



5. Click on the **Add-in Contents** tab and then right click on Toolbars. Create a new toolbar and name it **Update Address Units**.

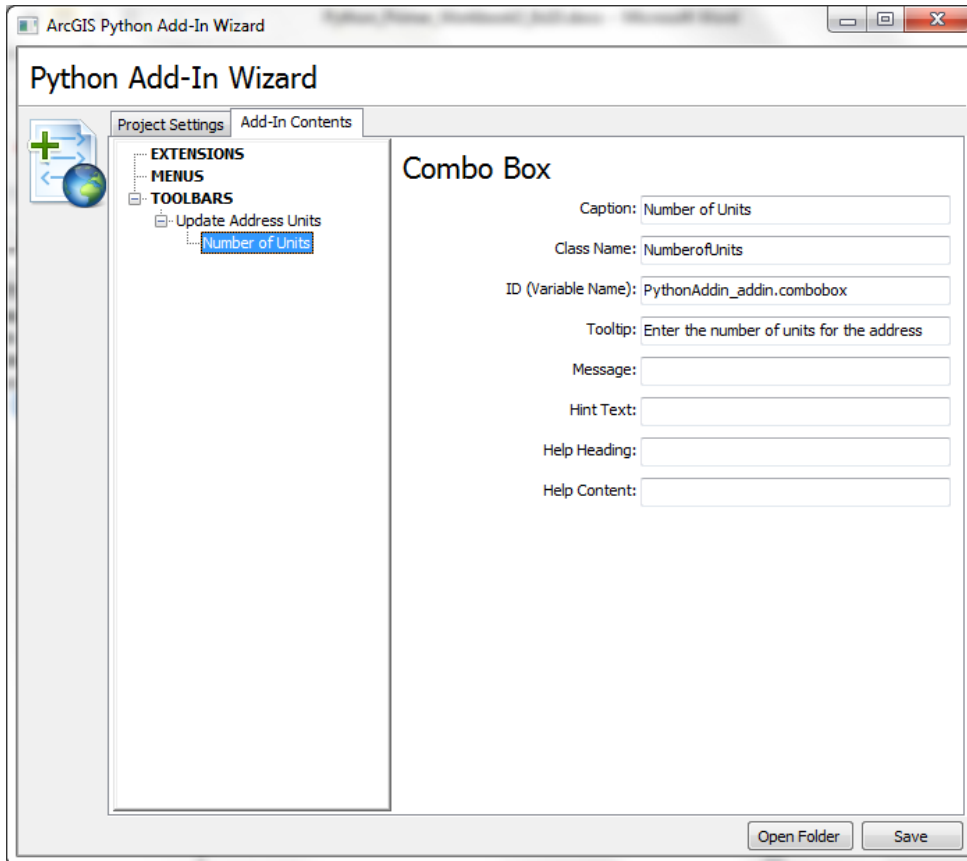


6. Right click on the toolbar just created. Right click and choose **New Combo Box**. Assign the following:

Caption – Number of Units

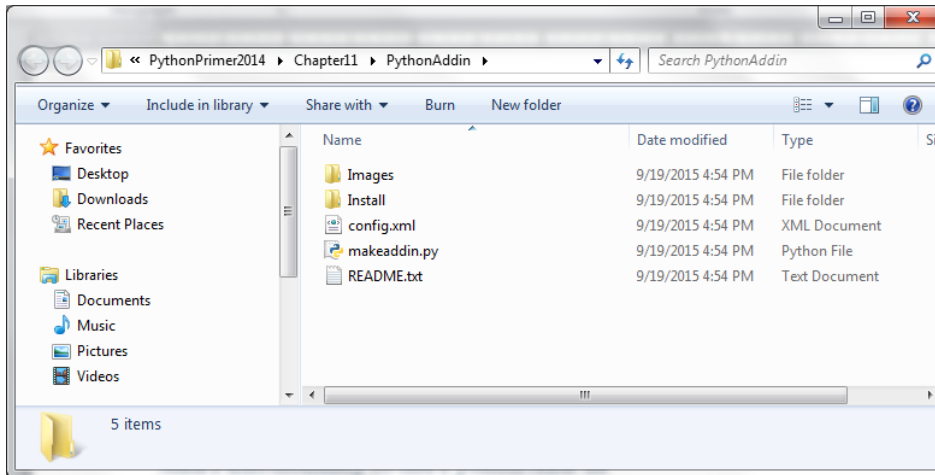
Class Name – NumberofUnits (note, no spaces)

Tool Tip - Enter the number of units for the address



7. Click Save and then click Open Folder

A series of folders appear. The **Install** folder will contain the Python script used for the add-in. If only changes need to be made to the functionality of the add-in (such as adding specific code for the different addin functions), the Python script in the **Install** folder will be modified and saved. The **makeaddin.py** file will be used to “make” (the first time the addin is created) or “remake” when subsequent addin changes are made to the code.



STEP 2 - Add Functionality to the Python Add-in

1. Double click on the **Install** folder and then right click on the **PythonAddin_addin.py** file to open the script file in IDLE. The following script appears.

```
import arcpy
import pythonaddins

class NumberofUnits(object):
    """Implementation for PythonAddin_addin.combobox
    (ComboBox)"""
    def __init__(self):
        #self.items = ["item1", "item2"]
        self.editable = True
        self.enabled = True
        self.dropdownWidth = 'WWWWWWW'
        self.width = 'WWWWWWW'
    def onSelChange(self, selection):
        pass
    def onEditChange(self, text):
        pass
    def onFocus(self, focused):
        pass
    def onEnter(self):
        pass
    def refresh(self):
        pass
```

Note the different functions provided. In this interface only the `onEnter` routine needs to be updated. The `def __init__(self)` routine does not need to be updated, since there is no “initial state” the interface requires.

2. Comment out the “bold” line above. This line is not required in the initialization function (i.e. the combo box will not have a set of drop down values).

The user will add in a number (for the number of units to add) for this field into the user interface of the combo box. If the line was left in the routine, then the combo box would appear with the drop down values “item1” and “item2”.

3. Open the **onEnterDEF.py** file in the **Chpater11** folder and replace the `onEnter` function with the code in the file. Make sure the proper indentation is maintained when replacing the code. The reader should review the code to see if other changes are required before adding. For example, the path to the geodatabase may need to be

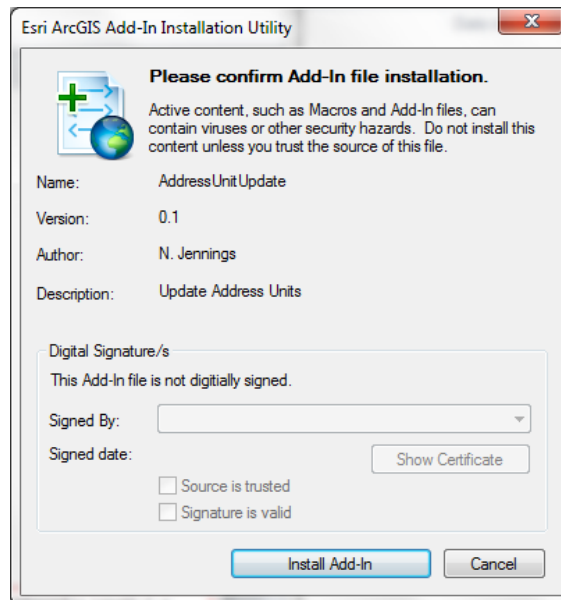
changed. Change the path in the code if the **addresses.gdb** is not located in the path shown below.

```
datapath = r"C:\pythonprimer\chapter11\data\addresses.gdb"
```

4. Save the **PythonAddin_addin.py** file (the Python file can remain open in IDLE).

STEP 3 - Make and Install the Python Add-in

1. In Windows Explorer open the **Chapter11\PythonAddin** folder.
2. Double click on the **makeaddin.py** file (this “makes or remakes” the Python Addin).
3. After this is complete (takes approx. 1 second), double click on the **PythonAddin.esriaddin** file. This actually installs the Addin to the computer system, so ArcMap documents (or ArcCatalog) can use the addin.
4. Click **Install Add-in** if this screen appears.



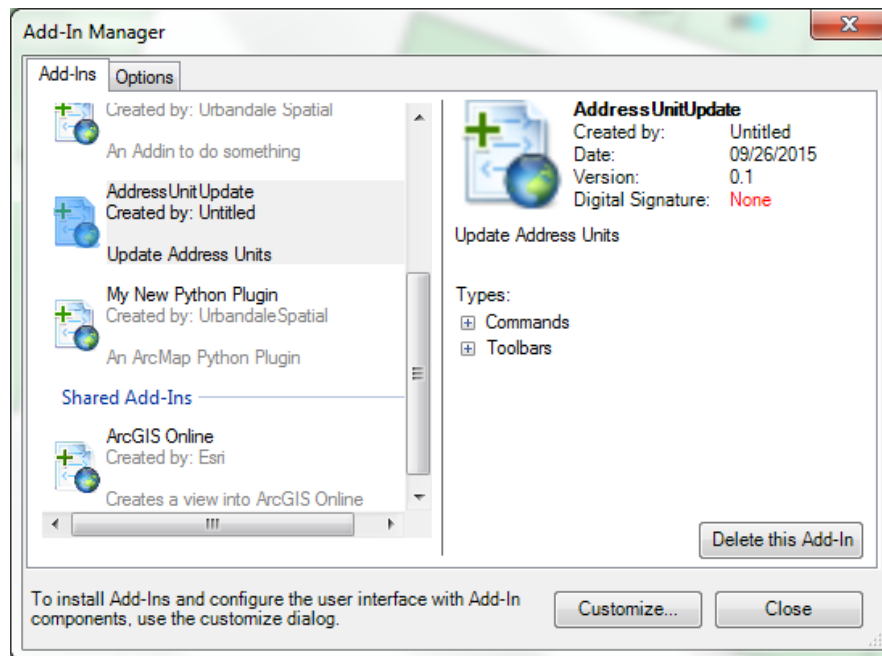
5. The following message may appear: “Installation succeeded.” Click Ok.

STEP 4 - Use the Python Add-in

1. Open the **Chapter11\Add_Address_Units.mxd**. The add-in may already appear.

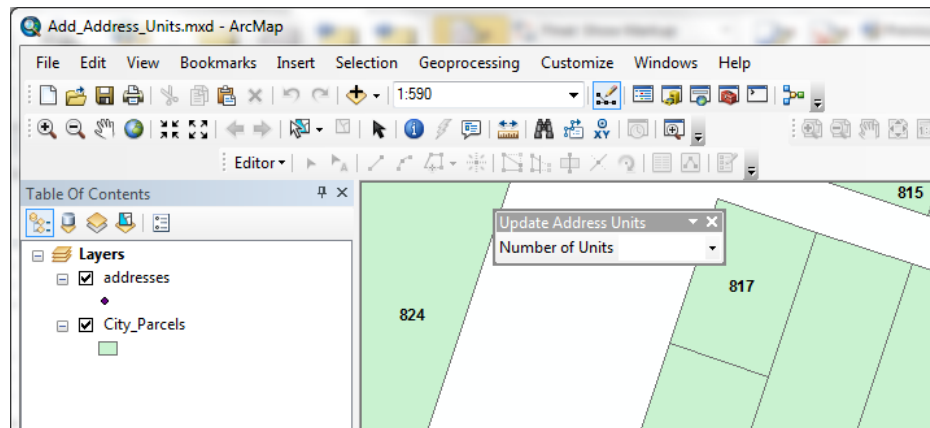
If the add-in does not appear, do the following.

Go to the **Customize—Add-ins Manager**. The **Address Unit Update** add-in may appear if the Add-ins Manager Options Tab is set to *“Load all Add-ins without restrictions (Least Secure).”* This may need to be changed if the add-in does not appear in the list of add-ins.



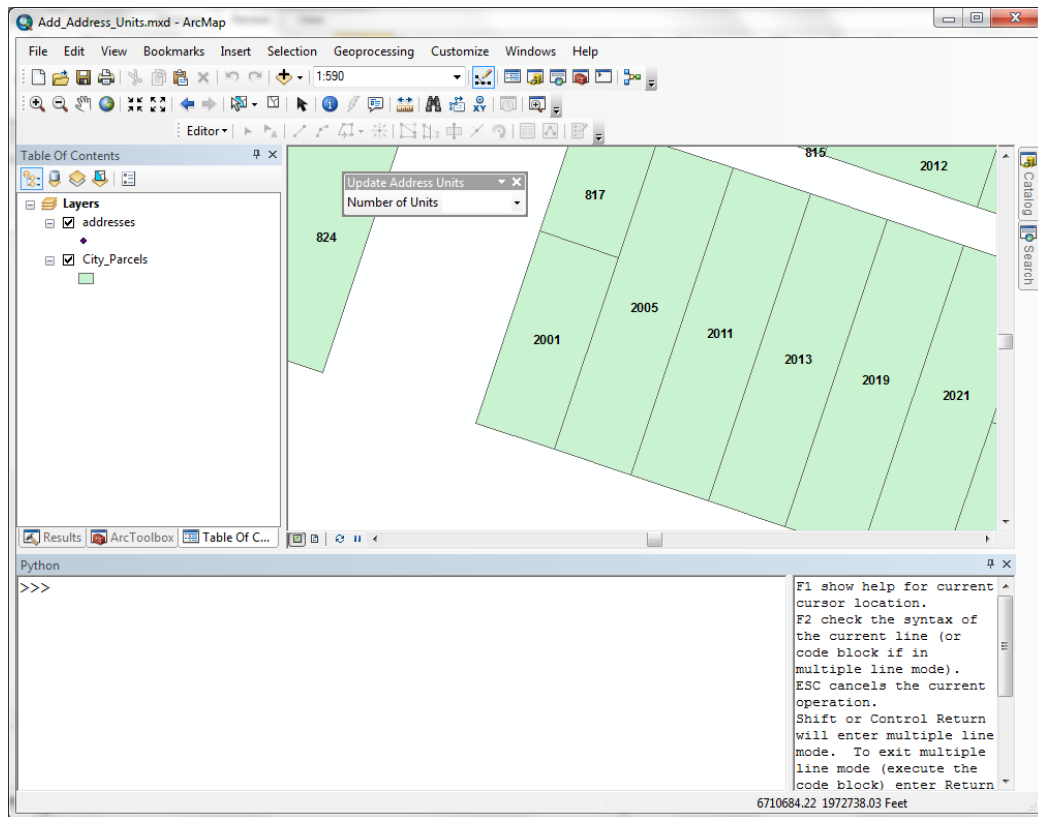
Click Close after verifying that the **Address Unit Update** add-in is present in the Add-in Manager.

2. To find the **Update Address Units** Toolbar (that was created as part of the **Address Unit Update** add-in), click **Customize—Toolbars** or right click in the toolbar area, find the **Update Address Units** toolbar from the list, and check it to make it visible in ArcMap.

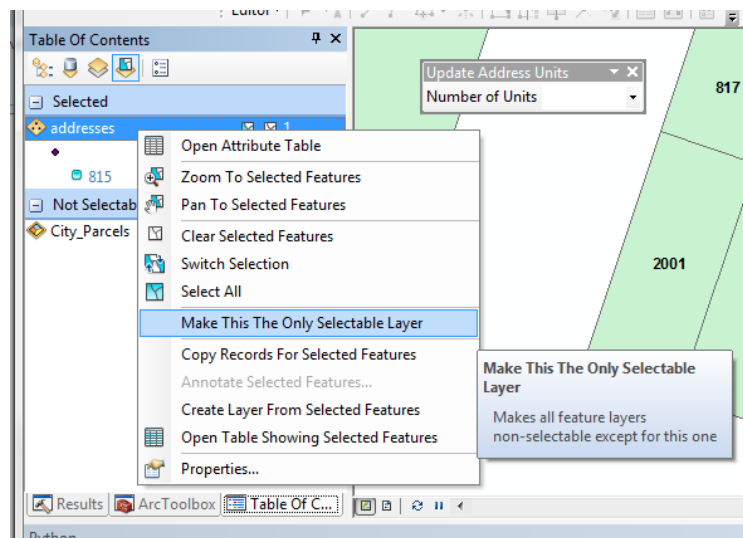


3. Add the **Python Window** (click the Python Window button at the top of ArcMap or **Geoprocessing—Python**) and dock it to the bottom of ArcMap. It may be useful to see the Python Window, since a few “print” statements are provided to show results of the **Update Address Units** routine.
4. Zoom into an area where the address numbers appear in the map.

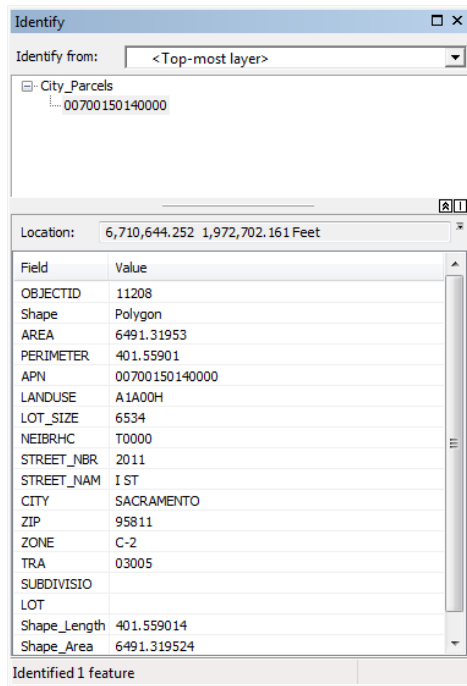
The ArcMap document should look similar to the following illustration.



5. Click on the **List by Selection** icon in the table of contents (fourth icon from the left just under the Table of Contents header). Right click and make sure only the **addresses** layer is the only selectable layer.

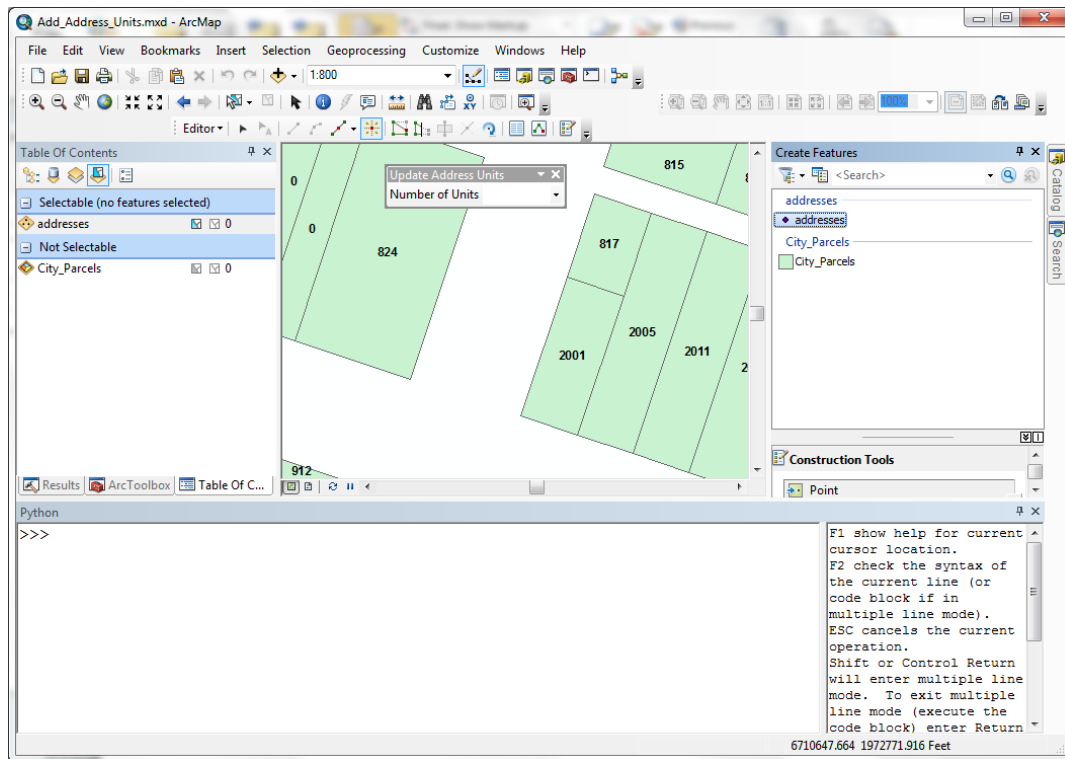


6. Use the ID tool to identify a parcel and view the parcel information. In this example the information for **2011 I St** is shown. See the **STREET_NBR** and the **STREET_NAM** fields below. The user may have chosen a different parcel (this is ok).



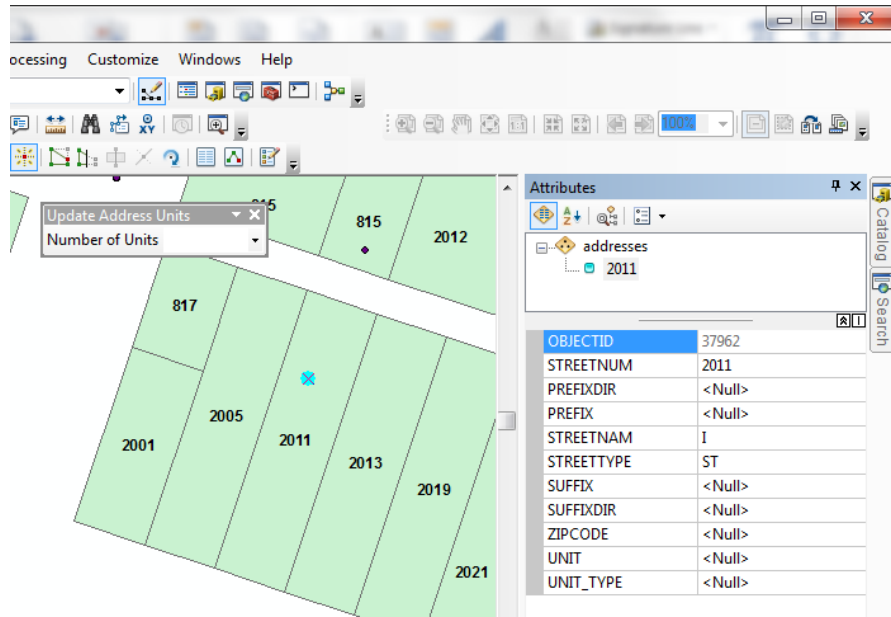
7. Move the Identify window out of the way of the map.
8. Start an **Edit** session. NOTE: only address points will be edited for this exercise.
9. Click on the **Create Features** tool in the **Edit Toolbar**.

10. Click on the **addresses** feature from the **Create Features** list.



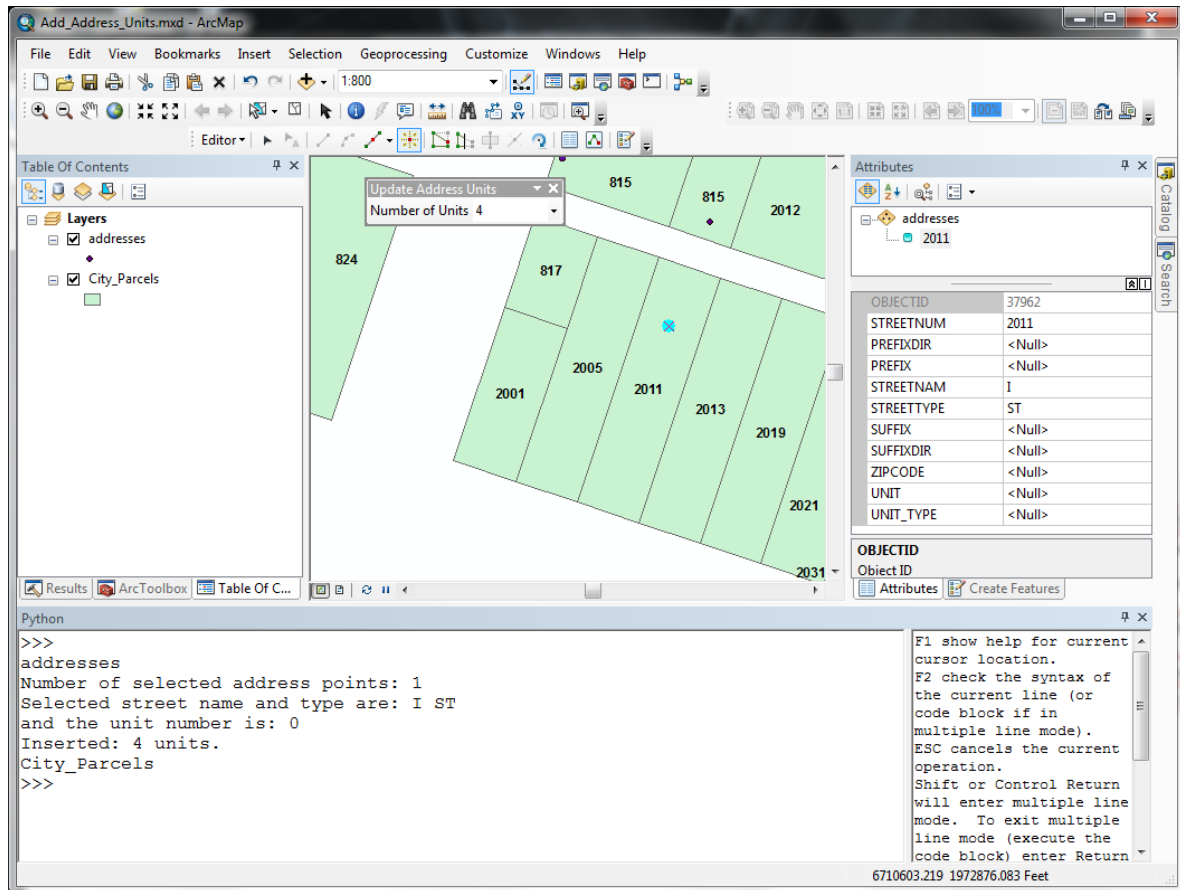
11. Click within the **2011 I St** parcel to add (create) a **“single”** (only one) address point.

12. Enter the street number, street name, and street type into the respective attribute fields. Note that the **UNIT** field is not filled in. This record serves as a “base” address number (an address that does not contain units).



13. Enter a value for the number of units in the **Update Address Units** toolbar in the **Number of Units** text box.
14. Tap the Enter key.

The Python window shows the print messages that appear when the add-in is run. The name of the street, the street type as well as the total number of “inserted” (created) units are reported in the Python window.



- Review the table to see that the number of “new” records is equivalent to the number entered in the add-in. NOTE: The add-in creates “stacked points” for the newly created units. The user will need to physically move them (or modify the add-in to do just this!) if desired. Moving the points programmatically could be challenging and is not the focus of this example.

OBJECTID *	SHAPE *	STREETNUM	PREFIXDIR	PREFIX	STREETNAM	STREETTYPE	SUFFIX	SUFFIXDIR	ZIPCODE	UNIT	UNIT_TYPE
37958	Point	222	<Null>	<Null>	ARDEN	BLVD	<Null>	<Null>	99999	11	APT
37959	Point	815	<Null>	<Null>	20TH	ST	<Null>	<Null>	<Null>	<Null>	APT
37960	Point	815	<Null>	<Null>	20TH	ST	<Null>	<Null>	<Null>	1	APT
37961	Point	815	<Null>	<Null>	20TH	ST	<Null>	<Null>	<Null>	2	APT
37962	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	<Null>	<Null>
37963	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	1	<Null>
37964	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	2	<Null>
37965	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	3	<Null>
37966	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	4	<Null>

16. Save the Edits and Stop Editing.

The **Address Unit Update** add-in can also be used outside of an “Edit” session. The only requirement is that a single address point be selected.

NOTE: An error handling message has not been added to the demo to check to see if more than one or “no” address points are selected. A keen programmer could add unique error handlers if he/she wishes!

17. Open the addresses attribute table. Select the last record added above (e.g. **2011 I St Unit 4**). There should only be **one** record selected.

OBJECTID *	SHAPE *	STREETNUM	PREFIXDIR	PREFIX	STREETNAM	STREETTYPE	SUFFIX	SUFFIXDIR	ZIPCODE	UNIT	UNIT_TYPE
37957	Point	222	<Null>	<Null>	ARDEN	BLVD	<Null>	<Null>	99999	10	APT
37958	Point	222	<Null>	<Null>	ARDEN	BLVD	<Null>	<Null>	99999	11	APT
37959	Point	815	<Null>	<Null>	20TH	ST	<Null>	<Null>	<Null>	<Null>	APT
37960	Point	815	<Null>	<Null>	20TH	ST	<Null>	<Null>	<Null>	1	APT
37961	Point	815	<Null>	<Null>	20TH	ST	<Null>	<Null>	<Null>	2	APT
37962	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	<Null>	<Null>
37963	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	1	<Null>
37964	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	2	<Null>
37965	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	3	<Null>
37966	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	4	<Null>

18. Add a number of units to the **Update Address Units** toolbar (e.g. 5).

19. Tap the Enter key.

20. Refresh the table (click the F5 key) or close and then reopen the table. Scroll to the bottom of the table to see that new records are created. Since the address selected had a unit number of 4, the unit number of the “new” addresses increment by 1 for each address that was added.

Table											
addresses											
OBJECTID *	SHAPE *	STREETNUM	PREFIXDIR	PREFIX	STREETNAM	STREETTYPE	SUFFIX	SUFFIXDIR	ZIPCODE	UNIT	UNIT_TYPE
37962	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	<Null>	<Null>
37963	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	1	<Null>
37964	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	2	<Null>
37965	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	3	<Null>
37966	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	4	<Null>
37967	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	5	<Null>
37968	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	6	<Null>
37969	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	7	<Null>
37970	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	8	<Null>
37971	Point	2011	<Null>	<Null>	I	ST	<Null>	<Null>	<Null>	9	<Null>

1 (1 out of 57 Selected)

addresses

21. Save the ArcMap if desired.

Chapter 11 Questions

1. What program must be downloaded and installed to create a Python Add-in?
2. Name at least three types of interfaces that can be coded using Python Add-ins.
3. What are the two steps to create and install a Python Add-in?
4. What is one reason why a programmer would choose to create a Python Add-in versus using a custom ArcToolbox with a Python script tool?

Chapter 12 Automating Geoprocessing Scripts

Overview

Throughout *A Python Primer for ArcGIS* the scripts have primarily been executed through the Python IDLE window or other Python editor. Chapters 10 and 11 illustrated other methods for implementing geoprocessing Python scripts by using custom and Python script tools in an ArcGIS toolbox or through custom user interfaces in custom ArcGIS toolbars. One of the primary reasons for writing Python scripts is to automate repetitive geoprocessing tasks that normally take considerable manual interaction with ArcMap or ArcCatalog. Throughout the workbooks a number of common geoprocessing tasks have been written in a logical order so that they can be “automatically” executed, if needed. Up to this point one requirement to execute Python scripts has been to do one of the following:

1. Click Run in Python IDLE
2. Fill in parameters to run a custom geoprocessing script tool
3. Interact with a Python Add-in

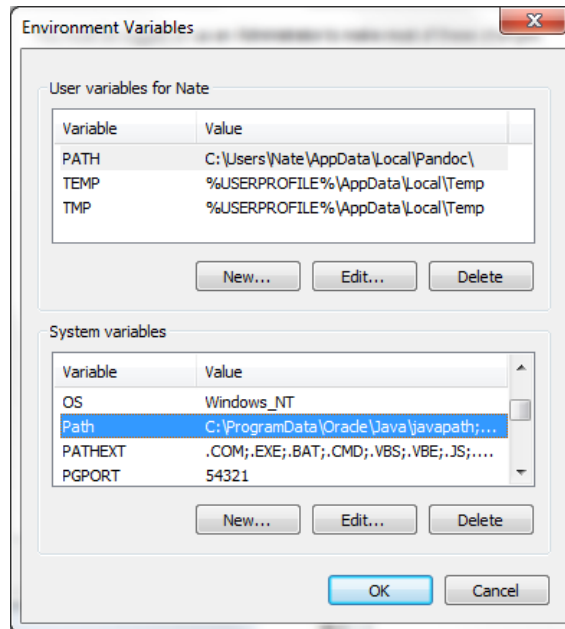
An alternative method not yet discussed is using the “command line” to execute a Python script. This involves typing in the syntax to execute a Python script as well as the specific parameters the script requires in the Command Prompt window. (See the figures throughout this chapter that show the Command Prompt window). This chapter illustrates this method, since it is the key to automatically run (i.e. batch process) a Python script.

A “batch file” is assigned to the Windows Task Scheduler to run a Python script at a specified time and/or frequency so that a specific user does not have to physically run the script. The batch file contains all of the information to automatically run a geoprocessing script.

Prerequisites

Make sure that Python is listed in the computer's "Path" Environment Variable. This allows the Command Prompt to execute Python scripts properly.

1. Click on the Windows **Start** button in the lower left (for Windows 7 machines. The reader should search the Internet for the appropriate location to make **Path Environmental Variable** changes on different Windows operating systems).
2. Click on the **Computer** option
3. Click on the **System Properties—Advanced system settings**
4. Click on the **Advanced** Tab
5. Click on the **Environmental Variables** button
6. Under the **System variables**, scroll down and locate the **Path** variable

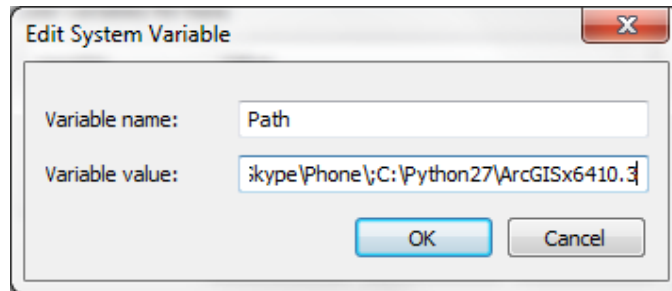


7. Click **Edit**
8. At the end of the "Variable value" list, add the following syntax to the **Path** variable if it does not already exist.

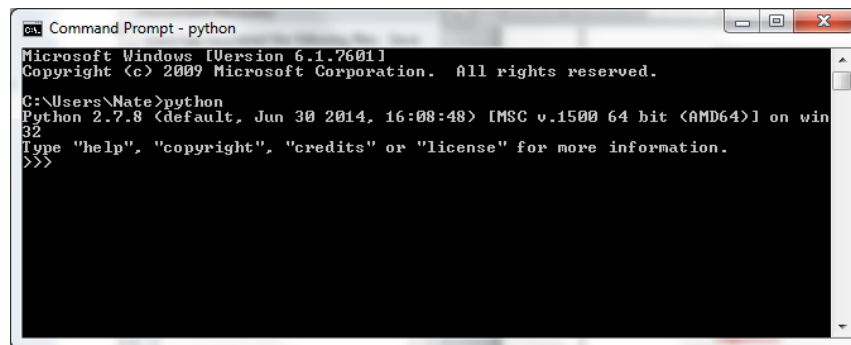
Note that that the default location for Python used with ArcGIS is in an ArcGIS folder that contains the bit type (32 or 64 bit) of the operating system and the version of ArcGIS (e.g. 27 for Python 2.7, 64 bit, ArcGIS version 10.3). The reader should make the appropriate modifications to the **PATH** variable, depending on the operating

system bit type and ArcGIS version. The leading semicolon is used to separate one path variable from another.

```
;C:\Python<version number>\ArcGISx6410.3
```



9. Click OK on all of the dialog boxes until the computer system properties appear again.
10. As a test to make sure the Path was set properly, start the Command Prompt from **All Programs—Accessories—Command Prompt**
11. Type in `python` at the prompt. The Python prompt ("`<<<`") and the Python version number should appear in the Command Prompt window.



12. Click the Ctrl button plus the "Z" key (Ctrl+Z) to return to the Windows command prompt

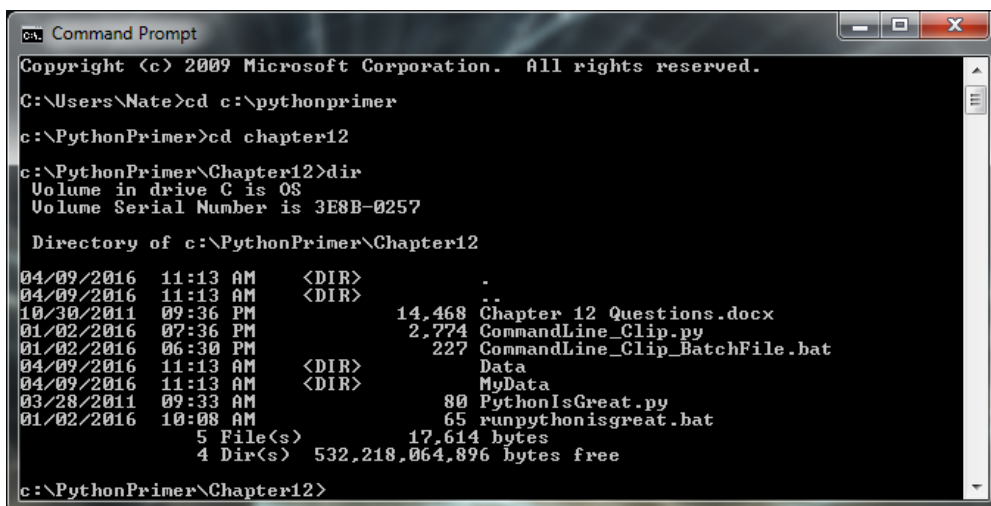
To run scripts with ArcGIS functionality, ArcGIS (at least Catalog) and the *arcpy* libraries must be installed on the system where script automation will occur (i.e. where batch files will be scheduled and processed). Normally, geoprocessing scripts are automatically processed on network servers within an organization. These servers will need to have ArcGIS (and optionally ArcGIS extensions) and the *arcpy* libraries installed.

The Batch File

Automated processing of programming routines is a common practice among programmers and is often performed by the use of a batch file (**.BAT**). Essentially, a batch file is a collection of one or more executable statement to run any number of processes without the user interacting with a specific program or routine. These could be data management activities such as copying data to different directories or folders, run executable commands with a variety of parameters, or run Python scripts that perform geoprocessing tasks and may also include parameters that are used by the script. Because Python files are “executable,” they function in a similar manner as other executable (.exe) files. Essentially, the batch file contains all of the required parameters to run the Python script.

Running a Python Script at a Command Prompt

As mentioned before, Python scripts can be executed through the Command Prompt. The Command Prompt is a window that contains a drive letter and any of the folders (directories) it contains. Before Windows environments were main stream to computers, the Command Prompt was the primary interface to interact with the computer. The Command Prompt persists so that computer users and programmers can execute (or run) commands and programs in an alternative manner to clicking an icon or navigating through a file management window (such as Windows Explorer) to locate a program and then execute it. Usually, the Command Prompt can be found under **All Programs—Accessories—Command Prompt** or the user can save a short cut to this window and place it on the desktop or in a launch bar at the bottom of the computer screen for easy access. The following figure represents the Command Prompt on a Windows operating system.



```

C:\ Command Prompt
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Nate>cd c:\pythonprimer
c:\PythonPrimer>cd chapter12
c:\PythonPrimer\Chapter12>dir
Volume in drive C is OS
Volume Serial Number is 3E8B-0257

Directory of c:\PythonPrimer\Chapter12

04/09/2016  11:13 AM  <DIR>          .
04/09/2016  11:13 AM  <DIR>          ..
10/30/2011  09:36 PM             14,468 Chapter 12 Questions.docx
01/02/2016  07:36 PM             2,774 CommandLine_Clip.py
01/02/2016  06:30 PM             227 CommandLine_Clip_BatchFile.bat
04/09/2016  11:13 AM  <DIR>          Data
04/09/2016  11:13 AM  <DIR>          MyData
03/28/2011  09:33 AM             80 PythonIsGreat.py
01/02/2016  10:08 AM             65 runpythonisgreat.bat
               5 File(s)          17,614 bytes
               4 Dir(s)  532,218,064,896 bytes free

c:\PythonPrimer\Chapter12>
```

The Command Prompt shows the drive letter at the top of the screen and a set of folders (directories and subdirectories); in this case **C:\PythonPrimer\Chapter12**. In this illustration the folder path indicates that the command prompt is at the *C drive* and has a directory called *PythonPrimer* and a subdirectory called *Chapter12*. The user can change locations in the Command Prompt by using common DOS (disk operating system) commands (such as **CD** for *change directory*). If the user types in **DIR** at the command prompt, a list of directories, subdirectories, and files can be seen. The reader can search the Internet for DOS commands to find out more information; however, for the purpose of creating batch files for automatically running Python scripts, all of the DOS commands that a code developer will typically use can be found in this chapter.

For the purposes of executing Python scripts, the user can change directories to the location of a Python script and type the following to execute a Python script.

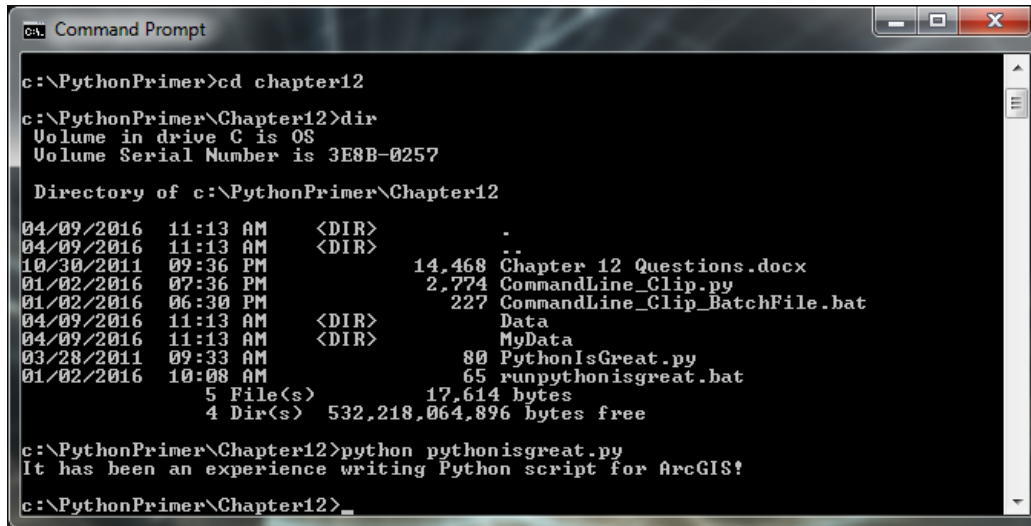
```
<drive><path where python file exists>python <name of python script>
```

For example, in the **pythonisgreat.py** script (found in the **\PythonPrimer\Chapter12** folder), the following can be written at the Command Prompt. If the reader copied the files to his/her local file system in the “C drive”, the following can be performed to run the script.

1. Start the Command Prompt (**All Programs—Accessories—Command Prompt**)
2. Type the command **CD C:\PythonPrimer\Chapter12** <enter>
3. Type the command **DIR** <enter>. Make sure the **pythonisgreat.py** file is listed in the directory (folder)
4. At the prompt (i.e. the location shown in the Command Prompt window) type the following: **python pythonisgreat.py** <enter>

In the figure below, the directory (folder location) the `python` command executes the `pythonisgreat.py` file.

`C:\PythonPrimer\Chapter12>python pythonisgreat.py`



```

c:\PythonPrimer>cd chapter12
c:\PythonPrimer\Chapter12>dir
Volume in drive C is OS
Volume Serial Number is 3E8B-0257

Directory of c:\PythonPrimer\Chapter12

04/09/2016  11:13 AM    <DIR>          .
04/09/2016  11:13 AM    <DIR>          ..
10/30/2011  09:36 PM             14,468 Chapter 12 Questions.docx
01/02/2016  07:36 PM             2,774 CommandLine_Clip.py
01/02/2016  06:30 PM             227 CommandLine_Clip_BatchFile.bat
04/09/2016  11:13 AM    <DIR>          Data
04/09/2016  11:13 AM    <DIR>          MyData
03/28/2011  09:33 AM             80 PythonIsGreat.py
01/02/2016  10:08 AM             65 runpythonisgreat.bat
               5 File(s)              17,614 bytes
               4 Dir(s)  532,218,064,896 bytes free

c:\PythonPrimer\Chapter12>python pythonisgreat.py
It has been an experience writing Python script for ArcGIS?
c:\PythonPrimer\Chapter12>_
  
```

When the “Enter” key is tapped after typing in `python pythonisgreat.py`, the Python file is executed and prints the statement above. The word *python* actually runs the **python.exe** file which then executes the Python script file, **pythonisgreat.py**. Since the path to the **python.exe** file is in a user’s computer “path” (i.e. the PATH environmental variable found under **My Computer—System Properties—Advanced Tab—Environment Variables**), the **python.exe** executable file can be run from any location simply by typing the word `python`. When the word `python` is typed at the Command Prompt, the Python Shell prompt appears (i.e. the Command Prompt shows (`<<<`)). See the Prerequisite section above.

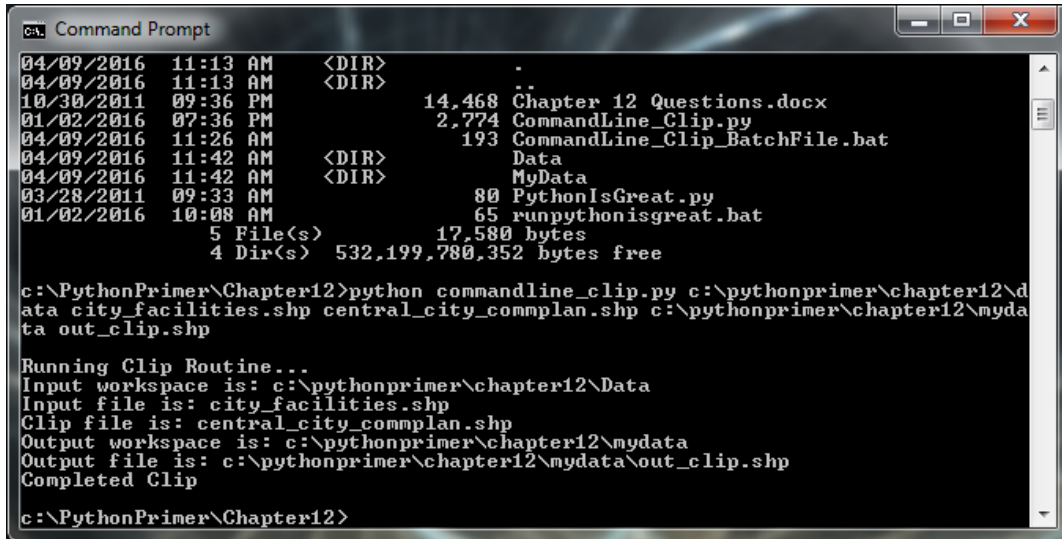
If a script requires parameters, the command line syntax shows the individual parameters separated by spaces. The **commandline_clip.py** script example used in the Chapter 12 demo (see the Chapter12 folder) contains five parameters:

1. *Workspace* – path to the folder containing data
(`c:\pythonprimer\chapter12\data`)
2. *Input file* – `city_facilities.shp`
3. *Clip file* – `central_city_commplan.shp`
4. *Output workspace* – path to a folder for output
(`c:\pythonprimer\chapter12\MyData`)
5. *Output file* – `out_file.shp`

Running the script through the Command Prompt, the following general syntax is used:

```
<command prompt>python <name of script> <parameter 1> <parameter 2> <...>
```

Notice each parameter is separated by spaces. For the **commandline_clip.py** script, the command line syntax looks like the following figure.



```

04/09/2016 11:13 AM <DIR> .
04/09/2016 11:13 AM <DIR> ..
10/30/2011 09:36 PM      14,468 Chapter 12 Questions.docx
01/02/2016 07:36 PM      2,774 CommandLine_Clip.py
04/09/2016 11:26 AM      193 CommandLine_Clip_BatchFile.bat
04/09/2016 11:42 AM      Data
04/09/2016 11:42 AM      MyData
03/28/2011 09:33 AM      80 PythonIsGreat.py
01/02/2016 10:08 AM      65 runpythonisgreat.bat
          5 File(s)      17,580 bytes
          4 Dir(s)  532,199,780,352 bytes free

c:\PythonPrimer\Chapter12>python commandline_clip.py c:\pythonprimer\chapter12\data city_facilities.shp central_city_complan.shp c:\pythonprimer\chapter12\mydata out_clip.shp

Running Clip Routine...
Input workspace is: c:\pythonprimer\chapter12\Data
Input file is: city_facilities.shp
Clip file is: central_city_complan.shp
Output workspace is: c:\pythonprimer\chapter12\mydata
Output file is: c:\pythonprimer\chapter12\mydata\out_clip.shp
Completed Clip

c:\PythonPrimer\Chapter12>
```

The command line syntax begins with the `python` command and then the name of the script **commandline_clip.py** followed by each parameter separated by a space. The list of parameters will span multiple lines. All of the commands are typed at the command prompt at one time. Notice that the extra “backspace” is not used in the command line syntax.

NOTE: If spaces exist in the directory path, then the path string will need to include double quotes so that Python interprets the string as a single parameter. For example, the following path contains spaces which are bounded by double quotes.

```
"c:\gis data\pythonprimer\chapter12"
```

The code developer should note that if a Python script requires parameters that “get” values (e.g. the lines of code that use the `arcpy.GetParameterAsText (n)` routine, where *n* is the argument number) these parameters need to be included in the command line arguments. The table below summarizes the script parameters and the values typed at the command prompt for the **commandline_clip.py** script.

Python Script Variable	Python Script Parameter <code>arcpy.GetParameterAsText (n)</code>	Value typed at Command Prompt
<code>arcpy.env.workspace</code>	0	C:\pythonprimer\chapter12\data
<code>infile</code>	1	city_facilities.shp
<code>clipfile</code>	2	central_city_commplan.shp
<code>output_ws</code>	3	c:\pythonprimer\chapter12\mydata
<code>outfile</code>	4	out_clip.shp

Creating a Python Batch File

The batch file is simply a text file that contains the command line syntax for running a Python script similar to the above figure. The file name must end with the **.BAT** extension which the Windows operating system recognizes as an executable batch file.

NOTE: Any file on a Windows operating system with the **.BAT** extension is executable.

The batch **.BAT** file can be created by using a text editor (such as Notepad), writing the complete command line syntax, and then saving the file with the **.BAT** extension. It is recommended to use a generic text editor versus Word or a specific document writer, since these programs may add special characters that are invisible to the user (such as line returns, paragraph breaks, tabs, etc.).

NOTE: The Python command file and script parameters are written on a single line with the parameters separated by a space.

The following text represents the syntax for the **commandline_clip.py** script that is written in a text editor. The code is written without line returns (i.e. the code is written on a single line). The batch file itself has been named **CommandLine_Clip_BatchFile.bat**. This file can be found in the **Chapter12** folder. Any text editor can be used to open the file and see the specific syntax within the **.BAT** file. Note: Double clicking the **.BAT** file actually runs the file. Use **right-click—Open** to open the **.BAT** file.

```
REM This is a Python batch script file

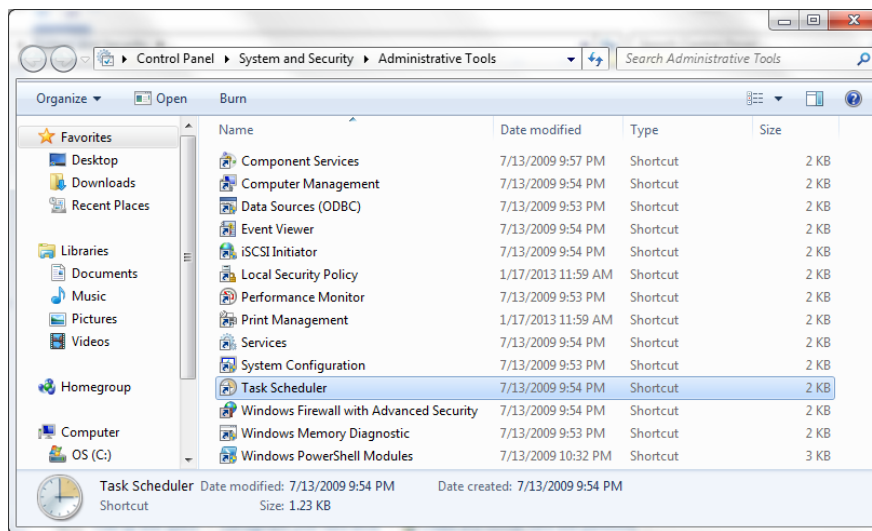
python commandline_clip.py c:\pythonprimer\chapter12\data
city_facilities.shp central_city_commplan.shp
c:\pythonprimer\chapter12\mydata out_clip.shp
```

In addition to double-clicking the **.BAT** file, the user can open the Command Prompt window and change directories (*CD*) to the location of the **.BAT** file and then type the name of the **.BAT** file at the command prompt to execute it. If any print messages exist within the script, they will print in the command line window.

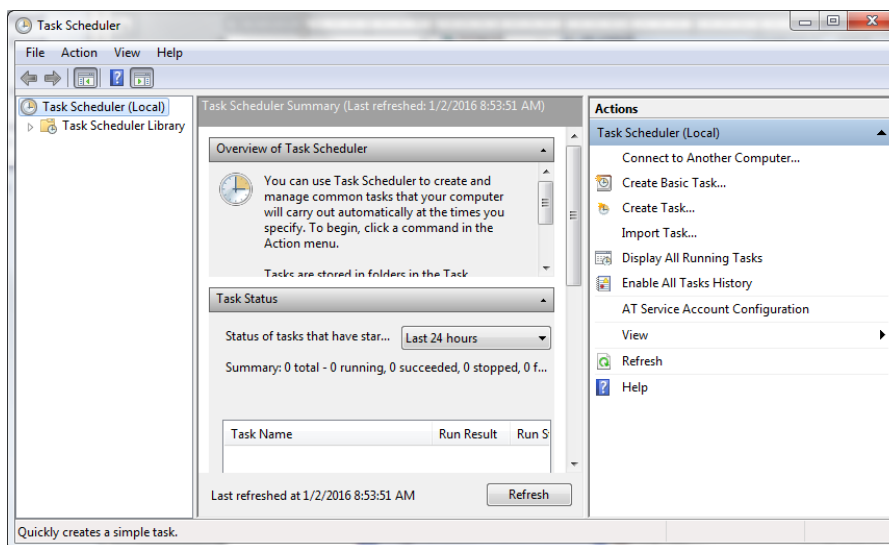
NOTE: The command line window will close automatically when the script successfully executes if the user double-clicks the **.BAT** file. If the user wants to keep track of the print messages as the script executes, the print statements can be written to a log file. See Chapter 8 for creating and using a log file.

Scheduling the Batch File to Automatically Run the Geoprocessing Script

The batch file developed above must be double-clicked in a Windows Explorer or typed in a command line prompt for the script to execute. To automatically run the script on a Windows machine, it must be scheduled using the Task Scheduler. The Task Scheduler can be found in **Start—All Programs—Accessories—System Tools—Task Scheduler** or **Control Panel—System and Security—Administrative Tools—Task Scheduler**. Other operating systems may have the Task Scheduler in a different location.



A task can be added and scheduled by clicking on **Create Task** from the **Actions** window of the **Task Scheduler**.

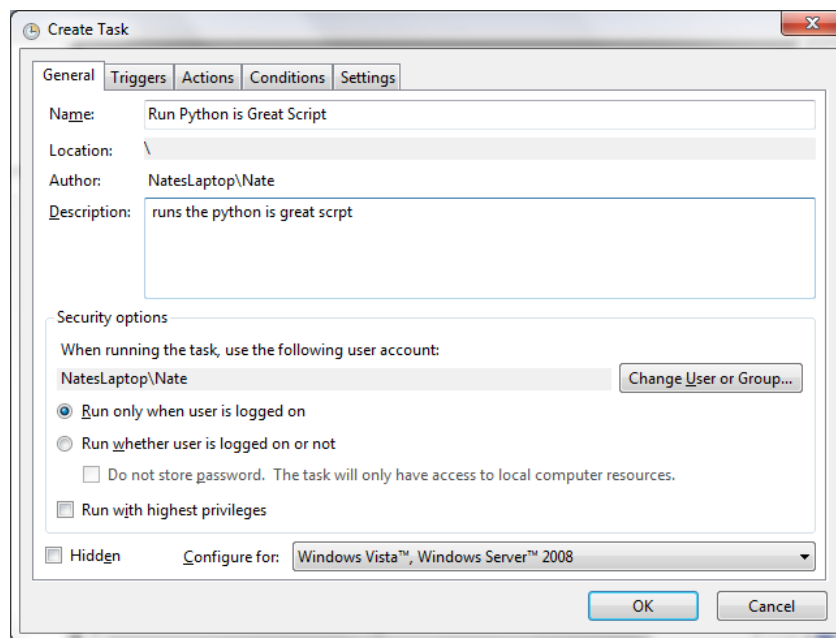


Fill in the following information for the **Create Task** tabs.

1. **General:** Create a name for the task. Description is optional.
2. **Triggers:**
 - a. Set the Begin task (such as **On a Schedule**).
 - b. Change settings to start the script to run (start day and time)
 - c. Select how often to repeat the task (hourly, daily, weekly, etc)
3. **Actions:**
 - a. Typically, create a new action as **Start a Program**.
 - b. Browse to the folder and choose the **.BAT** file to run.
 - c. Other options are not required.
4. **Conditions:** Typically, take default values.
5. **Settings:** Typically, take default values.

The following shows the information needed to schedule the **runpythonisgreat.bat** file.

Settings for the General Tab



Settings for the Triggers Tab

Edit Trigger

Begin the task: On a schedule

Settings

☒ One time Start: 1/ 2/2016 9:30:00 AM ☐ Synchronize across time zones

☐ Daily

☐ Weekly

☐ Monthly

Advanced settings

☐ Delay task for up to (random delay): 1 hour

☐ Repeat task every: 1 hour for a duration of: 1 day

☐ Stop all running tasks at end of repetition duration

☐ Stop task if it runs longer than: 3 days

☐ Expire: 1/ 2/2017 9:39:38 AM ☐ Synchronize across time zones

☒ Enabled

OK Cancel

Settings for the Actions Tab

Edit Action

You must specify what action this task will perform.

Action: Start a program

Settings

Program/script: C:\PythonPrimer\Chapter12\runpythonisgreat.bat Browse...

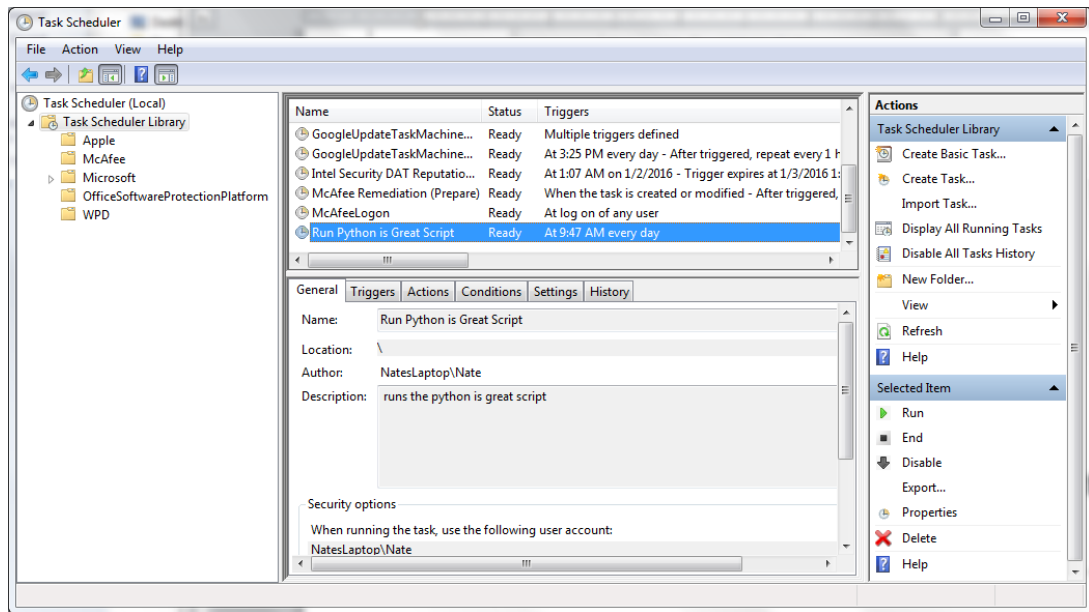
Add arguments (optional):

Start in (optional):

OK Cancel

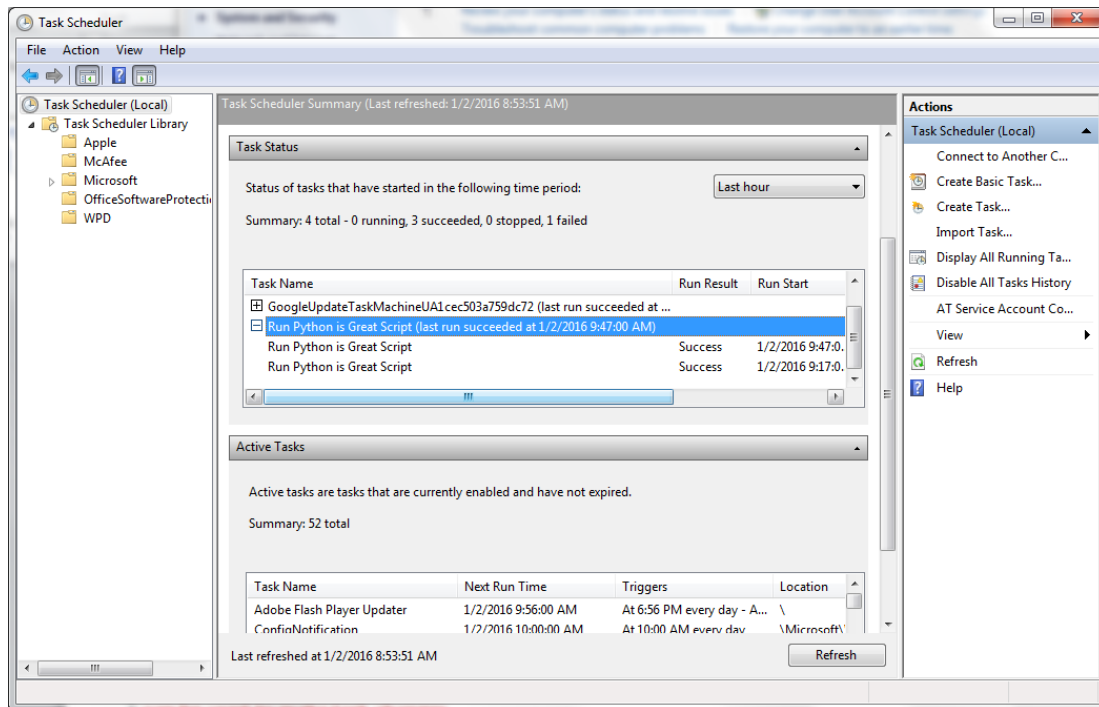
The rest of the default values are set for the **Conditions** and **Settings** tabs.

The new task can be seen in the list of scheduled tasks in the **Task Schedule Library**.



The task settings can be changed by left-clicking on the scheduled task and making the respective changes in each of the tabs. Alternatively, the Properties option on the selected task can be used to make task changes.

NOTE: Before a task is set to run, make sure to “Enable” All Task History. This can be changed by clicking on **Enable All Task History** in the upper right of the Task Scheduler. This will create a record in the History tab of the Scheduled Tasks Library as well as under the Task Scheduler view.

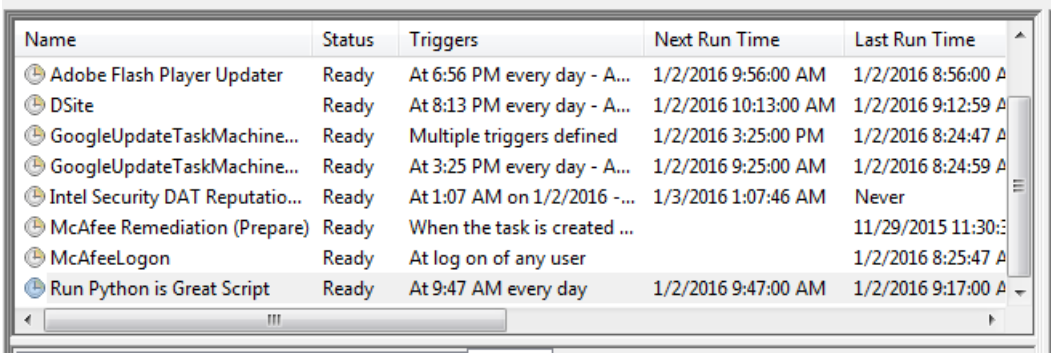


To properly schedule a task on a production server environment, a proper user name and password must be set. Normally, scheduled tasks are run on organizational servers that require proper login credentials. On a personal system the user can set up his/her respective set of credentials. The computer must have a password for the script to properly run or the scheduler set to run if the user is logged in. Not assigning a password will result in the scheduled task not being able to run and an Access Denied error will show when the user attempts to schedule the task without a password.

As an alternative, if an Access Denied message appears, the user can go into the Schedule Tasks Properties under the Task Tab and check the “Run only if logged in” button. This will alleviate the need to use a specific password. Also note that some settings may need to be changed if running the schedule task on a laptop (e.g. “Don’t start the script if the computer is running on batteries”, etc). For organizational systems it is recommended to use secure login credentials and that proper backup systems are in place.

In addition, when testing script automation processes it is good practice for an organization to have a “test environment” and a “production environment.” Test environments are used to test processes and methods and troubleshoot any pending issues with data, scheduling, and the processing of data. Only when a script has been thoroughly tested can it then be placed and scheduled on a “production environment;” one that includes a server used to perform real business functions and real (live) data, and includes the proper information security and backup system or server redundancy.

When a batch file is actually running the Status column will show “Running.” If the task runs successfully the “Last Result” column will show “0x0”. If the schedule task fails, the “Last Result” column will show “0x1” or some other similar value. The user should also review any log files that collect Python print statements to assist any trouble shooting efforts to remedy any scripting or potential network problems. Other system logs and computer administration may need to be contacted if required. If the scheduled task successfully executes, the Python script was automatically run. The scheduled task will continue to function properly unless changes to the process are required or problems with the machine or data server are incurred. If changes are made to the script and/or data changes, these should be tested on the test environment before deploying to the production environment.



Name	Status	Triggers	Next Run Time	Last Run Time
Adobe Flash Player Updater	Ready	At 6:56 PM every day - A...	1/2/2016 9:56:00 AM	1/2/2016 8:56:00 A
DSite	Ready	At 8:13 PM every day - A...	1/2/2016 10:13:00 AM	1/2/2016 9:12:59 A
GoogleUpdateTaskMachine...	Ready	Multiple triggers defined	1/2/2016 3:25:00 PM	1/2/2016 8:24:47 A
GoogleUpdateTaskMachine...	Ready	At 3:25 PM every day - A...	1/2/2016 9:25:00 AM	1/2/2016 8:24:59 A
Intel Security DAT Reputatio...	Ready	At 1:07 AM on 1/2/2016 -...	1/3/2016 1:07:46 AM	Never
McAfee Remediation (Prepare)	Ready	When the task is created ...		11/29/2015 11:30:3
McAfeeLogon	Ready	At log on of any user		1/2/2016 8:25:47 A
Run Python is Great Script	Ready	At 9:47 AM every day	1/2/2016 9:47:00 AM	1/2/2016 9:17:00 A

Summary

A Python script can be automatically run by using a Windows batch **.BAT** file and scheduling it in the Window Task Scheduler. By using these methods geoprocessing Python scripts can run after hours or during off-peak hours without the GIS staff or program developer to physically interact with them. Being able to automatically executing ArcGIS geoprocesses can free an analyst’s time to work on analytical, data management, cartography, or other GIS tasks. Automated geoprocessing tasks are often implemented to update relational databases or file and publication servers that serve out data, maps, web services, and other geospatial and tabular information.

The Chapter 12 demo walks through the steps to set up and schedule a geoprocessing **.BAT** file. The reader can use these methods to set up their own script automation.

Chapter 12 Demo Using a Batch File to Auto-run a Python Script

The following demonstrates creating, scheduling, and running a batch file for the **CommandLine_Clip.py** script. A Windows 7 (64-bit) machine is used in the demonstration. Other operating systems may have a slightly different interface for scheduling tasks. The script and data are located in the **\PythonPrimer\Chapter12** folder.

1. Review the existing **CommandLine_Clip.py** script. Notice that all of the parameters use the `GetParameterAsText (n)` routine where **n** is the parameter index value, starting at 0 for the first parameter. A number of print statements are written to a log file.
2. Open a text editor (such as Notepad) and add the following code (see the code after the table). Make sure the code is written on a single line with each parameter (i.e. the value in the third column in the table below) is separated by a space. Save the file as **CommandLine_Clip_BatchFile.bat**. The batch file includes the following parameters and the syntax is shown below.

Python Script Variable	Python Script Parameter <code>arcpy.GetParameterAsText (n)</code>	Value typed at Command Prompt
<code>arcpy.env.workspace</code>	0	<code>C:\pythonprimer\chapter12\data</code>
<code>infile</code>	1	<code>city_facilities.shp</code>
<code>clipfile</code>	2	<code>central_city_commplan.shp</code>
<code>output_ws</code>	3	<code>c:\pythonprimer\chapter12\mydata</code>
<code>outfile</code>	4	<code>out_clip.shp</code>

REM This is a Python batch script file

```
python commandline_clip.py c:\pythonprimer\chapter12\data
city_facilities.shp central_city_commplan.shp
c:\pythonprimer\chapter12\mydata out_clip.shp
```

“REM” (remark) is used to “comment” lines that are not processed in the **.BAT** file.

3. Test the batch file by opening Windows Explorer. Navigate to the **\PythonPrimer\Chapter12** folder and find the **.BAT** file. Double click on the **.BAT** file to run it. Make sure the script executes successfully. If errors occur, check the syntax of the **.BAT** file to make sure it is correct. Refer to the **CommandLine_Clip_BatchFile.bat** file that is provided, if needed. A successfully executed script will result in the Command Line window appearing and then disappearing.
4. Check the **Chapter12\MyData** folder to see that the **out_clip.shp** file was created with the current date and time stamp as well as a **log file** with the current date.
5. If the above process is successful, open the **Task Scheduler** program from **Start—All Programs—Accessories—System Tools—Task Scheduler** or **Control Panel—System and Security—Administrative Tools—Task Scheduler**.
6. Click **Create Task**. Set the General, Trigger, and Action tabs.

General: Create name for the task. Description is optional.

Triggers:

- a. Click **New** to set a new trigger.
- b. Set the **Begin the task** to **On a Schedule**.
- c. Change settings to start the script to run. Choose a day and time to run (e.g. several minutes from the current time today).
- d. Select how often to repeat the task (hourly, daily, weekly, etc). This can be set if desired. This is only a demo exercise, so the repeat time does not need to be set.

Actions:

- a. Click **New** to set a new action.
- b. Use **Start a Program**.
- c. Browse to the **Chapter12** folder and choose the **.BAT** file just created.

7. Check at a later time to see if the batch process ran successfully. This can be checked by reviewing the date and time stamp of the files in Windows Explorer or ArcCatalog, as well as, checking the log file.

Chapter 12 Questions

1. What are 3 different ways to run a Python script?
2. What is a primary benefit of scheduling a Python script?
3. To schedule a script for automatic processing, what kind of file is used in the scheduler?
4. What kind of information must this file include?
5. If the Task Scheduler is used on a server, what information is likely needed to successfully schedule the script?

Accessing Data, Demos, and Code

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

<http://pythonprimer.urbandalespatial.com/resources>

Username: PythonPrimer

Password: PP4AGIS!

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

References

Author's website – www.urbandalespatial.com

Jennings, Nathan. "Managing Street Sign Assets: An enterprise geospatial business systems integration solution." *ArcUser*TM, Winter 2009. Date Accessed: 11.09.2011

<http://www.Esri.com/news/arcuser/0109/streetsigns.html>

ArcGIS

ArcGIS Resource Center - <http://resources.arcgis.com/>

ArcGIS Web-based Help - <http://resources.arcgis.com/en/communities/>

ArcGIS Blog - <http://blogs.Esri.com>

ArcGIS Forums - <https://geonet.esri.com/community/developers/gis-developers/python>

Geoprocessing script examples and models -

<http://resources.arcgis.com/en/communities/python/>

Esri Training courses - <http://www.esri.com/training/main>

ArcUser - <http://www.Esri.com/news/arcuser>

ArcGIS Resource Center. Exception Code Snippet. Esri, 2011.

<http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/002z0000000q000000>.

Python

Python website – www.python.org

Lutz, Mark. Learning Python, 4th Ed. Beijing: O'Reilly Media, Inc., 2009.

Organizations

Esri – www.Esri.com

American River College GIS Program - <http://wserver.arc.losrios.edu/~earthscience/>

Sierra College GIS Program - <http://www.sierracollege.edu/academics/divisions/science-math/geography.php>

UC Davis Extension - <https://extension.ucdavis.edu/subject-areas/geographic-information-systems>

Del Mar College - http://www.delmar.edu/CIS_-_Geographical_Information_Systems.aspx

City of Sacramento GIS – www.cityofsacramento.org/gis

County of Sacramento GIS – www.sacgis.org

Cal Atlas – <http://atlas.ca.gov>

INDEX

A

addin_assistant, 83, 84, 99, 100

B

batch, 79, 119, 121, 122, 123, 126, 127, 128, 133, 135, 136

C

class, 54, 55, 56, 72, 88, 90, 96, 105
custom toolbox, 25, 26, 28, 29, 55, 63, 65

E

environment variable, 120
esriaddin, 91, 92, 96

G

GetParameterAsText, 25, 26, 41, 42, 43, 47, 63, 81,
126, 135

I

image processing, 71, 77, 78

P

parameter properties, 31, 32, 34, 37, 63, 65, 66, 81
Python Add-in, 83, 84, 85, 89, 90, 91, 92, 94, 96, 97, 99,
100, 105, 106, 107, 119
Python toolbox, 53, 55, 56, 57, 58, 59

S

Spatial Analyst, 71, 75, 76, 77

T

tasseled cap, 71, 75, 76, 77, 78