# Introduction

* Abstract

Geospatial data analysis is a key component of decision-making and planning for numerous applications.

Geographic Information Systems (GIS), such as ArcGIS® , provide rich analysis and mapping platforms.

Modern technology enables us to collect and store massive amounts of geospatial data.

The data formats vary widely and analysis requires numerous iterations.

These characteristics make computer programming essential for exploring this data.

Python is an approachable programming language for automating geospatial data analysis.

This chapter discusses the capabilities of scripting for geospatial data analysis, some characteristics of the Python programming language, and the online code and data resources for this book.

After downloading and setting up these resources locally, readers can walk through the step-by-step code example that follows.

Last, this chapter presents the organization of the remainder of the book.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Articulate in general terms, what scripting can do for GIS workflows.

• Explain why Python is selected for GIS programming.

• Install and locate the sample materials provided with the book.

• Contrast the view of compound GIS datasets in Windows Explorer and ArcCatalog™.

• Run code in the ArcGIS® Python Window.

Geographic data analysis involves processing multiple data samples.

The analysis may need to be repeated on multiple fields, files, and directories, repeated monthly or even daily, and it may need to be performed by multiple users.

Computer programming can be used to automate these repetitive tasks.

Scripting can increase productivity and facilitate sharing.

Some scriptable tasks involve common data management activities, such as, reformatting data, copying files for backups, and searching database content.

Scripts can also harness the tool sets provided by Geographic Information Systems (GIS) for processing geospatial data, i.e., geoprocessing.

This book focuses on the Python scripting language and geoprocessing with ArcGIS software.

Scripting offers two core capabilities that are needed in nearly any GIS work:

• Efficient batch processing.

• Automated fi le reading and writing.

Scripts can access or modify GIS datasets and their fields and records and perform analysis at any of these levels.

These automated workflows can also be embellished with GUIs and shared for reuse for additional economy of effort.

## Python and GIS

## Sample Data and Scripts

The examples and exercises in this book use sample data and scripts available for download from <http://www.springer.com/us/book/9783319183978>.

Click on the ‘Supplementary Files’ link to download ‘gispy.zip’.

Place it directly under the ‘C:\’ drive on your computer.

Uncompress the fi le by right-clicking and selecting ‘extract here’.

Once this is complete, the resources you need for this book should be inside the ‘C:\gispy’ directory.

Examples and exercises are designed to use data under the ‘gispy’ directory structured as shown in Figure1.1.

## GIS Data Formats

Several GIS data formats are used in this book, including compound data formats such as GRID rasters, geodatabases, and shapefiles.

In Windows Explorer, you can see the fi le components that make up these compound data formats.

In ArcCatalog™, which is designed for browsing GIS data formats, you see the software interpretation of these fi les with both geographic and tabular previews of the data.

This section looks at three examples (GRID rasters, Shapefiles, and Geodatabase) comparing the Windows Explorer data representations with the ArcCatalog ones.

We will also dis cuss two additional data types (dBASE and layer files) used in this book that consist of only one fi le each, but require some explanation.

### GRID

AGRID raster defines a geographic space with an array of equally sized cells arranged in columns and rows.

Unlike other raster formats, such as JPEG or PNG, the file name does have a file extension.

The fi le format consists of two directories, each of which contains multiple fi les.

One directory has the name of the raster and contains ‘.adf’ files which store information about extent, cell resolution, and so forth; the other directory, named ‘info’, contains ‘.dat’ and ‘.nit’ files which store file organization information.

Figure1.2 shows a GRID raster named ‘getty\_rast’ in Windows Explorer (left) and in ArcCatalog (right).

Windows Explorer, displays the two directories, ‘getty\_rast’ and ‘info’ that together define the raster named ‘getty\_rast’.

The ArcCatalog directory tree displays the same GRID raster as a single item with a grid-like icon.

### Shapefile

A shapefile (also called a stand-alone feature class), stores geographic features and their non-geographic attributes.

This is a popular format for storing GIS vector data.

Vector data stores features as sets of points and lines as opposed to rasters which store data in grid cells.

The vector features, consisting of geometric primitives (points, lines, or polygons), with associated data attributes stored in a set of supporting files.

Though it is referred to as a shapefile, it consists of three or more fi les, each with different file extensions.

The ‘.shp’ (shapefile), ‘.shx’ (header), and ‘.dbf’ (associated database) files are mandatory.

You may also have additional fi les such as ‘.prj’ (projection) and ‘.lyr’ (layer) files.

Figure1.5 shows the shapefile named ‘park’ in Windows Explorer (which lists multiple files) and ArcCatalog (which displays only a single file).

Shapefiles are often referred to with their ‘.shp’ extension in Python scripts.

### dBASE Files

One of the shapefile mandatory fi le types (‘.dbf’) can also occur as a stand-alone database file.

The ‘.dbf’ file format originated with a database management system named ‘dBASE’.

This format for storing tabular data is referred to as a dBASE file.

If a dBASE file appears in a directory with a shapefile by the same (base) name, it is associated with the shapefile.

If no shapefile in the directory has the same name, it is a stand-alone fi le.

### Layer Files

A ‘.lyr’ fi le can be used along with a shapefile to store visualization metadata.

Usually when a shapefile is viewed in ArcGIS, the symbology (the visual representation elements of features, such as color, outline, and so forth) is randomly assigned.

Each time it is previewed in ArcCatalog a polygon shapefile might be displayed with a different color, for example.

The symbology can be edited in ArcMap® and then a layer fi le can store these settings.

A layer file contains the specifications for the representation of a geographic dataset (a shapefile or raster dataset) and must be stored in same directory as the geographic data.

A common source of confusion is another use of the term ‘layer’.

Data that is added to a map is referred to as a layer (of data).

This is not referring to a fi le, but rather an attribute of the map, data which it displays.

### Geodatabase

Esri has three geodatabase formats: file, personal, and ArcSDE™.

A geodatabase stores a collection of GIS datasets.

Multiple formats of data (raster, vector, tabular, and so forth) can be stored together in a geodatabase.

Figure1.3 shows a file geodatabase , ‘regions.gdb’ in Windows Explorer and in ArcCatalog.

The left image shows that ‘region.gdb’ is the name of a directory and inside the directory is a set of fi les associated with each of the datasets (with extensions .freelist, .gdbindexes, .gdbtable, .gdbtablx, and so forth), only a few of which are shown in Figure1.3 .

The ArcCatalog view in Figure1.3 shows the five datasets (four vector and one raster) in this geodatabase.

The geodatabase has a silo-shaped icon.

Clicking the geodatabase name expands the tree to show the datasets stored in the geodatabase.

The dataset icons vary based on their formats: ‘fireStations’ is a point vector dataset, ‘landCover’ and ‘workzones’ are polygon vector datasets, ‘trail’ is a polyline dataset, and ‘tree’ is a raster dataset.

The vector format files are referred to as geodatabase feature classes, as opposed to shapefiles, which are stand-alone feature classes.

Both geodatabases feature classes and stand-alone feature classes store geographic features and their non-geographic attributes.

## An Introductory Example

The ArcGIS Buffer (Analysis) tool, creates polygon buffers around input geographic features (e.g., Figure1.4 ).

The buffer distance, the side of the input feature to buffer, the shape of the buffer, and so forth can be specified.

Buffer analysis has many applications, including highway noise pollution, cell phone tower coverage, and proximity of public parks, to name a few.

To get a feel for working with the sample data and scripts, use a line of Python code to call the Buffer tool and generate buffers around the input features with the following steps:

## Organization of This Book

This book focuses on automatically reading, writing, analyzing, and mapping geospatial data.

The reader will learn how to create Python scripts for repetitive processes and design flexible, reusable, portable, robust GIS processing tools.

The book uses Python to work with the ArcGIS arcpy™ package, as well as HTML, KML, and SQL.

Script tool user-interfaces, Python toolboxes, and the arcpy map ping module are also discussed.

Expositions are accompanied by interactive code snippets and full script examples.

Readers can run the interactive code examples while reading along.

The script examples are available as Python fi les in the chapter ‘sample scripts’ directory downloadable from the Springer Web site (as explained in Section1.2).

Each chapter begins with a set of learning objectives and concludes with a list of ‘key terms’ and a set of exercises.

The chapters are designed to be read sequentially.

General Python concepts are organized to build the Python skills needed for the ArcGIS scripting capabilities.

General Python topic discussions are presented with GIS examples.

Chapter 2 introduces the Python programming language and the software used to run Python scripts.

Chapters 3 and 4 discuss four core Python data types.

Chapters 5 and 6 cover ArcGIS tool help and calling tools with Python using the arcpy package.

Chapter 7 deals with getting input from the user.

Chapter 8 introduces programming control structures.

This provides a backdrop for the decision-making and looping syntax discussed in Chapters 9 and 10.

Python can use special arcpy functions to describe data (used for decision-making in Chapter 9) and list GIS data.

Batch geo processing is performed on lists of GIS data in Chapter 11.

Chapter 12 highlights some additional useful list manipulation techniques.

As scripts become more complex, debugging (Chapter 13 ) and handling errors (Chapter 14 ) become important skills.

Creating reusable code by defining functions (Chapter 15 ) and modules (Chapter 16 ) also becomes key.

Next, Chapter 17 dis cusses reading and writing data records using arcpy cursors.

In Chapter 18, another Python data structure, called a dictionary, is introduced.

Dictionaries can be useful during fi le reading and writing, which is discussed in Chapter 19.

Chapter 20 explains how to access online data, decompress fi les, and read, write, and parse markup languages.

Another code reuse technique, the user-defined class, is presented in Chapter 21.

Chapters 22 and 23 show how to create GUIs for fi le input or other GIS data types with Script Tools and Python toolboxes.

Finally, Chapter 24 uses the arcpy mapping module to perform mapping tasks with Python.

## Key Terms

Geoprocessing

GRID rasters

Vector data

Symbology

dBASE file

Layer file

Geodatabase

Feature class

Window Explorer vs. ArcCatalog

Buffer (Analysis) tool

ArcGIS Python Window

# Beginning Python

Abstract Before you can create GIS Python scripts, you need to know where to write and run the code and you need a familiarity with basic programming concepts.

If you’re unfamiliar with Python, it will be worthwhile to take some time to go over the basics presented in this chapter before commencing the next chapter.

This chapter discusses Python development software for Windows® operating systems, inter active mode and scripting, running scripts with arguments, and some fundamental characteristics of Python, including comments, keywords, indentation, variable usage and naming, traceback messages, dynamic typing and built-in modules, functions, constants, and exceptions.

* Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Test individual lines of code interactively in a code editor.

• Run Python scripts in a code editor.

• Differentiate between scripting and interactive code editor windows.

• Pass input to a script.

• Explain the advantages of using an integrated development environment, over a general purpose text editor.

• Match code text color with code components.

• Defi ne eight fundamental components of Python code.

## Where to Write Code

## How to Run Code in PythonWin and PyScripter

## How to Pass Input to a Script

## Python Components

### Comments

### Keywords

### Indentation

### Built-in Functions

### Variables, Assignment Statements, and Dynamic

### Variables Names and Tracebacks

Variable names can’t start with numbers nor contain spaces.

For names that are a combination of more than one word, underscores or capitalization can be used to break up the words.

Capitalizing the beginning of each word is known as camel case (the capital letters stick up like the humps on a camel).

This book uses a variation called lower camel case—all lower case for the first word and capitalization for the first letter of the rest.

For example, inputRasterData is lower camel case.

### Built-in Constants and Exceptions

### Standard (Built-in) Modules

## Key Terms

Integrated development environment (IDE)

Python script

PythonWin

PythonWin Interactive Window

PythonWin script window

PyScripter

Python prompt (>>> )

Python interpreter

Window focus

Script arguments

Context highlighting

Code comments

Hash sign

Python keywords

Variables

Assignment statement

Dynamic typing

Block structure, block of code

Dedented

Tracebacks

Built-in functions, constants, and exceptions

Function arguments

Function signatures

Function parameters

Exceptions thrown

True ,False ,None

NameError ,SyntaxError , and TypeError

Module

Standard modules math ,sys ,os , shutil

## Exercises

# Basic Data Types: Numbers and Strings

* Abstract

All Python objects having a data type.

Built-in Python data types, such as integers, floating point values, and strings are the building blocks for GIS scripts.

This chapter uses GIS examples to discuss Python numeric data types, mathematical operators, string data types, and string operations and methods.

* Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Perform mathematical operations on numeric data types.

• Differentiate between integer and floating point number division.

• Determine the data type of a variable.

• Index into, slice, and concatenate strings.

• Find the length of a string and check if a substring is contained in a string.

• Replace substrings, modify text case in strings, split strings, and join items into a single string.

• Differentiate between string variables and string literals.

• Locate online help for the specialized functions associated with strings.

• Create strings that represent the location of data.

• Format strings and numbers for printing.

## Numbers

## What Is a String?

By default, PythonWin uses olive-green text to display string literals.

A string literal is a set of characters surrounded by quotation marks.

A variable assigned a string literal value is called a string variable.

The difference between these two terms is important, but both of these items are sometimes referred to simply as ‘strings’.

The characters inside the quotes in the string literal are interpreted literally.

In GIS scripts, we use string literals for things like the name of a workspace, input file, or output file.

Meanwhile, the characters that make up a string variable are simply a name for the program to use to represent its value.

Variable names should not be surrounded by quotes.

The operations and methods described in the next section apply to both string literals and string variables.

Both are objects of data type str (for string).

## String Operations

### Find the Length of Strings

### Indexing into Strings

### Slice Strings

### Concatenate Strings

### Check for Substring Membership

Thein keyword enables you to check if a string contains a substring.

Python documentation refers to this as checking for ‘membership’ in a string.

Suppose you want to check if a fi le is a buffer output or not, and you have named each buffer output file so that the name contains the string buff.

## More Things with Strings (a.k.a. String Methods)

Along with the operations described above, processing GIS data often requires additional string manipulation.

For example, you may need to replace the special characters in a field name, you may need to change a file name to all lower case, or you may need to check the ending of a file name.

For operations of this sort, Python has built-in string functions called string methods.

String methods are functions associated particularly with strings; they perform actions on strings.

Calling a string method has a similar format to calling a built-in function, but in calling a string method, you also have to specify the string object—using dot notation.

Dot notation for calling a method uses both the object name and the method name, separated by a dot.

The general format for dot notation looks like this:

The examples here demonstrated a few of the string methods.

In fact there are many more, including

capitalize, center, count, decode, encode, endswith, expandtabs, find, index, isalnum, isalpha, isdigit, islower, isspace, istitle, isupper, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust, rpartition, rsplit, restrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, and zfill,

Help documentation and a comprehensive list of string methods is available online (search for ‘Python String Methods’).

String method names are often intuitive.

Testing them in the Interactive Window helps to clarify their functionality.

## File Paths and Raw Strings

When dealing with file paths, you will encounter strings literals containing escape sequences.

Escape sequences are sequences of characters that have special meaning.

In string literals, the backslash (\) is used as an escape character to encode special characters.

The backslash acts as a line continuation character when placed at the end of a line in a string literal as described in Section3.2.

But when a backs lash is followed immediately by a character in a string literal, the backslash along with the character that follows it are called an escape sequence and the backslash is interpreted as a signal that the next character is to be given a special interpretation.

For example, the string literals '\n' and '\t' are escape sequences that encode ‘new line’ and ‘tab’.

New line and tab characters are used to control the space around the visible characters.

They are referred to as whitespace characters, because the characters themselves are not visible when printed.

## Unicode Strings

When you start using ArcGIS functionality in Chapter 6 , you will begin to see a lowercase u preceding strings that are returned by GIS methods.

The u stands for unicode string.

A unicode string is a specially encoded string that can represent thousands of characters, so that non-English characters, such as the Hindi alphabet can be represented.

A unicode string is created by prepending au to a string literal,

as shown here:

## Printing Strings and Numbers

The built-inprint function is used frequently in scripting, so we’ll show a few examples of how it can be used.

As mentioned earlier, the print function does not use parentheses around the arguments (though this changes in Python 3.0 in which the parentheses become required).

The arguments are the expressions to be printed.

We often want to print multiple expressions within the same print statement and these expressions need to be linked together so that the print statement uses them all.

Here we demonstrate three approaches to linking expressions to be printed:

Commas.

When commas are placed between variables in a print expression, the variable values are printed separated by a space.

The print statement inserts a space where each comma occurs between printed items:

## Key Terms

int data type

float data type

Integer division

str data type

String literal

String variable

Line continuation

Zero-based indexing

Built-in len function

Slicing

Concatenating

Casting

Thein keyword

Dot notation

Objects

Methods

Context menus

Whitespace characters

Escape sequences

Raw strings

Unicode strings

String formatting

## Exercises

# Basic Data Types: Lists and Tuples

* Abstract

GIS scripting frequently involves manipulating collections of items, such as files or data records.

For example, you might have multiple tabular data fi les with special characters in the fields preventing direct import into ArcGIS.

A script to replace the characters can use lists to hold the file names and field names.

Then the files and fields can be batch processed.

Built-in Python data types, such as lists, are useful for solving this kind of GIS problem.

This chapter presents the Python list data type, list operations and methods, the range function, mutability, and tuples.

The chapter concludes with a debugging walk-through to demonstrate syntax checking and tracebacks.

* Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Create Python lists.

• Index into, slice, and concatenate lists.

• Find the length of a list and check if an item is in a list.

• Append items to a list.

• Locate online help for the list methods.

• Create a list of numbers automatically.

• Differentiate between in-place methods and methods that return a value.

• Create and index tuples.

• Check script syntax.

• Interpret traceback error messages.

## Lists

A Pythonlist is a data type that holds a collection of items.

The items in a list are surrounded by square brackets and separated by commas.

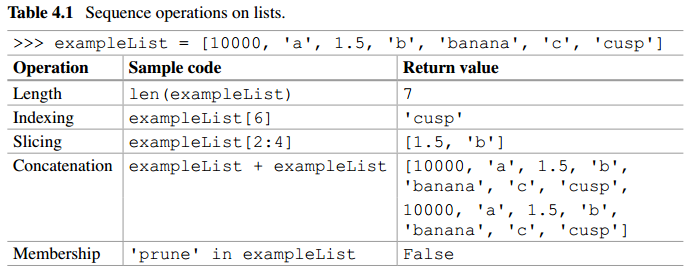
The syntax for a list assignment statement looks like this:

### Sequence Operations on Lists

Lists, like strings are one of the sequence data types in Python.

The sequence operations discussed in the context of string data types also apply to lists; the length of a list can be found, lists can be indexed, sliced, and concatenated, and you can check if an item is a member of a list.

Table4.1 shows sample code and output for each of these operations.



### List Methods

Also like strings, list objects have a specific set of methods associated with them including append ,extend ,insert ,remove ,pop ,index ,count ,sort , and reverse .

For a complete description of each, search online for Guido Van Rossum’s Python Tutorial for the ‘More on lists’ section.

Like string methods, list methods use the dot notation:

### The Built-in range Function

The built-in range function is a convenient way to generate numeric lists, which can be useful for batch processing tasks.

The range function takes one to three numeric arguments and returns a list of numbers.

If you pass in one numeric argument, n, it returns a Python list containing the integers 0 through n-1 , as in the following example:

### Copying a List

As we’ve just discussed, in-place methods like reverse and sort alter the order of the list.

For example, here the reverse method reverses the order of the fire IDs list:

## Tuples

## Syntax Check and Tracebacks

## Key Terms

List data type in-place method

Mutable vs. immutable

Shallow copy vs. deep copy

Built-in list function

Sequence operations (length, indexing, concatenation, slicing, membership)

Range built-in function

List methods

tuple data type

## Exercises

# ArcGIS and Python

* Abstract

ArcGIS provides a palette of sophisticated tools for processing and analyzing geographic data.

There are several ways in which these tools can be used.

For example, ArcToolbox tools can be run from ArcToolbox by clicking on the tool and filling out a form to specify the parameters; they can be run from ModelBuilder Models, and they can be run from Python scripts using the arcpy package.

This chapter discusses ArcToolbox, ModelBuilder, ArcCatalog, Python’s ArcGIS capabilities, the arcpy package, arcpy functions, and environment settings.

* Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Describe the ArcToolbox hierarchy.

• Search for tools in ArcCatalog.

• Locate tool help on ArcGIS Resources online.

• Export a script from a visual workflow model.

• Modify and run exported scripts.

• Preview geoprocessing output.

• Release locks on data.

• Explain, in general terms, the capabilities of thearcpy package.

• Define the Python terms module and package .

• Set geoprocessing environment variables.

## ArcToolbox

## ArcGIS Python Resources

The ‘ArcGIS Resources’ site (resources.arcgis.com) is the foremost reference you will need for working with Python in ArcGIS.

The site provides full documentation for ArcGIS Python functionality.

The ‘Search ArcGIS Resources’ box is indispensable for working with Python in ArcGIS.

Use the search box to get a list of pages matching your query and use the search filters to narrow the search (Figure5.4 ).

For example, enter ‘buffer’ in the search box.

This returns thousands of results including blogs, bug reports, web mapping help, and so forth.

Narrow the search by using the ‘Help’ and ‘Desktop’ filters as shown Figure5.5.

Notice that the results are different from the ArcCatalog search results for the same term (Figure5.3 ); the ArcGIS Desktop help is organized differently than the online help.

The online help provides the most current, comprehensive documentation.

A set of descriptive identifiers is provided for each link in the online help.

The Buffer (Analysis) link has the identifiers ‘Tool Reference’ and ‘ANALYSIS’ (Figure5.6 ).

The last identifier shows the date the content was last modified.

Each ArcGIS tool has a ‘Tool Reference’ page that corresponds to the built-in help for that tool.

Chapter 6 discusses components in ‘Tool Reference’ pages.

This site is referred to as the ‘ArcGIS Resources’ site in this book.

Key search terms will be provided to direct you to specific help topics within this site.

## Exporting Models

Python can call almost all the tools in ArcToolbox and, in this way, repetitive processes can be automated.

Before we begin writing scripts from scratch, we’ll start with an example automatically generated by the ArcGIS ModelBuilder application.

ModelBuilder is an application built into ArcCatalog (and ArcMap) that allows users to create a workflow visualization, called a model.

Models not only visualize the workflow, but can also be can be run to execute the workflow.

ArcGIS Toolbox tools can also be run via ModelBuilder; Tools can be dragged into the model panel and connected to create the workflow.

When a model runs, it executes tools and the underlying code statements that correspond to pieces of the model.

The underlying code can be exported to a Python script and we can compare the workflow visualization with the code.

Follow steps 1–3 to create and export a simple model.

## Working with GIS Data

Each time you run a Python script that generates geoprocessing output, you will want to check the results.

Esri geographic data features and the tables associated with them can be viewed in the ArcCatalog ‘Preview’ tab.

Browse to a shapefi le in the ArcCatalog ‘Catalog Tree’ and select the ‘Preview’ tab.

Select ‘Geography’ preview from the bar at the bottom of the Preview pane to view the geographic feaures.

Then select ‘Table’ to see the associated attribute table.

Figures5.9 and5.10 show the geography and table views of ‘park.shp’.

Only seven rows of the table are shown in Figure5.10.

There are 426 rows in total, one data record for each polygon.

When the data is being viewed in ArcCatalog, it is locked, so that other programs can’t modify it simultaneously.

A fi le with an ‘sr.lock’ extension appears in Windows Explorer when the data is locked.

For example, the fi le could be named something like ‘park.shp.HAL.5532.5620.sr.lock’ on a computer named ‘Hal’.

When you perform processing on the fi le in a Python script, you need to make sure that the data is not locked.

To unlock the data after previewing it in ArcCatalog, select the parent workspace (‘C:/gispy/data/ch05’ in Figures5.9 and5.10 ) and refresh ArcCatalog (press F5).

Selecting another file within the same workspace and refreshing the Catalog Tree will not release the lock; the parent workspace must be refreshed.

To see the geographic view of more than one fi le at a time, you need to use ArcMap.

To view your data in ArcMap, the simplest approach is to browse to the data in the ArcCatalog tree embedded in ArcMap and drag/drop it onto a blank map.

As long as ArcMap is still running, data used in this way will be locked.

Even if the map document is closed, the locks may not be released until the program itself is exited.

## ArcGIS + Python = arcpy

Now that you’re familiar with ArcToolbox, ModelBuilder, and ArcCatalog data previews, it is time to introduce the arcpy Package.

Geoprocessing scripts begin by importing arcpy (e.g., Figure5.8 ); The arcpy package is Python’s means for accessing ArcGIS from Python.

To use ArcGIS functionality in Python, a script needs to import arcpy.

The terms import, package, and arcpy are explained here:

* The keyword import is used to reference a module or a package.

This provides access to functionality beyond the built-in Python functionality.

As discussed in Chapter 2, the term following the import keyword is a module or a package.

* Recall that a module is a single Python script (‘.py’ file) containing tightly related definitions and statements.

A package is a special way of structuring a set of related Python modules.

A package is a directory containing modules and sometimes subpackages, which also contain modules.

A module named ‘\_\_init \_\_ .py’ tells Python that the directory contains a package.

Modules structured within a package and items in those modules can be accessed using the dot notation.

* arcpy is a package installed with ArcGIS.arcpy can also be thought of as a Python object that has numerous methods, including geoprocessing tools.

Search under the ArcGIS install and you will find an arcpy directory.

This is the package being imported.arcpy provides a variety of functionality that we’ll be using throughout the remainder of the book.

Table5.1 lists the main arcpy topics covered in this book.

## arcpy Functions

The arcpy functions (top left box in Figure5.11) provide support for geoprocessing workflows.

For example, functions can be used to list datasets, retrieve a data set’s properties, check for existence of data, validate a table name before adding it to a geodatabase, or perform many other useful scripting tasks.

The syntax for calling functions in the arcpy package uses the dot notation with arcpy before the dot and the function name after the dot:

## Environment Settings

Each tool has settings it uses to execute an operation, such as a tolerance or output location.

The environment settings are conditions that can be applied to all tools within an application (e.g., all operations in ArcCatalog can be limited to an extent set in the environment settings).

ArcGIS has default values for these settings and users can modify the values via a dialog box in ArcGIS (Geoprocessing > Environments launches the dialog box shown in Figure5.12).

Python commands can also be used to get or set the values of these settings.arcpy has anenv class, a special structure for holding related properties.

The env properties control the environment settings.

env belongs toarcpy and the properties belong to the env class, so the property names have two dots, one after arcpy and one after env .

The format for setting these properties is:

## Key Terms

ModelBuilder models

import keyword

arcpy package

Python package

Python module

Environment settings

ArcToolbox

ArcCatalog tool search

ArcGIS Resources site

ModelBuilder

Model parameter

ArcCatalog geography and table previews

Data locks

## Exercises

# Calling Tools with Arcpy

* Abstract

ArcGIS offers a large array of sophisticated geoprocessing tools which can be accessed from Python using the arcpy package.

Chapter 5 introduced the arcpy package and used it to set environment variables.

This chapter focuses on how to call ArcGIS geoprocessing tools with arcpy and how to use the tool help documentation and get sample code.

We explain the general format for calling tools and how to handle linear unit parameters, multi-value parameters, optional parameters, and return values.

We also discuss tools which require specialized approaches, including Calculate Field, Raster Calculator, Make XY Event Layer tools, and tools in the Spatial Analyst toolbox.

Then we show how to call tools in a sequential work flow and how to call custom tools.

* Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Invoke geoprocessing tools with Python.

• Interpret ArcGIS Resources scripting help for geoprocessing tools.

• Copy code snippets from hand-run tools.

• Calculate field values with Python expressions.

• Format linear units, multi-value input, and optional input for GIS tools.

• Consume values returned by GIS tools.

• Call Spatial Analyst tools and perform map algebra.

• Save temporary raster and feature layer data.

• Use output from a GIS tool as input to another GIS tool.

• Import custom toolboxes and call custom tools.

## Calling Tools

ArcGIS users know how to run tools from ArcToolbox.

ArcToolbox tools, like other arcpy functions, use dot notation with arcpy before the dot.

The tool name and toolbox alias separated by an underscore are placed after the dot.

All ArcGIS tools with the exception of tools in the Spatial Analyst Toolbox can be called with the following syntax:

## Help Resources

The general format is consistent for most tools, but each tool has unique parameter requirements.

To call tools from scripts, you need to know how to interpret the documentation and find code samples.

The next sections walk through the tool help documentation and explain how to generate custom code samples.

### Tool Help

As mentioned in Chapter 5, the ArcGIS Resources site hosts a ‘Tool reference’ page for each tool.

Search on the name of the tool and filter by ‘Help’ and ‘Desktop’ to find a tool’s page.

Tool reference pages begin with an explanation of how the tool works (usually consisting of Summary, Illustration, and Usage sections).

The next sections, the ‘Syntax’ and ‘Code sample’ sections are guides for Python scripting.

As an example, search for the Buffer (Analysis) tool.

Locate the ‘Syntax’ section for the Buffer tool.

At the very beginning of this section, tool syntax is represented by a code signature.

This is a blueprint for how to call the tool via Python.

The code signature for the Buffer (Analysis) tool looks like this:

### Code Snippets

Code snippet captures provide an additional source for code samples.

ArcGIS enables you to run a tool ‘by hand’ using the dialog box and then copy the Python code it creates from that tool run.

Here are the steps to follow:

1. Locate the tool in ArcToolbox.
2. Execute the tool using its dialog box.
3. Open the ‘Results’ window (Geoprocessing menu > Results).
4. In the Results window, right-click on the tool name in the results listed there and select ‘Copy as Python snippet’.
5. Paste the snippet into PythonWin and examine the syntax.

This technique is particularly useful when the structure of the required input is complex.

For example, the Weighted Overlay (Spatial Analyst) tool requires an arcpy weighted overlay table parameter that can be complicated to compose by hand.

Running the tool with your desired input and copying the snippet will provide an easier starting point.

## Tool Parameters

Many geoprocessing tools require a linear unit parameter (e.g., a buffer distance for buffer analysis, an aggregation distance for aggregation tools, a simplification tolerance for simplification tools, and so forth).

Other tools require a code expression parameter (e.g., an expression to calculate a value for the field calculation tool).

Still other tools accept one or multiple values for a single parameter (e.g., a single file or a list of files).

Meanwhile, most tools have both required and optional parameters.

The next sections discuss how Python handles linear unit, code expression, multi value, and optional parameters.

### Linear Units

A linear unit is an Esri data type for Euclidean distances.

When we use a tool’s GUI interface, there’s a text box and a combo box for typing a number and selecting a unit of measure (Figure6.2 ).

In Python, linear units are specified as a string, with a number and a unit of measure, separated by a space (e.g.,'5 miles' ).

A list of recognized units of measure is given in Table6.2.

Singular and plural forms of these units are both accepted.

Also, linear unit names are not case sensitive.

If only a numeric value is specified and a distance unit is not specified (e.g., '5' ), the tool uses the units of the input feature, unless the Output Coordinate System environment property has been set.

### Python Expressions as Input

Some tools, such as the Calculate Field and Calculate Value (Data Management) tools, take parameters which are themselves Python expressions.

The expression can be very simple.

For example, the following code sets every entry in the 'result' field of the ‘data1.shp’ shapefile to 5:

The last parameter in this Calculate Field call indicates the type of the expression (The choices are 'PYTHON' , 'PYTHON\_9.3' , or 'VB' . The latter, which stands for Visual Basic, is the default).

The Calculate Field expressions can also use the values in other fields and combine them with mathematical operators.

In these expressions, field names need to be surrounded by exclamation points.

The following code calculates the 'result' field again using an expression which multiplies the value in the 'measure' field by two and subtracts the value in the 'cover age' field:

Note that the expression must be compatible with the field type.

The ‘result’ field is a ‘float’ type field, so mathematical values can be used.

Feature class field calculation expressions can also use the ‘shape’ field (Only feature classes have a ‘shape’ field).

The shape field contains a set of arcpy Geometry objects (listed on the arcpy cheat sheet under ‘Other objects’).Geometry objects have properties such as ‘area’ and ‘length’.

These properties can be used with dot notation on the ‘shape’ field in the expressions, still surrounded by exclamation points.

The following code uses the 'area' geometry property to calculate a field named 'PolyArea' for the 'special\_regions.shp' polygon shapefile:

### Multivalue Inputs

Some tools accept multiple values as input for a single parameter.

If this is the case, a Python list appears behind the variable name in the ‘Parameter’ column.

For example, the Merge (Data Management) tool takes a list of input datasets.

The table entry for this parameter is shown in Figure6.3.

Multivalue input tools are usually ones that combine the input in some manner (e.g., merge, intersect, or union the data).

A Python list may be the simplest way to input these values, especially if you already have the input in a list.

However, there are two other ways to provide input to these tools.

They also accept a semicolon delimited string or a ValueTable object.

In the following code a Python list is used to specify the three input files to merge:

>>> inputFiles = ['park.shp', 'special\_regions.shp', 'workzones.shp']

>>> arcpy.Merge\_management(inputFiles, 'mergedData.shp')

In some cases, you may have values in a multivalue string, a string where the values are separated (or delimited) by semicolons.

Multivalue parameters can be specified in this way too.

Here the three inputs are given in a multivalue string:

>>> inputFiles = 'park.shp;special\_regions.shp;workzones.shp'

>>> arcpy.Merge\_management(inputFiles, 'mergedData2.shp')

A ValueTable is an arcpy object for storing rows and columns of information.

To use it, you create aValueTable object using the ValueTable function.

This function returns aValueTable object, which has methods and properties.

One of the ValueTable methods is addRow.

This example shows how to use the value table to merge three files:

>>> vt = arcpy.ValueTable()

>>> vt.addRow('park.shp')

>>> vt.addRow('special\_regions.shp')

>>> vt.addRow('workzones.shp')

>>> arcpy.Merge\_management(vt, 'mergedData3.shp')

The merge example is a one-dimensional example.

That is, it only has one item in each row.

The true advantage of the ValueTable object is for dealing with higher dimensions of data.

The ValueTable approach provides a convenient way to organize data when the input is a list of lists.

As an example, the ‘in\_features’ argument for the Intersect tool can be a list of input fi le names or alternatively, it can be a list of fi le names and priority rankings, as in the following example:

>>> inputFiles = [['park.shp', 2], ['special\_regions.shp', 2], ['workzones.shp',1]]

>>> arcpy.Intersect\_analysis(inputFiles, 'intersectData.shp')

Instead of using a list of lists, you could use a ValueTable for the intersection example as follows:

>>> vt = arcpy.ValueTable()

>>> vt.addRow('park.shp 2')

>>> vt.addRow('special\_regions.shp 2')

>>> vt.addRow('workzones.shp 1')

>>> arcpy.Intersect\_analysis(vt, 'intersectData.shp')

### Optional Parameters

Tool parameters must be used in the order they are listed in the tool code signature.

The required tool parameters always come at the beginning of the list.

Optional parameters can be omitted or a number sign (' #' ) can be used as a place holder.

In either of these cases, the default value is used for that parameter.

For example, the Polygon Neighbors (Analysis) tool, which returns a table of statistics about neighboring polygons has two required parameters and six optional parameters, as shown in the code signature:

PolygonNeighbors\_analysis (in\_features, out\_table, {in\_ fields},

{area\_overlap}, {both\_sides}, {cluster\_tolerance}, {out\_linear\_units},

{out\_area\_units})

In the following example, the Polygon Neighbors tool is called two different ways, but these lines of code are two equivalent ways of using the default values for the six optional parameters in the tool:

>>> arcpy.env.workspace = 'C:/gispy/data/ch06/'

>>> arcpy.env.overwriteOutput = True

>>># Use default values for the last 6 args.

>>> arcpy.PolygonNeighbors\_analysis('park.shp', 'PN.dbf')

>>># Another way to use default values for the last 6 args.

>>> arcpy.PolygonNeighbors\_analysis('park.shp', 'PN.dbf', '#', '#','#', '#','#', '#')

If you want to set some but not all optional parameters, you must use number signs as place holders for interior optional arguments.

In the following example, no place holder is needed, because we’re using the first optional argument and we can simply omit the last five parameters:

>>># Use default values for the last 5 parameters.

>>> arcpy.PolygonNeighbors\_analysis('park.shp', 'PN.dbf', 'COVER')

However, if we want to set the ‘area\_overlap’ parameter, but use the default value for the ‘in\_fi elds’ parameter, the place holder is needed.

The ‘in\_fi elds’ parameter precedes ‘area\_overlap’ in the parameters list, so if we failed to use ‘#’ as a place holder, the tool would assume ‘AREA\_OVERLAP’ was a fi eld name.

>>># Use default value for in\_fields,but set the value for area\_overlap.

>>> arcpy.PolygonNeighbors\_analysis('park.shp','PN.dbf','#','AREA\_OVERLAP')

## Return Values and Result Objects

When a geoprocessing tool is run, the tool returns anarcpy Result object.

The Result object contains information about the tool run, such as input values for the parameters and whether or not it was successful.

It also contains a list of one or more output values.

The returned object can be stored in a variable by using an assignment statement:

## Spatial Analyst Toolbox

The spatial analyst toolbox contains a variety of tools for calculating spatial measures such as density, distance, and neighborhood properties as well as map algebra, logical, and mathematics tools, extraction, generalization, and interpolation tools, and other specialized tools for hydrology, groundwater, and solar calculations, to name just a few.

This section contrasts the Python approach for the Spatial Analyst toolbox with the approach for other toolboxes.

Specifically, this section discusses the Python format for calling these tools, the import statements in the tool help, and access to raster calculation.

### Calling Spatial Analyst tools

The syntax for calling Spatial Analyst tools is slightly different from other tools.

Double dot notation must be used instead of an underscore to specify the toolbox.

arcpy.toolboxAlias.toolName(arg1, arg2, arg3,...)

There’s another difference in the way Spatial Analyst tools work.

Instead of returning a Result object, Spatial Analyst tools return Raster objects.

The Raster object temporarily holds the output raster in memory.

Unless explicitly saved, it will be deleted when the IDE session is closed.

To store the raster permanently, we have to save the raster in a separate step.

The following code assigns the environment settings, checks out the Spatial Analyst extension and then calls the Square Root tool on a raster named 'getty\_rast' .

The Raster object being returned is assigned to a variable named outputRast.

>>> import arcpy

>>> arcpy.env.workspace = 'C:/gispy/data/ch06/'

>>> arcpy.env.overwriteOutput = True

>>> inRast = 'getty\_rast'

>>> arcpy.CheckOutExtension('Spatial') u'CheckedOut'

>>> outputRast = arcpy.sa.SquareRoot(inRast)

The next step is to save the output raster.

The Raster objects have a save method which takes one optional parameter, a name.

When the Square Root tool was run, it automatically generated a name.

If no parameter is used, the raster is saved by that name.

The following line of code saves the output raster with the default name:

>>> outputRast.save()

>>> outputRast

### Importing spatial analyst

The ArcGIS Resources Spatial Analyst tool reference pages sometimes use a variation of the import statement in the code samples as in the following code:

>>>from arcpy.sa import \*

This statement creates a direct reference to the Spatial Analyst tool; It provides a shortcut so Spatial Analyst tools can be called without prependingarcpy.sa .

For example, it allows you to replace this statement:

>>> outputRast = arcpy.sa.SquareRoot(inRast)

with this statement:

>>> outputRast = SquareRoot(inRast)

This type of import saves some typing, but it can lead to confusion.

For example, the Spatial Analyst has an Int tool which converts each cell raster value to an integer; At the same time, Python has a built-in integer function named int.

Using the special import shortcut could lead to mystifying code such as the following:

>>> outputRast = Int(inRast)

>>> outputNum = int(myNum)

These statements look almost identical, but one takes a raster and returns a Raster object; The other takes a number and returns a number.

Using the standard import and full tool call (e.g., arcpy.sa.Int ) avoids ambiguities.

### Raster Calculator

Python can call any tool in ArcToolbox with the exception of the Spatial Analyst Raster Calculator tool.

The Raster Calculator performs what is known as ‘map algebra’; it performs mathematical operations on each cell of a raster grid.

For example, multiplying a raster by two using the Raster Calculator doubles the value stored in each cell.

The Raster Calculator tool is not designed to be called from a stand-alone script.

Instead, the script can call tools from the Spatial Analyst Math Toolset.

For example, multiplying the cells in ‘getty\_rast’ by 5 and subtracting 2 from each cell, in Raster Calculator would look like this: 5\*'getty\_rast'-2.

The values in ‘getty\_rast’ are 1 and 2, so the resulting values are 3 and 8.

To do this in a Python script, you could use two tool calls.

Output from the first statement would provide input for the second as shown in Example6.2 .

Since the initial values are 1 and 2, outRast1 has the values 5 and 10.

These values are used in the Minus tool call, so the values in outRast2 are 3 and 8.

The second to last line of code saves this raster.

The last line of code deletes the Raster objects using the del keyword.

This avoids locking issues.

Another way to handle raster calculations is to create arcpy Raster objects using the Raster method and then construct expressions using mathematical operators in combination with the Raster objects.

The following code is equivalent to the code used in Example 6.2 , producing values inequationRast2 identical to those inequationRast :

>>> rastObj = arcpy.Raster('dataRast')

>>> outRast = 5\*rastObj - 2

>>> outRast.save('equationRast2')

>>> deloutRast

One advantage of this approach is that pairwise mathematical operators (\* ,/ ,+ , - , etc.) can be used more than once, so that more than one pair of items can be multiplied in one statement, whereas, the Times (Spatial Analyst) tool can only handle pairs.

The following code uses multiple Raster objects and numbers and mathematical operators in a single expression and saves the results in the 'output' raster:

>>> r1 = arcpy.Raster('out1')

>>> r2 = arcpy.Raster('out2')

>>> r3 = arcpy.Raster('out3')

>>> outRast = (5\*r1\*r2\*r3)/2

>>> outRast.save('output')

>>> deloutRast

## Temporary Feature Layers

Unlike the Spatial Analyst raster computations, most ArcGIS tools that compute vector data output create a long term file (such as a shapefile or geodatabase feature class) without calling a save function.

Some exceptions can be found in the Data Management toolbox.

Some of these tools create a temporary feature layer, a collection of features in temporary memory that will not persist after the IDE session ends.

For example, the Make XY Event Layer (Data Management) tool creates a temporary feature layer.

This tool takes a tabular text file and creates a new point feature layer based on the x and y coordinate fields in the file.

The following code makes a temporary layer file ('tmpLayer') from the ‘xyData.txt’ file which contains ‘x’, ‘y’, and ‘butterfly’ fields.

>>> myData = 'C:/gispy/data/ch06/xyData.txt'

>>> arcpy.MakeXYEventLayer\_management(myData,'x','y','tmpLayer')

You can perform additional tool calls on the temporary layer within the same script (or within the same Interactive Window session).

The following code gets the record count:

>>> countRes = arcpy.GetCount\_management('tmpLayer')

>>> countRes.getOutput(0)

u'8'

No file named 'tmpLayer' ever appears in the ‘C:/gispy/data/ch06’ directory.

To save this file in long-term memory, you’ll need to call another arcpy tool.

There are a number of Data Management tools that can be used for this.

The following code uses the ‘CopyFeatures’ tool to save the layer to a shapefile:

>>> arcpy.CopyFeatures\_management('tmpLayer', 'butter flies.shp')

<Result 'C:/gispy/data/ch06\\butter flies.shp'>

The output from the Make XY Event Layer (Data Management) tool is used as input for the ‘Copy Features’ tool and a ‘butterflies.shp’ file is saved in the Chapter 6 data directory.

When the current IDE session ends (for example, when PythonWin is closed), this file will persist, but 'tmpLayer' will be destroyed.

## Using Variables for Multiple Tool Calls

Calling more than one tool in a script often involves using the output from one tool as the input for the next tool.

In this case, it becomes useful to employ variables to store tool arguments so that string literals are not repeated.

The temporary feature layer example in the previous section repeats the string literal 'tmpLayer' three times.

Using a variable name would make it easier to change this name if needed (since it would only need to be changed in one place).

Additionally, output fi le names can easily be created based on input fi le names.

For example, the following code slices the fi le extension and appends ‘Buffer.shp’ to the name:

>>> fireDamage = 'special\_regions.shp'

>>> fireBuffer = fireDamage[:-4] + 'Buffer.shp'

>>> fireBuffer

'special\_regionsBuffer.shp'

In Example6.3 , ‘special\_regions.shp’ represents regions damaged by a fire.

To fi nd the portion of the park (in ‘park.shp’) which lies within 1 mile of the fi re damage, we buffer the fi re damaged regions and then clip the park polygons on the buffer zone.

The output from the Buffer tool is input for the Clip tool.

Figure6.5 shows the input data, the intermediate output, and the final output.

Example6.4 finds the lengths of the fi re damage boundaries in two steps.

First a field is added to store this information.

Second, shape lengths are calculated in the new field.

Both of these tools use the same input dataset and field name, and these variable values are set before the tool calls.

The other two Calculate Field parameters are set as string literals and only used once in this example.

The length is calculated using the '!shape.length!' expression and 'PYTHON' indicates that it is a Python type expression.

## Calling Custom Tools

Many useful tools can be found by searching online.

You may find models or Script Tools that you want to call from within your code.

To use custom tools (those that are not built-in to ArcToolbox) use the arcpy ImportToolbox function and then call the tool using the tool name and the toolbox alias.

Tool names can differ from the labels displayed in ArcCatalog.

Labels are allowed to contain spaces.

To determine the tool name, right-click on the tool and view the ‘Name’ property on the ‘General’ tab.

To determine the toolbox alias, right-click on the toolbox and view the ‘Alias’ property on the general tab.

If it doesn’t already have one, you can set the alias here.

Example6.5 pertains to ‘customTools.tbx’ which has a Script Tool named ‘Inventory’.

The toolbox alias is ‘custom’.

The tool points to a script named ‘inventory.py’.

The header comments in this script say the following:

## A Word About Old Scripts

Python programming for ArcGIS has changed a great deal since it was first introduced.

ArcGIS Python is backwards compatible.

In other words, Python code writ ten for previous versions of ArcGIS will still run in the newer versions.

For example, an ArcGIS 9.2 Python script will work in ArcGIS 10.1.

The arcpy package is how ArcGIS 10.x gives Python access to geoprocessing.

In 9.x versions, the arcgis scripting package was used.

arcpy encompasses and enhances arcgis scripting.

Table6.3 shows prior version equivalents to importing arcpy.

## Discussion

Calling tools in ArcGIS follows a basic pattern with some variations.

The parameters required by the tools vary based on the kinds of input the tool needs.

To use Python for ArcGIS, you’ll need to become familiar with the basic pattern and use the tool help for more detailed information.

Context menus help with spelling and parameter requirements.

The ArcGIS Resources tool help parameter tables and script examples provide more detailed information.

This chapter presented the basic pattern for calling tools and highlighted the most common parameter usage questions.

Spatial Analyst toolbox objects such as the arcpy.sa.RemapValue and arcpy.sa.WOTable are somewhat complex and highly specialized.

For tools that require these arguments, run the tool via ArcToolbox and ‘Copy the code snip pet’ to get an example with sample input data.

For the vast majority of tool calls though, you won’t need to copy code snippets, because the syntax will become routine after a little practice.

The ValueTable, Result, and Raster objects introduced in this chapter are just three of the many arcpy objects.

The arcpy cheat sheet detail diagrams in this book, like the one in Figure6.3 , provide selected method and property names for several arcpy objects.

These can be used as a quick reference, but the ArcGIS Resource site contains full property and method documentation for each arcpy object.

When searching, for an arcpy object, use the word arcpy along with the name of the object to improve the search results.

For example, instead of searching for ‘result’, search for ‘result arcpy’, to bring the Result object help page to the top.

## Key Terms

Full path file name

Python expression parameters

Optional parameters

Result object

Multivalue input

Multivalue string

ValueTable object

Temporary feature layers

Map Algebra

Raster object

## Exercises

Write a Python script to solve each problem and test the script on sample data in ‘C:/ gispy/data/ch06/’.

Hard-code the workspace to this directory.

Hard-coding means specifying it with a string literal (arcpy.env.workspace = 'C:/gispy/ data/ch06' ).

To achieve a deeper understanding of the code syntax, write the scripts without using ModelBuilder.

Name the script as specified in bold print at the beginning of each question.

# Getting User Input

* Abstract

Scripts that are designed to be flexible can be reused with varying parameter values.

Relying on hard-coded parameters, specified with string literals, numbers and so forth, means users can’t change the values without opening the script and modifying the code.

A flexible script that accepts arguments for the parameters can be easier to reuse.

The geoprocessing examples in Chapter 6 specified the tool parameters inside the script.

These processes can be made dynamic by accepting user input.

This chapter covers the arcpy approach to receiving arguments and contrasts this with using sys.argv.

Then it discusses argument spacing and os module methods for handling file path arguments and getting the script path.

* Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Capture user input with two different techniques.

• Predict script behavior in case of too few arguments.

• Group characters in arguments appropriately.

• Explain why single quotations cannot be used to group arguments.

• Extract the file name, directory, and fi le extension from a full path file name.

• Join base file names with paths.

• Get the size of a file.

• Get the path of the current script.

## Hard-coding versus Soft-coding

Hard -coded tool parameters are encoded specifically within the script, so that the code needs to be altered in order to change this values.

More flexible scripts get input from the user—they are soft-coded.

The input and output features (“park.shp” and “boundingBoxes.shp”) in Example 7.1 are hard-coded.

In this chapter, we compare two techniques for accessing user arguments:

1. An arcpy function named GetParameterAsText.

2. The sys Python standard module variable named sys.argv.

## Using GetParameterAsText

The arcpy package has a GetParameterAsText function that gets input from the user.

To use this arcpy function:

1. Use arcpy.GetParameterAsText(index) , starting with index = 0, to replace the hard-coded parameters .
2. Add usage and example comments in the header to tell the user how to run the script.

Example7.2 shows the script modified with these changes.

Indices start at zero and correspond to the order in which the arguments are passed.

To run the script with user input in PythonWin, place arguments, separated by spaces, in the ‘Arguments’ list in the ‘Run Script’ dialog.

In PyScripter, go to Run > Command Line Parameters… and place the arguments in the text box (space separated).

Make sure that ‘Use Command Line Parameters?’ is checked.

Figure7.1 shows ‘boundingGeomV2.py’ being run in PythonWin with these arguments: “C:/gispy/ data/ch07” “park.shp” “boundingBoxes.shp”.

## Using sys.argv

If the script imports thearcpy package,GetParameterAsText is a good approach to use.

However, if the script is designed to be used on a machine that might not have ArcGIS installed, the built-in system module should be used instead.

Thissys module has a property named argv (short for argument vector).

The sys.argv variable stores:

1. The name of the script, including the path where it is stored (the full path fi le name of the script).

2. The arguments passed into the script by the user.

sys.argv is a zero-based Python list that holds the script name and the arguments.

The first item in the list is the full path fi le name of the script itself.

The next item (indexed 1) is the first user argument.

To soft-code a script using the sys module, you need to do these three things:

1. Import sys.
2. Replace hard-coded parameters with sys.argv[index], starting with index = 1 (since the zero index is used for the script path).
3. Add usage and example comments in the header to tell the user how to run the script.

To get three arguments passed by the user, ‘boundingGeomV3.py’ in Example 7.3 usessys.argv[1] ,sys.argv[2] , andsys.argv[3] .

Figure7.2 shows ‘boundingGeomV3.py’ being run in PythonWin with these arguments: “C:/gispy/data/ch07” “park.shp” “boundingBoxes.shp”.

## Missing Arguments

A script can fail if the user does not provide enough arguments.

When an argument is missing, the GetParameterAsText method itself does not throw an error.

Instead, it returns an empty string.

If the script in Example7.2 is run without any arguments, the script assigns an empty string to the input and output features.

Then when the tool is called, the following error is thrown, since the tool is being run on empty strings (only the last three lines of the error are shown here):

## Argument Spacing

Arguments are space delimited in the ‘Arguments’ list.

Consequently, if an argument itself has embedded spaces, it must be surrounded by quotations else it will be interpreted as more than one argument.

For example, if the intended argument is a fi le path such as “C:/African Elephant/rasters”, Figure7.3a will produce an incorrect result, but Figure7.3b will produce the desired results.

## Handling File Names and Paths with os Module Functions

GIS scripts often need to separate or join fi le names and fi le paths.

The standard os (operating system) module introduced briefly in Chapter 2, has a path sub-module for file path name manipulation, including the functions dirname, basename, join, and getsize.

Since path is a submodule of os, double dot notation (os.path.functionName) is used to access these functions.

The following code uses the basename method to extract the file’s base name from the full path file name:

This operation could be performed with concatenation too, but the join automatically takes care of slashes.

In Example7.5 , ‘copyFile.py’ uses os.path.basename to get the input file name so that it can be copied to a backup directory.

This script also uses os.path.join to create the new full path name.

Chapter 6 demonstrated how to use slicing to separate a file path from its extension when the length of the extension is known.

For example, the following code removes a three character extension (and the dot):

>>> myShape file = 'parks.shp'

>>> rootName = myShape file[:-4]

>>> rootName

'parks'

If the extension length is unknown, an os.path method can be used to split a fi le extension from its name.

The os.path.splitext splits the fi le name at the dot in the name (if there is more than one dot, it uses the last one).

It returns a tuple containing the two parts, the root name and the extension:

>>> os.path.splitext(myShape file)

('parks', '.shp')

If the name has no extension, the first item is the name and the second is an empty string: fc = 'farms'

>>> os.path.splitext(fc)

('farms', '')

Indexing the first item retrieves the root name:

>>> os.path.splitext(myShape file)[0]

'parks'

>>> os.path.splitext(fc)[0]

'farms'

Slicing, on the other hand, may not work as expected, if the fi le extension length is unknown:

>>> fc[:-4]

'f'

### Getting the Script Path

A simple way to share a script is to provide the script and the data it needs within a single common folder.

The relative path to the data remains the same as long as the recipient leaves the data in the same relative position.

In this case, the script can use its own location as the workspace.

The command os.path.abspath( \_\_file\_\_) can be used inside a script to get the full path fi le name of the script being run by PythonWin.

The \_\_ file\_\_ variable returns the name of the script.

Depending on how the script is run, this may or may not be the full path fi le name.

But calling the abspath method on the \_\_ file\_\_ constant returns the full path fi le name.

The dirname method can then be used to get the script's directory as shown in Example 7.7 .

This script also demonstrates another useful os command method mentioned in a previous chapter.

The listdir method returns a list of fi le names for every fi le in the directory that's passed into it as an argument.

The output from running this script in a directory containing 9 fi les looks like this:

## Key Terms

Hard-coding vs. Soft-coding

arcpy.GetParameterAsText(index)

sys.argv

os.path.dirname(fileName)

os.path.basename(fileName)

os.path.splitext(fileName)

os.path.abspath(path)

\_\_ file\_\_

File base name

Full path file name

## Exercises

# Controlling Flow

* Abstract

As you study the capabilities of programming for GIS, applications to your own work may come to mind.

For example, you may need to run a geoprocessing tool on distributed batches of datasets or tweak multiple tables before they can be imported into ArcGIS.

Before writing scripts for tasks like these, it’s helpful to outline the main steps without belaboring syntax details, but instead using pseudo code, a generic format for outlining workflow.

You also need to be familiar with the building blocks of workflow.

How will the code make decisions based on the input?

How will it repeat a process for each fi le? Chapters 9 and 10 describe the Python syntax for ‘branching’ and ‘looping’, but that part will seem easy, compared to the logic involved in designing the workflow.

This chapter uses pseudocode examples to introduce ‘branching’ and ‘looping’.

* Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Define the terms algorithm and pseudocode.

• Translate a GIS problem into a set of succinct steps.

• Use the terms looping, and branching to a describe workflow.

• Identify the three key components in a looping structure.

• Use indentation to group blocks of pseudocode.

• Interpret pseudocode.

## Outlining Workflow

## Key Terms

Workflow structures: sequential steps, repetition, decision-making

Looping

Branching

Pseudocode

Block structures

WHILE -loop

FOR -loop

Looping variable

Iterate

Functions (procedures, subroutines)

Nested structures

## Exercises

# Decision-Making and Describing Data

* Abstract

Scripts routinely need to perform different operations based on some deciding criteria.

The decision may be very simple, (‘if the field doesn’t exist, add the field’) or it may be more complex (‘if the shapefile has point geometry, compute a buffer, but if it has polygon geometry, perform an intersection’).

Chapter 8 introduced ‘decision- making’ structures.

These are often referred to as ‘branching’ structures, because the workflow diagram branches where the decision occurs.

For example, the ‘Yellowstone Hiker’s Flowchart’ branches in three places: ‘Do I see an animal?’, ‘Is it a bear?’ and ‘Is it a killer rabbit?’.

Pseudocode uses IF, THEN, ELSE IF, ELSE, and ENDIF key words to express decision-making workflows.

This chapter presents the Python syntax, conditional expressions, ArcGIS tools that make selections, the arcpy Describe method, handling optional input, and creating directories.

* Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Implement IF, ELSE IF, and ELSE structures in Python.

• Explain when to use only an IF block, when to use anELSE IF block, and when to use an ELSE block.

• Specify decision-making conditions with comparison, logical, and membership operators.

• Select data within a table using SQL comparison and logical operators.

• Design syntactically and logically sound compound conditional expressions.

• Identify code testing cases for branching.

• Use data properties to make decisions.

• Handle optional user input.

• Safely create output directories.

Decision-making is expressed with conditional constructs.

If some condition is true, some action is taken.

E.g., if the animal is a bear, ring a bell.

In Python, conditional constructs begin with the if keyword.

This keyword, followed by a condition, followed by a block of indented code, make up the simplest conditional construct.

The syntax looks like this:

## Conditional Expressions

### Comparison Operators

### Equality vs. Assignment

### Logical Operators

### Membership Operators

## ArcGIS Tools That Make Selections

### Select by Attributes and Temporary Feature Layers

The Select Layer By Attribute (Data Management) tool is another tool that uses a where\_clause parameter.

This tool requires a little bit of explanation for use in Python because both the input and output are temporary layers not feature classes such as shapefiles or geodatabase feature classes (Section 6.6 discussed the temporary layers in the context of the Make XY Event Layer (Data Management) tool).

Performing a selection on a feature class and saving the selection as a feature class requires making a feature layer, selecting, and saving the selection, as in the following steps:

## Getting a Description of the Data

For geoprocessing, we often make decisions based on data properties.

Thearcpy function namedDescribe allows you to access properties of a data object.

It takes one argument, the name of an Esri data element or geoprocessing object.

Examples of valid arguments include a Feature Class name, a Layer name, a Raster Dataset name, and so forth.

The following code calls theDescribe function with the Raster Dataset named'getty\_rast' as an argument:

>>>importarcpy

>>> arcpy.env.workspace = 'C:/gispy/data/ch09'

>>> rastFile = 'getty\_rast'

>>> desc = arcpy.Describe(rastFile)

TheDescribe function returns aDescribe object, which is stored in the variable nameddesc:

>>> desc

<geoprocessing describe data object object at 0x0092D2D0>

### Describe Object Properties

The Describe object has a set of properties with information about the data. These properties fall into two categories:

• Universal properties—any valid input has all of these properties.

• Specialized properties—these properties depend on the data type of the input.

Whenever you create a Describe object, it has values for the set of universal properties that are common to any object being described.

These are things like baseName, dataType, andextension, to name just a few.

To access these properties, use dot notation with the Describe object.

In this example, we print a few of the universal properties:

>>> desc.dataType

u'RasterDataset'

>>> desc.baseName

u'getty\_rast'

>>> desc.extension

u''

>>> desc.catalogPath

u'C:/gispy/data/ch09/getty\_rast'

The additional, specialized properties depend on the data type of the argument.

For example, when the argument is aRasterDataset type, like ‘getty\_rast’, there are a set ofRasterDataset Properties, such asbandCount ,compres sionType , andformat .

Again, access these with dot notation:

>>> desc.CatalogPath

u'C:/gispy/data/ch09/getty\_rast'

>>> desc.bandCount

1

>>> desc.compressionType

u'RLE'

>>> desc.format

u'GRID'

The Describe object format property should not be confused with the string method by the same name.

The format property refers to the image format (GRID, JPEG, PNG, etc.) Many of the properties return string values, but some return arcpy objects.

These objects may have their own set of methods and properties, which can be accessed with dot notation.

For example, the extent property returns an Extent object with minimum and maximum values for x, y, z, and m, as well as other properties.

The following code assigns an Extent object to a variable named bounds, then uses the variable to find the maximum x value:

>>> bounds = desc.extent

>>> bounds

<Extent object at 0x3338c50[0x125b4d70]>

>>> bounds.XMax

2167608.390378157

### Lists of

The ArcGIS Resources site lists the universal and specialized properties.

The ‘Describe object Properties’ page lists the universal properties; any valid input has all of these properties.

The ‘Describe (arcpy)’ page has a list of the specialized data types, each linked to a list of properties.

Many of the special data types have access to a few general types.

The Dataset properties and Table properties are available to many types of Describe objects.

For example, the ‘Raster Dataset Properties’ page says ‘Dataset Properties also supported.’ This means Raster Dataset typeDescribe objects also have all the Dataset properties, such as, extent andspatialReference .

Figure9.1 depicts a portion of the Describe functionality.

The Describe function is one of the arcpy functions.

It returns a Describe object.

The box highlighted in Figure9.1 lists a few of the Describe object properties.

The box (labeled “describe data objects”) represents the Describe object and lists a few of its universal properties.

The darker boxes inside this one represent some of the specialized property groups.

For example, FeatureClass type objects have six properties, plus they have access to Dataset and Table properties and RasterDataset type objects have five properties, plus they have access to Dataset and Raster Band properties.

Only a few of the properties are listed in the boxes as a reminder of the many available properties.

The ArcGIS Resources documentation contains the complete property lists for each data type.

### Using Specialized Properties

Care must be taken when using the specialized properties.

If you try to use the wrong type of property, an error will occur.

The following code attempts to use the format property but it fails because ‘park.shp’ is a ShapeFile data type so it doesn’t have this specialized property:

>>> fcFile = 'C:/gispy/data/ch09/park.shp'

>>> desc2 = arcpy.Describe(fcFile)

>>> desc2.dataType

u'ShapeFile'

>>> desc2.format

Traceback (most recent call last):

File "<interactive input>", line 1, in <module>

AttributeError: DescribeData: Method format does not

Exist

For this reason, the specialized properties are usually used inside conditional constructs which check the dataType of the object before using these properties.

The dataType property returns a string value, an alias that arcpy uses for the given type of data.

Table9.4 lists the dataType values for ten sample datasets.

Others can be found in the ArcGIS Resources pages for the data types, linked to the ‘Describe (arcpy)’ page.

Example9.7 checks the value of the dataType property instead of just using the format property on any Describe object.

If the second argument is ‘getty.tif’, the script prints ‘Image format: TIFF’.

If the input is ‘park.shp’, the script prints nothing.

### Compound vs. Nested Conditions

Sometimes you may want to check more than one condition before calling a geo processing tool.

For example, to use the Smooth Line (Cartography) tool, you can check thedataType and theshapeType .

You can do this with a compound statement, but any specialized conditions (such asshapeType ) must be placed after the more general conditions (such asdataType ) because the conditions in a compound statement are evaluated left to right.

This code causes an error:

>>> desc = arcpy.Describe('C:/gispy/data/ch09/getty.tif')

>>> desc.dataType

u'RasterDataset'

>>>ifdesc.shapeType == 'Polyline' and \

dsc.dataType in ['FeatureClass','Shape file']:

... print'Smooth line'

...

Traceback (most recent call last):

File "<interactive input>", line 1, in <module>

AttributeError: DescribeData: Method shapeType does not exist

How could you reorder the compound condition to correct this problem?

In Example9.8 , ‘smoothLineCompound.py’ correctly uses a compound statement to only call the Smooth Line tool for 'FeatureClass' or 'ShapeFile' types with ashapeType of 'Polyline' .

Compound conditions work well as long as exactly the same behavior is desired for both conditions.

However, suppose we want to tailor warning messages depending on which condition fails.

Then we need to use nested conditional constructs.

The pseudocode for this would look like this:

The nested conditions allow the script to use one ELSE block for each condition.

Example9.9 shows the corresponding script.

### Testing Conditions

Scripts that have conditional constructs, should be tested with input that fulfill each of the conditions they are built to handle.

For example, ‘smoothLineNested.py’, in Example9.9, considers three outcomes.

We can test for each of these by varying the input.

1. Input data type is FeatureClass or Shapefile and shape type is Polyline.

Input: C:/gispy/data/ch09 trails.shp

Output:

Smooth line created C:\gispy\scratch\output.shp.

1. Input data type is FeatureClass or Shapefile but shape type is not Polyline.

Input: C:/gispy/data/ch09 park.shp

Output:

Warning: shape type is Polygon. SmoothLine only works on Polyline shape types.

1. Input data type is neither FeatureClass nor Shapefile.

Input: C:/gispy/data/ch09 getty.tif

Output:

Warning: Input data type must be 'FeatureClass' or 'ShapeFile'. Input dataType: RasterDataset

## Required and Optional Script Input

Conditional constructs can also be used to handle optional script arguments.

The script must somehow check whether or not optional arguments have been given and set a default value if necessary.

For example, the workflow represented by the following pseudocode optionally takes one argument, a directory path, and lists the fi les in the given directory or the fi les in the script directory:

IF argument found THEN

SET working directory to argument.

ELSE

SET working directory to script directory.

ENDIF

List files in working directory.

This can be implemented using thesys moduleargv list, as in Example9.10 .

Sincesys.argv is a Python list, if a script uses an index that is not available, an

IndexError will be reported:

>>>importsys

>>> sys.argv[1]

Traceback (most recent call last):

File "<interactive input>", line 1, in <module>

IndexError:list index out of range

To avoid this, you can check the length of the list before trying an index for an optional argument.

Recall that the fi rst item in this list (sys.argv[0] ) is the script file name:

>>> sys.argv[0]

'C:\\gispy\\sample\_scripts\\ch09\\scriptPathOptionalv1.py'

This means that the length of the list is the number of user arguments plus one.

Therefore, if the user passes in one argument, the list length is 2.

Alternatively, this can be implemented with the GetParameterAsText function, as in Example9.11 .

The default value returned by this function is an empty string (‘ ‘).

Since an empty string is considered false, and all other strings are considered true, we can use the value of arcpy.GetParameterAsText(0) as the conditional expression.

Both versions of this script work with or without an argument.

Conditional constructs can also be used to enforce required arguments.

The km/mile converter in Example9.12 requires two arguments, a numerical distance and a distance unit.

All arguments are passed into sys.argv as string types.

The first argument is a numerical value, so we need to use the built-in float function to get the numerical value (cast the value).

The unit is converted to all lower case when it’s tested (usingunit.lower()), so that the input is not case sensitive.

If either argument is missing, the script will throw an IndexError exception.

Since the user might not understand this, it could be helpful to instead print a message about how the script is designed to be run.

Example9.13 adds this information and also makes the second argument optional.

If no user arguments are given, the script warns the user and exits.

A sys module function named exit is used to exit the script.

If one user argument is given, the script sets a default distance unit, otherwise the unit is set to the second argument.

The remainder of the script is the same as version 1.

When handling several user input cases, it’s a good idea to test an example of each input.

Table9.5 shows test cases for ‘distanceConvert2.py’.

The last test case (2.2 bananas) gives an example of how the script could be made more robust.

This improvement is left as an exercise.

## Creating Output Directories

Sometimes we want to store output fi les in a different workspace than the input file workspace.

For example, to create a backup copy of fi les using the same name in another directory or to group a batch of output fi les in a directory on their own, we need to send the output to another directory.

In some cases, the output directory may exist; in other cases, the script might create it for us.

If this isn’t handled carefully, the script could try to write output in a directory that doesn’t exist or it could try to create a directory that already exists.

Either of these would cause the script to fail.

os module functions can be used to avoid these problems.

The os.path.exists function checks if a fi le or directory exists and the os.mkdir creates a new directory as shown in the following code:

## Key Terms

If, elif, else, Python keywords

Conditional constructs

Boolean expressions

Conditional expressions

Compound conditional expressions

The os.path.exists function

The os.mkdir function

The CreateFileGDB tool

## Exercises

# Repetition: Looping for Geoprocessing

Abstract

Advances in technology are enabling the collection and storage of massive amounts of GIS data.

To learn from this data we need to conduct analysis efficiently.

Batch processing is a powerful scripting capability, saving time by auto mating repetitive tasks.

Chapter 8 discussed the three basic workflow structures (sequences, decision-making, and repetition) in terms of pseudocode and Chapter 9 presented the Python syntax for decision-making and describing GIS data.

This chapter focuses on Python repetition structures and looping syntax, Python FOR loops and WHILE -loops, looping with the range function for geoprocessing, nested looping, and listing directory contents.

Then the chapter concludes with a tip about debugging whitespace glitches.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Implement WHILE -loops and FOR -loops in Python.

• Identify the three key iterating variable components in aWHILE -loop.

• Explain how PythonFOR -loops work.

• Repair infinite loops.

• Call a geoprocessing tool in aWHILE -loop to vary a numerical parameter.

• Automatically generate numerical lists.

• Loop with therange function.

• Branch and loop within loops.

• List the files in a directory.

• Geoprocess each file in a list.

• Correct indentation errors.

## Looping Syntax

Python has two structures for performing repetition, WHILE -loops and FOR -loops, which use the Python keywords while and for.

They provide two slightly different approaches.

FOR -loops are used more often in geoprocessing scripts; however, both looping techniques are used, so it is helpful to be familiar with both.

We’ll start with WHILE -loops which provide an easier introduction to looping because Python WHILE -loops have the looping mechanisms exposed; whereas, the workings of Python FOR -loops are less explicit.

Many programming languages share a similar WHILE -loop structure with Python; whereas, the structure of Python FOR -loops is less common; both will seem easy, once you gain some practice.

### WHILE-Loops

### FOR-Loops

## Nested Code Blocks

Scripts often need to use nested codes structures—such as conditional blocks inside FOR -loops (Example10.6 ) orFOR -loops insideFOR -loops (Example10.7 ).

Notice how the indentation steps inward as the nesting occurs.

In Example10.6 theif , elif , andelse keywords are aligned at one level and theprint keywords con tained within these blocks are aligned at the next level.

The ‘emotaLoop.py’ script gets its name from the emoticons it prints.

This script loops through each fi le name in the list and determines which face to print, based on the fi le extension.

The stringend swith method, which returnsTrue orFalse , is used to check the fi le extension.

## Directory Inventory

FOR -loops can be used with Python built-inos module functionslistdir and walk to navigate data and directories.

We demonstrate the simplelistdir func tion here and the more complexos.walk function in Chapter 12 .

Thelistdir function lists the fi les in an input directory, such as sample data directory ‘pics’.

The following code prints a list of the fi les in ‘C:/gispy/data/ch10/pics’:

## Indentation and the TabNanny

Chapter 4 introduced the PythonWin syntax check button which checks the syntax and runs the TabNanny as well.

Now that you’re writing indented blocks of code, let’s revisit the TabNanny, which checks for consistent spacing.

The TabNanny inserts underline marks before indented code if inconsistencies in the indentation are found.

By now you know that indentation within a block needs to be aligned.

The TabNanny marks indicate a more subtle problem which sometimes occurs when code is copied from sources such as PDF fi les or Microsoft® software docu ments into PythonWin.

For example, the following code was copied from a PowerPoint® presentation to PythonWin:

## Key Terms

The while keyword

The for keyword

The for and in keyword pairing

Iterating variable

Sequence data objects

Nested looping

The os.listdir function

## Exercises

# Batch Geoprocessing

Abstract

Python enables you to perform geoprocessing tasks on batches of Esri format data files, fields, and workspaces.

This chapter focuses on how to get lists of Esri data and use these lists together with FOR -loops for batch geoprocessing.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• List the file rasters, feature classes, workspaces, datasets, workspaces, and tables within a workspace.

• List files with a common type and/or substring in the name.

• List the fields in an attribute table.

• Get the name, type, and length of an attribute table field.

• List the printer names, tool, toolboxes, and environment settings for an ArcGIS install.

• Batch geoprocess lists of GIS data.

• Step through each line and watch variables to detect bugs.

## List GIS Data

Python enables you to perform geoprocessing tasks on batches of data fi les, fields, and workspaces.

The os.listdir function lists all the fi les in a directory, but we often need to perform batch processing on a specific GIS fi le type.

The arcpy module has a set of listing methods to get lists of these items.

The arcpy overview diagram shows two of these methods in the left column under ‘listing’.

The large box in Figure11.1 shows a more extensive list of these methods.

Categories I and II methods are used for listing data.

As the names imply, these methods return lists of datasets, feature classes, files, fields, and so forth.

This chapter discusses how to use these methods.

Datasets’, ‘files’, ‘tables’, ‘rasters’, and ‘workspaces’ are fairly broad terms.

If you’re not sure which types of files these methods will list, check the ArcGIS Resources site.

Or, if you have specific data in mind, it’s easiest to simply try it on sample data—set the workspace, call the data listing method, and print the returned list.

Example11.1 lists the Esri workspaces found in ‘C:/gispy/data/ch11’.

According to this function, a fi le geodatabase (e.g., ‘C:/gispy/data/ch11\tester.gdb’), as well as a folder (e.g., ‘C:/gispy/data/ch11\pics’) is a workspace.

The second loop lists the tables found in ‘C:/gispy/data/ch11’.

In this example, it returns some ‘csv’ fi les, ‘txt’ files, and ‘dbf’ files.

Others items not listed here might qualify as well.

All ‘txt’ files are listed, regardless of content (‘loveLetter.txt’ has no field headers or records).

Not all ‘dbf’ (dBASE) tables are listed.

Only independent dBASE tables in the workspace are listed.

Dependent tables are ones associated with shapefiles.

Even though ‘park.dbf’ is in ‘C:/gispy/data/ch11’, this table is not listed because it is part of a set of files that make up a shapefile.

## Specify Data Name and Type

Most of the arcpy listing methods have optional parameters.

The purple box in Figure11.1 shows the function signatures, which list the arguments in parentheses behind each method name with optional arguments in curly braces.

Most list methods have two optional arguments for filtering names and types.

For example, the ListRasters signature includes no required arguments, but it has two optional arguments, wild\_card and raster\_type , as shown here:

If no arguments are used, all raster datasets in the current arcpy workspace are listed.

The optional arguments allow you to restrict the search to a subset of the rasters based on names and/or types.

The wild\_card argument is a string name which can use asterisks as wild cards.

A wild card stands for a string (or substring) that can have any value.

Any string of characters can be substituted in the position where the asterisk appears in the string.

For example,'\*am' could stand for strings such as ‘spam’, ‘ham’, and ‘am’, but it could not stand for ‘hamper’ and the wild card string'\*elev\*' could stand for strings like ‘elevation’, ‘relevant’, ‘peak\_ elev’, and ‘elev’.

The wild\_card argument allows you to specify a data name substring using the asterisk as a placeholder for any string.

Then if data has a semantic naming schema, e.g., any rasters related to elevation, might contain the substring ‘elev’, you can use this to select a subset of fi les for processing.

The asterisk can be used any where in the string.

Example11.2 demonstrates wild card usage.

Can you predict the output from each of these?

Example11.2a lists all the rasters in the workspace using'\*' as the wild card.

This has the same affect as passing no arguments.

Example11.2b uses'elev\*' as the wild card parameter; it specifies a prefix, by placing an asterisk at the end of the string.

To specify a suffix, use an asterisk at the beginning.

If no asterisk is used in a string, it looks for that exact string.

Though it can be done, you would usually not want to use the wild card without asterisks.

Example11.2c uses ‘elev’ as the wild card.

This returns only the raster named ‘elev’.

If this is the intended outcome, thearcpy.Exists function would be a better choice for checking one specific data element.

## List Fields

Unlike Category I methods, Category II methods require one argument, an input dataset—the items they are listing (such as fields) pertain to a dataset, not an entire workspace.

These methods are slightly more complicated than Category I because they return lists of objects instead of string names.

The ListFields method returns a list of Field objects, not string names of fields.

For example, the following code gets the list of Field objects for input file ‘park.shp’:

Each item in this list is a Field object (the numbers indicate location in memory).

A Field object can be thought of as a compound structure that contains multiple pieces of information about the field, that is, the field properties such as name, type, length, and so forth (Figure11.2 lists more of these properties).

Usually, we don’t want to use the Field object, but rather the field name or some other property.

To access these properties, you need to use object.property format.

The following code prints the field names as seen in the attribute table:

## Administrative Lists

GIS scripts use Category I and II listing methods most frequently.

Category III methods, like Category I methods return lists of names.

Category III methods provide access to ArcGIS software and computer system information for administrative purposes.

For example, ListEnvironments returns a list of the environment settings, ListInstallations returns a list of the type of ArcGIS software installed on the system (server, desktop, engine), ListPrinterNames returns a list of printer names available to the system, ListTools returns a list of the ArcGIS tools, and ListToolboxes returns a list of the built-in ArcGIS tool box names and aliases with the format ‘toolbox\_name(toolbox\_alias)’.

The arcpy package has additional listing methods for web mapping administration.

## Batch Geoprocess Lists of Data

## Debug: Step Through Code

## Key Terms

The ListFeatureClasses, ListRaster, ListTables, and ListFields methods

Asterisks as wild cards

The arcpy Field object

Step though code

Step, step over, step out buttons

Watch window

## Exercises

# Additional Looping Functions

Abstract

Most GIS scripting projects are motivated by the desire to automate repetitive tasks.

The Python programming language has a number of special functions and structures designed to facilitate various common demands in repetitive workflows.

In this chapter, we’ll highlight some additional useful techniques for GIS scripting, related to Python FOR -loops, including list comprehension, the built in enumerate and zip functions, and the os.walk function.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Modify lists efficiently.

• Get the index and the value of a list within a FOR -loop.

• Loop through two or more lists simultaneously.

• Traverse subdirectories.

## List Comprehension

## The Built-in enumerate Function

## The Built-in zip Function

## Walking Through Subdirectories

FOR -loops can be used with Python built-in os module functions listdir and walk to navigate data and directories.

As discussed in Chapter 10, the listdir function lists the fi les in an input directory, but does not list fi les in subdirectories.

The os function calledwalk provides a handy way to process fi les distributed in multiple levels and subdirectories.

To use thewalk function in aFOR -loop, you need to placethree comma-separated variables between the for and in keywords as shown in Example12.3 which prints the values of the iterating variablesroot, dirs , andfiles .

We use the name root to stand for root directory , which is the top-most directory in a hierarchy.dirs stands for subdirectories in the root directory and files stand for fi les in the root directory.

The iterating variables are automatically assigned values as the FOR -loop iterates.

The listdir function allows a script to list the fi les immediately inside of a given directory; whereas, the walk allows a script to walk recursively to every fi le under a given directory.

It does this by using an internal list of subdirectories.

This list is not exposed to the user, but in order to explain how the walk works, we’ll name it toDoList and keep track of its contents.

The walk adds each subdirectory that it encounters to the to-do list and it visits each one in the list.

It removes a directory once it has been visited.

When the to-do list is emptied, the walk ends.

In Example12.3, ‘walkPics.py’ is walking through ‘pics’, a directory of travel pictures from Turkey, Italy, and Israel.

Figure12.1 depicts the ‘pics’ directory tree.

The ‘pics’ directory contains Turkish photos and has two subdirectories, ‘Italy’ and ‘Jerusalem’, which, themselves contain more items.

Compare the directory tree in Figure12.1 with the output in Example12.3.

The first time through the loop, the print statements yield this:

## Key Terms

List comprehension

The string lstrip method

Built-in enumerate function

Built-in zip function

The os.walk function

The arcpy.Exists function

## Exercises

# Debugging

Abstract

Even experienced programmers create bugs —coding errors that take time to correct.

Syntax has to be precisely correct and programming is a complicated task.

Computers do exactly what the program says; they are unforgiving.

Fortunately, there are tools for locating, understanding, and handling coding errors.

Integrated Developments Environments (IDEs), such as PythonWin, have syntax checking and debugging tools.

The arcpy package has functions for gathering information about errors and the Python language itself has functions and keywords for handling errors gracefully.

This chapter covers error debugging and related topics.

The three types of programming errors are: syntax errors, exceptions, and logic errors.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Check for syntax errors.

• Recognize exceptions.

• Run code in debug mode.

• Watch variables as code is running.

• Step through code one line at a time.

• Set stopping points (breakpoints) at crucial points within the code.

• Run the code to a breakpoint.

• Set conditions for stopping at a breakpoint.

• Design input to test the code.

## Syntax Errors

## Exceptions

The parser can only detect violations of syntax rules.

If the syntax rules are adhered to, Python will run (or attempt to run) the script.

If it encounters an error when attempting to execute a line of code, it throws an exception and stops the run at that point.

Chapter 2 discussed Python built-in exceptions such as NameErrors and the traceback errors they print.

Can you see why the following script throws a NameError exception?

## Logic Errors

Errors do not always generate exceptions.

A program containing logic errors can run smoothly to completion and have inaccurate results.

Because logic errors are not detectable by the computer, we have to understand the problems we’re solving and inspect the code closely when we perceive unexpected results.

The following code for normalizing the time-series dates contains a logic error:

Logic errors come in many forms, but they all arise from the same basic problem: your code is doing something other than what you desire and expect.

Once you perceive a mistake in the results, it may take time to discover the source.

Many beginning programmers avoid using IDE’s built-in debugger and try to use the “poor man’s debugger”—debugging by simply printing variable values throughout the code.

This is effective some of the time, but in many cases, using the debugging functionality reveals the problem more quickly.

The remainder of this chapter dis cusses debugging toolbar buttons and how to use breakpoints to improve debugging technique.

## PythonWin Debugging Toolbar

When code raises exception errors or contains logic errors, you often need to look more closely at variable values.

With an IDE such as PythonWin, code can be run in the standard way, using the ‘Run’ button or it can be run in debug mode.

Chapter 11 showed how to use the debugger to step through the code one line at a time and watch variable values change as the code executes.

Recall these points from the debugger discussion in Chapter 11 :

### Using Breakpoints

## Running in Debug Mode

## PyScripter Debugging Toolbar

## Debugging Tips

## Key Terms

Bugs

Syntax errors

Parsing

Exceptions

Logic errors

Debugger

Debugging toolbars

Debug mode

Watches

Breakpoints

Breakpoint conditions

## Exercises

# Error Handling

Abstract

Data corruption and locking can cause geoprocessing scripts to crash and throw exceptions.

Other influences, such as user input values, can also cause scripts to crash.

The topic of this chapter is using error handling structures to control script behavior when exceptions occur.

Error handling can suppress those alarming trace back messages that exceptions throw and provide a smoother way to proceed.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Handle potential errors from user input.

• Use error handling keywords,try and except.

• Anticipate named exceptions.

• Print geoprocessing messages.

• Avoid crashing in the midst of a loop.

• Identify when to use error handling instead of conditional blocks.

## try/except Structures

### Using Named Exceptions

### Multiple except Blocks

### Error Handling Gotcha

## Geoprocessing and Error Handling

### Getting Geoprocessing Messages

### The arcpy Named Exception

## Catching Exceptions in Loops

## Discussion

Exceptions come in two flavors, ones which are caused by errors in the code and ones which are caused by some outside influence, such as input data.

The first kind should always be resolved by the programmer; The second kind resides in a somewhat gray area.

The need for error handling depends on the context.

For example, if a script that requires numerical input is being run via a graphical user interface which limits the user input to numerical values, there’s no need to handle a ValueError for that input.

Some situations also require a decision between using conditional constructs and error handling.

Conditional blocks have certain similarities to try/except blocks.

With both of these structures, the execution can be diverted depending on conditions.

In some situations, conditional blocks could be used to achieve the same effect as try/except blocks.

The try/except blocks should be used to handle exceptional cases—when something has gone wrong.

When, on the other hand, both IF and ELSE alternatives are normal acceptable behavior, conditional blocks are more apt.

For example, the following code checks if an argument has been provided and if not, it sets a default value:

## Key Terms

try and except blocks

Named exceptions

arcpy.GetMessages()

Catch exceptions

The arcpy.ExecuteError exception

arcpy.GetMessage(index)

arcpy.GetMessageCount()

## Exercises

# User-Defined Functions

Abstract

Workflows often have sequences of common steps repeated within or across scripts.

Functions allow programmers to group related steps of code, name them, and reuse them by calling them by name.

This chapter discusses defining functions with required and optional parameters, returning values from functions, organizing code with functions, and managing variables inside and outside of functions.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Define and call custom functions.

• Pass values into and out of functions.

• Write functions with optional arguments.

• Create functions which return multiple values.

• Describe when to employ a user-defined function.

• Explain why some functions have parameters and/or return values.

## A Word About Function Words

By now, you are quite familiar with calling functions that are built-in (e.g., float , len , and range ) or ones that are available when a module is imported (e.g., os.path.dirname or arcpy.Describe ).

As your own scripts become more complex, you will want to start writing your own functions.

Functions provide a way to organize code so that it can easily be reused.

As a quick preview, look at the following code, which defines a function named printBird:

### How It Works

We’ll use the three step buttons on the debugging toolbar (Figure15.1 ) to explore how functions work.

The ‘‘Step” button would be more aptly named ‘Step in’.

In this book, we’ll refer to it as the “Step (In)” button.

This button steps into a function that is being called.

The ‘Step over’ button executes the function being called, but does not step inside.

The ‘Step out’ button steps out of the current function.

The black lines on the buttons represent lines of code with indentation and the arrows represent how the buttons control the debugging cursor.

### The Docstring

The string literal on the first line of code inside a function (such as,'''Print user arguments.''' in Example15.1 ) is called a docstring.

The docstring documents the function’s purpose.

A one line docstring suffices for simple functions, but more than one line can be used if needed.

Triple quotes are necessary for multi-line docstrings and style guidelines encourage triple quotes for single line docstrings to maintain consistency.

## Custom Functions with Arguments

### Script Arguments vs. Functions Arguments

### Optional Arguments

Many functions have optional parameters as well as required parameters.

For example, the arcpy Buffer function requires three arguments (input features, output name, and buffer distance), but it also has four optional arguments that can be used to further specify the buffer behavior.

Custom functions can be designed to handle optional parameters.

Optional parameters are given a default value in the signature with the following format:

## Returning Values

We’ve used functions that simply print information or modify the environment (e.g., the built-in help function or the arcpy Delete function);

And we’ve used others that return values (e.g., the built-in round function or the arcpy ListRasters function).

Now we’ll discuss how to return values in custom functions.

In terms of our butler metaphor, he can perform tasks that modify our environment (e.g., dim the lights) and he can also perform tasks that generate a tangible result (e.g., bring tea).

Custom functions can call other custom functions that are defined within the same module.

Both getfieldNames and fieldExists are defined in the ‘field Handler.py’ script, so fieldExists can call getFieldNames .

A custom function can also be called from another script, but the syntax is slightly different as discussed in the upcoming chapter on custom modules.

Whenever a return statement is reached within a function, the execution exits the function and goes back to where the function was called.

This means that any lines of code that follow a return statement in the execution flow will not be executed.

The ‘count Intersection’ function in Example15.4 handles this incorrectly.

The function calculates a temporary dataset, an intersection of features and then it calculates the number of features in the intersection fi le, and returns the value.

Though the author may have intended to delete the temporary dataset before leaving the function, the code will never reach the Delete (Data Management) tool call because it occurs after the return statement.

The examples given so far have used the return keyword only once.

It can be used more than once in a function, though it’s not recommended.

Example15.5 imports the Python built-in datetime module to deal with Gregorian calendar dates.

The call to this function passes in the birth date April, 20, 2003.

This is converted to a datetime object.

It then finds this year’s birthday and compares the birthday to today’s date.

If the birthday has already occurred this year, it returns the difference between the years;

Else, it takes an additional year off and returns this value.

The *calculateAge* function uses two return statements, but it could easily be rewritten to only use one.

Execution leaves the function when it reaches a return statement.

When multiple return statements are used, there is more than one way to exit the function.

Style guidelines discourage multiple return statements because, ideally there should only be one way to exit a function— making the code easier to interpret.

### A Common Mistake: Where Did the None Come from?

In Python, all functions return something even if a return statement is not used.

The return statement returns a value explicitly and should be used if the function is intended to return a value.

However, if no return statement is used, the function returns None , which is a Python built-in constant that represents a null value.

It’s as if an implicit return None statement is added to the functions when no explicit one is used.

The following script demonstrates a common mistake involving this phenomenon:

### Returning Multiple Values

At times, you may want to return more than one value with a function.

Suppose, for example, the function returns both the x and y coordinates of a point.

Both x and y can be returned as separate values by using a comma to separate the values in the return statement.

This returns a Python tuple.

The following code defines amid Point function which returns an x and a y value in the return statement:

Alternatively, a single tuple variable can be used to store multiple returned values.

Example15.6 gets the current time before and after walking through the given subdirectory using the arcpy.da.Walk method.

The count is found and printed for each file.

Then the diffTime function is used to calculate the amount of time elapsed.

The diffTime function returns a tuple of time components (weeks, days, hours, and so forth).

The variable t is used to store the tuple when it is returned.

Then a print statement indexes into the tuple to access the returned values.

## When to Write Functions

To identify places where functions would be useful in your code, find related blocks of code that are repeated within scripts.

For example, the following code uses several statements to print selected portions of the geoprocessing messages after two tools are called:

Example15.7 moves the repeated code in ‘scriptWithoutFunction.py’ into a function name dreportResults .

The script then calls the function twice, clear ing the clutter so that it’s easier to tell at a glance the main activity occurring in the script—calling three geoprocessing tools, AverageNearsestNeighbor , Intersect , and GetCount .

It may only make sense to call a function such as printArgs (in Examples15.1 ) one time in a script—there is only one set of arguments coming into the script, so it’s unlikely to be useful to print them more than once.

However, the function still bundles those related code statements together, so that the details of the operation appear as one statement within the main flow of the code and the reader can chose to ignore these details to focus on understanding the overall purpose of the function.

A well-named function and its docstring can serve as a sufficient shorthand to signify its purpose.

Also, although a function such as printArgs may only be useful once in each script, it is likely to be useful in many scripts.

Grouping the code into a function makes it easier to grab the related block and insert it into another script.

In fact, in an upcoming chapter, we’ll discuss how to call a function that’s in another script, so that you can write a function once and call it from many scripts.

### Where to Define

In addition to the constraint of defining functions before calling them, programmers follow a few other patterns so that it’s easy to locate function definitions in the script.

The standard pattern groups function definitions together near the beginning of the script.

Generally, they should not be dispersed throughout the code.

Figure 15.2 shows an example of poor organization (on the left) and improved organization on the right.

The header comments should be first, followed by any imports.

Next, all the functions should be defined (usually in alphabetical order by the function names).

The main processing activities and any calls to the function should follow the function definitions.

Generally, functions should not be defined inside a loop.

## Variables Inside and Outside of Functions

When programming with functions, you need to be aware of several scenarios where mutable variables work differently from immutable ones.

This section gives some examples and guidelines for avoiding confusion.

### Mutable Arguments Can Change

Mutable and immutable variables behave somewhat differently when it come to functions.

We previously discussed the concept of mutability in the context of lists and strings and how their methods work.

Recall that indexing can be used to change the values of items in a mutable sequence such as a list:

### Pass in Outside Variables

The first line of code in Example15.8 assigns the value of 5 to the variable named x.

This variable is created within the script but not within a class or another module—we’ll refer to this as a script level variable.

When a function uses a variable such as this, the variable should be passed into the script as an argument.

The con sequences of failing to do so are different depending on mutability.

If the data type is immutable, the function cannot alter the value of the variable.

Attempting to do so throws an exception.

In Example 15.8, the numerical variable, x , is defined at the beginning of the script, outside of the function definition.

This numeric (immutable) variable can then be used anywhere in the script outside of the function.

E.g., it is printed before the call to addOne .

But x cannot be modified inside the definition of addOne .

An Unbound LocalError occurs when the function attempts to assign a new value to x.

## Key Terms

Custom functions

Default arguments

Docstrings

The return keyword

Script level variable

## Exercises

# User-Defined Modules

Abstract

User-defined functions enable code reuse within a script; they can also be called from other scripts.

To amplify code reusability, functions can be defined in a supporting script.

Scripts that house sets of related function definitions and other related code are referred to as modules.

This chapter focuses on defining and importing user-modules.

It also discusses how to structure development code within a module.

Finally, the chapter concludes with a practical example for managing GIS temporal data and time attributes.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Articulate the purpose of a user-defined module.

• Create a user-defined module containing related functions.

• Import distributed user-defined modules.

• Write absolute and relative versions of a path.

• Call a function in a user-defined module.

• Use a conditional construct to exclude code in a user-defined module.

## Importing User-Defined Modules

Functions within a user-defined module can be called from other scripts.

A module is a Python script containing related function definitions and Python statements.

Every Python script is a module, but we refer to it as a module particularly when we are importing it to use its functions in another script.

You’re already familiar with importing Python built-in modules and the arcpy package installed with ArcGIS software.

Our geoprocessing scripts often begin with an import statement like this:

## Using Functions in Another Module

To call a function in another module, import the module and then you can use dot notation with the following format:

## Modifying User-Defined Modules (Reload!)

When you import a user-defined module, it loads the module and stores the name in an internal list.

As you step through a script in the debugger, you may notice that the first time you step past an import statement for a large package, such as arcpy, the import statement takes a few seconds; whereas, if you run the same script again within the same PythonWin session, you can step past this import instantly.

This is because during the second run, Python quickly looks up the module in its internal list of loaded modules, finds that module name, and does not reload the module contents.

This makes Python more efficient, but it can be baffling when you first work with user-defined modules, because it can seem as if your updates to a user defined module are not being heeded.

As an example, run sample script ‘sortString.py’ (if you haven’t already done so).

Next, make a change to ‘listManager.py’ by adding the delimStrLen function in ‘listManager.py’.

To do so, you can select the following lines of code which are commented out in the script and use the block uncomment shortcut (Alt + 4):

## Am I the Main Module? What’s My Name?

When you reload a module or import it for the first time, it not only stores the names of the functions, but it also executes any statements in that script that are outside of the functions.

As an example, uncomment the reload statement in ‘sortString.py’ and then uncomment the following code at the end of ‘listManager.py’:

## Time Handling Example

User-defined modules group related functions for convenient re-use.

Python has two built-in time related modules, time and datatime.

The user-defined report Time module in Example16.3 (located in ‘C:\gispy\solutions\ch16\exercises\exerSupportCode’) contains some convenient functions that leverage these built-in modules.

It uses the ctime and sleep methods from the time module.

The time module can access the current time and convert time across time zones.

The ctime function returns a string representing the current date and time.

The following example shows what it would have printed if called just before midnight on New Year’s Eve back in 1999:

## Summary

We’ll be discussing a Python structure called a class in an upcoming chapter.

User-defined modules are often used as containers for classes, so several of these concepts will come up again when we reach that topic.

The following list summarizes the key concepts in this chapter:

* When calling a function from within the module where it is defined, you just need to use the name. When calling it from another module, you need to start with the name of the module and use dot notation, so that Python knows where to find the function.
* Importing a user-defined module in the same directory is the simplest scenario. The system path does not need to be appended. In all other cases, the system path needs to be appended.
* You must use a separate statement to import sys and a user-defined module when the system path needs to be appended. First import sys before appending the path, then use a second import statement for the user-defined module.
* You can append a hard-coded path, but in most applications this approach hinders portability. Generally, scripts dynamically determine their own location and then append a relative path to imported modules.
* When you make modifications to a module, remember to reload it so that the changes are recognized by the script that imports the module.
* You can use a conditional expression that checks if you’re in the main script to exclude select portions of user-defined module code from imports. This allows you to use the script both as a main script and as a supporting module.

## Key Terms

User-defined module

sys.path variable

Built-in \_\_ file\_\_ variable

Absolute vs. relative paths

Built-inreload function

Built-in \_\_name\_\_ variable

Built-in time and datetime modules

## Exercises

# Reading and Writing with Cursors

Abstract

Batch processing can be extended to work with individual records within GIS data tables.

Previous chapters listed fields in multiple fi les within nested directories; this chapter adds another dimension to data exploration.

Each data table is made of records, the rows of values in the attribute table, which can also be listed within a script.

Though the term ‘records’ may be more familiar to database users, we’ll use the term ‘rows’ to refer to data records for consistency with the arcpy documentation.

Cursors and file input/output techniques, enable reading and writing individual rows within data tables.

The focus of this chapter is arcpy cursor functions, which work with the rows in Esri attribute tables, such as feature class tables.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Read and modify GIS attribute table files.

• Query the data for a select set of table rows.

• Use error handling in cursor scripts.

• Prevent data locking errors.

• Update or insert field values.

• Insert point and polygon data.

## Introduction to Cursors

GIS attribute tables are in proprietary formats not accessible with generic Python text file reading methods.arcpy cursors are objects designed specifically to accommodate GIS tables.

You can create a cursor by calling an arcpy function and specifying a table.

Then, depending on the type of cursor you requested, it can do one or more of the following:

• Read attribute values in rows.

• Modify attribute values in rows.

• Delete rows.

• Insert new rows.

For example, with cursors, you could filter to find the location of last year’s forest fi res, you could increment the walkability index for homes with average commute times less than 15 minutes, you could delete the records in a parcel dataset that have a blank land use value, or add new emergency call records to a log.

The arcpy package provides three flavors of cursors to handle different functionalities:

• Search cursors for reading.

• Update cursors for updating existing rows and deleting rows.

• Insert cursors for inserting new rows.

The 10.1 and higher arcpy package provides both classic cursor functions and newer data access cursor functions.

The classic cursors are the original implementation.

The classic cursor function signatures are shown in Table17.1 .

If you are using an earlier version of ArcGIS, you’ll need to use this syntax.

These are still available in newer versions, but a new set of cursor functions have been added.

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

Double dot notation is used to call these functions within the da module.

Table17.2 shows their signatures.

Notice the differences between the classic and data access SearchCursor signatures; Single dot versus double dot and one required argument versus two are some of the obvious differences.

Both sets of functions return a cursor object which allows you to iterate through the rows.

However, the data access cursors are significantly faster than the classic cursors, so this chapter focuses on the data access cursors.

Cursors are quite useful, but proceed with caution; Cursors lock the data while they are operating, meaning other ArcGIS Desktop processes cannot simultaneously delete or otherwise modify the data.

Data cannot be previewed while edits are being made with update or insert cursors.

If necessary, make a copy of the data and view this table instead of the original during cursor operations.

Scripts need to delete cursor objects for locks to be released.

Also, update and insert cursors can modify your data, so backup data before testing cursor code.

This chapter begins with search cursors, to illustrate basic cursor object syntax.

## Reading Rows

The data access SearchCursor function has two required arguments: the input table and a list of fi eld names.

This function returns a search cursor object.

The cursor object is used to access the rows.

The following code creates a search cursor for the ‘FID’, ‘FireId’, and ‘FireName’ fields in the table shown in Figure17.1 :

## The Field Names Parameter

The field names can be specified in any order within the field\_names parameter list.

As mentioned earlier, the row index for that field depends on the order in which they are specified, not the order of the fields in the attribute table.

Compare the field name ordering in the following code to the order of the fields in the attribute table shown in Figure17.1 :

The classic arcpy cursors do not require a field name list; These cursors always contain all fields.

Data access cursors can be made to simulate this behavior.

Using a string asterisk ('\*') for the field name parameter obtains all fields.

The drawback is that this may affect performance for large files.

In case the field names are not hard-coded in the script, the field names can be derived using the ListFields method.

Example17.1 gets a subset of the fields based on the field type.

This script lists all the fields, then calls a user-defined function that returns a list of field names for all fields except those that are ‘Geometry’ or ‘OID’ type data.

OID stands for object identifier, the unique key for each record created automatically when a table is created.

Example17.1 uses a data access cursor without hard-coding the field names.

The last line of code in Example17.1 uses the del key word to delete the cursor.

We’ll look at what this does and why this is important shortly.

## The Shape Field and Geometry Tokens

Cursors not only provide access to the row values you see in the attribute table of a file, they can also access each feature’s geometric information.

When a feature class attribute table is viewed in ArcGIS, the ‘Shape’ column shows a shape type, such as ‘Polygon’.

However, more information about each feature is stored internally for visualization and geoprocessing operations.

This information can be accessed through arcpy Geometry objects.

Notice that the field ‘type’ of the ‘Shape’ field is a ‘Geometry’.

## Looping with Cursors

Repeatedly calling the next method yields each subsequent row until the rows are exhausted.

However, most search and update cursor scripts use looping.

If the intention of the script is to use all rows, a FOR -loop should be used instead of the next method.

The cursor is not a Python list, but is an iterable object—an object which is capable of returning its members one at a time.

In other words, you can use it as the sequence in a FOR -loop just like looping on a list.

The iterating variable gets the row tuples as it loops.

The following code loops through the rows of the fi re table and prints each FireName :

## Locking

### The del Statement

The del Python keyword deletes objects from memory.

The following code deletes the cursor object and releases all locks:

### The with Statement

The ESRI help describes a second technique for writing cursor scripts to release locks using the Python with and as keywords.

When scripts are run within the ArcGIS Python Window, the *with* block automatically deletes data locks when the code exits the with block, even if there is an exception.

Though this technique also works for arcpy cursors when code is run inside the ArcGIS Python window, this does not work for stand-alone scripts.

The format for using a *with* block for an *arcpy* cursor is as follows:

## Update Cursors

Update cursors are not only able to read field values, but can also modify them.

Update cursors share some commonalities with search cursors.

They have the same required and optional parameters (See Table17.2 ).

They can both be used in a loop and in a with block.

Like the search cursor, the update cursor has next and reset methods (See Table17.3 ).

The next method returns a list instead of a tuple.

Recall that Python lists are mutable, unlike Python tuples.

Using a Python list during update operations makes it possible to modify individual elements of the row.

Update cursors additionally have updateRow and deleteRow methods for modifying records.

Examples17.3 and17.4 demonstrate these methods.

Example17.3 increments the FireType\_P by 2 and standardizes the FireName capitalization scheme.

It performs multiple updates within the same loop by modifying the corresponding row elements.

Notice that there are two steps to updating a table row.

The first step changes the value of the row list.

This just changes the value of a Python variable, but this does not change the value in the table.

The second step changes the table value with the updateRow method.

## Insert Cursors

Use insert cursors to insert new rows.

Like the other cursor functions, the InsertCursor method requires two arguments, an in\_table and a field\_ names list.

The code should create the cursor withfield\_names set to list the fields that you want to specify in the new row.

Not all fields need to be specified.

Fields with no values specified will be assigned a default value.

Then there are two steps to inserting a new row.

First, create a Python list of the values to insert in the row.

The list should be the same length as the field\_names parameter list and specify the value for each field in the order of this list.

Second, to insert the row, call the insertRow method.

After the insertion, the cursor must be deleted to release locks.

The code in Example17.5 creates the last row in Figure17.3.

### Inserting Geometric Objects

The rows added in Examples17.6 and17.7 appear in the tables, but no new points appear because we have not specified locations.

To set the ‘Shape’ field, the script must create the appropriate arcpy Geometry objects.

The arcpy package has functions named Point ,Multipoint ,Polyline , and Polygon for creatingGeometry objects.

The following code creates an arcpy Point object with x = -70.1 andy = 42.07 and stores it in the variable named myPoint :

Multipoint, Polyline, and Polygon features are composed of multiple points.

These more complex shape types require the user to create an Array object and then pass this to the Geometry object to create the feature.

The following code creates a Polyline connecting points a and b:

Examples17.7 and17.8 useGeometry objects, along with insert cursors, to insert features.

Example17.7 inserts a point.

It uses theSHAPE@XY token since it is only specifying a point to be added.

The script creates a Point object and uses the point to defi ne the new row.

The new row is a Python list variable, as usual.

The order of the values in the new row list corresponds to the order of the fi elds in the field\_names parameter list when the cursor was created.

In other words, the FireId value, of 500000, is placed first and theShape object is placed second.

There is nothing special about this ordering except that these two lists must use the same ordering.

Finally, the new row is added to the table.

Prior to the call to the insertRow method, no changes were made to the attribute table.

Once this call has succeeded, the new point is added.

Figure17.4 shows this point highlighted when the new row is selected in ArcMap.

Example17.8 inserts a polygon.

It uses theSHAPE@ token to get the entire Geometry object, since it is inserting a polygon instead of a line.

Inserting any feature more complex than a point requires this because SHAPE@XY is the shape’s centroid.

The script creates three points and then creates an Array object by passing it a list of the points (myArray = arcpy.Array([a,b,c]) ).

Then the polygon is created from the Array object (the polygon is created with this line of code:poly = arcpy.Polygon(myArray) ) and the Shape field is set using that polygon.

In this example, the geometry token is given first in the field\_names parameter list, so it is listed first in the new row.

The new row is inserted, which adds the triangle highlighted in Figure17.5.

## Selecting Rows with SQL

The SearchCursor and UpdateCursor functions have an optional where\_ clause parameter that can be used to refi ne a selection of features, such as selecting the features in ‘fires.shp’ which have a class size of A .

This can be accomplished by looping with a nested condition, but the cursor is optimized to make this selection faster, which can be important for large fi les.

Just like the where\_clause parameters in ArcGIS tools that make selections, these expressions must be specified as SQL statements that use SQL comparison operators.

The following expressions are examples of where\_clause values that could be used to select rows in the ‘fires’ shapefile data:

The same formatting described in Section 9.2, ‘ArcGIS Tools that make selections’, applies here.

We can make a few observations from these examples.

The first two expressions involve text fields so they use the single quotations around the field value.

The SQL inequality operator (<> ) is different from the Python one we use (!= ).

Only one pair of quotation marks is used for numeric fields.

Fields other than text and numeric types, such as the date field used in query4, require specialized syntax.

Read more in the ArcGIS Desktop help topic, ‘SQL reference for query expressions’.

To select a subset of rows, create the cursor and use a where\_clause query as the third parameter.

Example17.9 usesquery1 to print the years for the records when the fi re size class is‘A’ :

TheSizeClass fi eld is used to specify a selection (those rated‘A’ ) and the CalendarYe is printed for these seven records.

The printed output looks like this:

Queries can be formed without hard-coded field names, by using string formatting.

The query used in Example17.10 will have the same effect as query3 if the user passes FID as the second script argument.

This query would select the row with FID greater than 6.

A list comprehension is used to create a list of the values of fieldToPrint .

When this script is run with the example input, the results look like this:

## Key Terms

Cursor

Record

arcpy DataAccess (da) module search, update, and insert cursors

Search and update cursor next and reset methods

Cursor fields property

Update cursor updateRow and deleteRow methods

Insert cursor insertRow method

arcpy geometry tokens

arcpy Geometry objects

arcpy Point ,Array ,Polyline , and Polygon methods

## Exercises

# Dictionaries

Abstract

In addition to integers, floats, strings, and lists, Python has a powerful built-in Python data type called a dictionary.

Dictionaries are useful for storing tables of information with a unique identifier for each record.

Dictionaries provide a mapping from a set of keys to a set of values.

In other words, given a key, a dictionary can look up the value associated with it.

These have many applications in GIS, including reading GIS attribute tables or text data fi les and modifying them within a script.

Also, they are often used to store pairs of items that go together.

For example, soil science uses standard classifications for soil, abbreviated with terms such as ‘Ap’ and ‘Cg’;

However, more explicit names such as ‘Plinthic Acrisol’ (for ‘Ap’) and ‘Gleyic Chernozem’ (for ‘Cg’) are needed for some analysis.

A dictionary can be used to store these terms so that the abbreviations are associated with the full names.

This chapter shows the dictionary syntax for creating associations like this and then it shows how to access, update, and modify dictionaries.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Create Python dictionaries.

• Add an item to a dictionary.

• Modify or delete an item from a dictionary.

• Check if a dictionary has a key.

• Explain KeyError exceptions.

• Replace IF/ELSE IF structures with dictionaries.

• List the keys, values, and items in a dictionary.

• Loop through a dictionary.

• Populate a dictionary based on user input.

• Populate a dictionary using arcpy cursors.

• Embed Python lists as values.

## Dictionary Terms and Syntax

The terms used to refer to the elements in a Python dictionary are ‘items’, ‘keys’, and ‘values’.

Python dictionaries store a collection of items; Each item consists of a key and a value.

An item is like a GIS attribute table record.

Each record has a unique identifying attribute, such as its FID value, which allows you to access the rest of the values in that record.

A Python dictionary item is a unique identifier, a key, along with the related value.

The following code creates a Python dictionary containing five items:

The items in the dictionary are zipcode-county pairs; Each key is a North Carolina zip code and each value is the enclosing county name.

Keys must be unique, but values can be duplicated.

For example, two zip codes fall within Wake County.

This is consistent with English language dictionaries;

The words defined in an English dictionary must have only one entry, but synonyms have the same definition.

Dictionary keys are most often strings or numbers.

Mutable types such as lists or dictionaries cannot be used as keys.

However, the item value can be of any data type, including a mixture of numbers, strings, lists, and so forth:

The definition of a dictionary can be distributed across multiple lines.

When multiple lines are used to assign a dictionary in the Interactive Window, three dots appear at the beginning of the lines:

The triple dots appear automatically if the dictionary extends across multiple lines in the Interactive Window to indicate that the lines are grouped together until the closing curly brace.

Python reports the data type of Python dictionaries as type'dict' :

Dictionaries can have zero or more items.

The syntax for an assignment statement for a dictionary that has three items looks like this:

Spacing doesn’t matter, but the punctuation is required.

Each key is followed by a colon and then its corresponding value.

The pairs are separated by commas.

The curly braces indicate that it is a dictionary, just as square braces are used to create lists.

Also similar to lists is the syntax for creating an empty dictionary.

If dictionary items are not known ahead of time, you can create an empty dictionary and then add items to it.

To create an empty dictionary, set a variable equal to a pair of empty curly braces:

### Access by Key, Not by Index

The similarities between list and dictionary syntax can be misleading.

Dictionaries store sequences of items, so you might assume you can try to index into a dictionary like a list or a string.

But dictionaries are not designed to work that way.

Instead, values need to be accessed using keys.

The following code assigns a dictionary of student ids and dormitory room numbers ('emforste' and 'pgwodeho' are roommates):

### Conditional Construct vs. Dictionary

A dictionary can sometimes be used to replace a multi-way conditional construct, one with multiple serially checked conditions.

When a selection statement is control ling flow based on the value of some variable and actions taken inside each branch are based on the value of a second variable, the pairing of these values can be replaced with a dictionary.

Some programming languages use a switch statement for this purpose; Python uses dictionaries instead.

Take the following code as an example:

### How to Modify: Update/Add/Delete Items

Dictionaries can be modified by updating existing items, adding new items, and deleting items.

Each of these three actions access the dictionary using a key in square braces.

Modifying mutable and immutable item values is handled slightly differently.

When the item values are immutable data types like strings or numbers, the syntax for modifying an item and adding a new item is identical:

## Dictionary Operations and Methods

Like strings and lists, dictionaries have specialized operation and methods for dictionary operations.

This section focuses on four frequently used operations and methods: thein keyword, keys, values, and items.

### Does It Have That Key?

Thein method can check if a dictionary has a particular key to avoid overwriting an existing key or to avoid throwing a KeyError .

Thein keyword can be used on keys in a manner similar to how it works on lists.

It returns True or False .

To check for the keys 'cslewis' and 'cslouis' in the dormitory room dictionary, use the following code:

### Listing Keys, Values, and Items

Like other strings and lists, dictionaries have specialized methods for dictionary operations.

Certain dictionary methods allow you to list the keys, values, and items.

keys returns a list of the keys in the dictionary in arbitrary order:

### Looping Through Dictionaries

The listing methods can be used to loop through a dictionary.

Typically, you won’t need to store the lists returned by these methods, since they are already stored in the dictionary; In this case, you can just use the method call as the sequence that comes after the in keyword in a FOR -loop.

For the keys and values method, place a single variable between the for and in keywords.

The variable iterates through the keys or values:

## Populating a Dictionary

The examples thus far have hard-coded dictionaries to introduce dictionary syn tax, but you’ll often want to generate dictionaries dynamically.

This section shows how to populate a dictionary to store data from user input.

As a starting point, the fi rst examples use a hard-coded set of keys and the built-inraw\_ input function to gather values.

The remaining examples generate the entire dictionary dynamically, using directory listings orarcpy cursors to retrieve data from attribute tables.

The script ‘healthyLiving.py’ in Example18.2 surveys user preferences.

The script starts with an empty dictionary and uses an assignment statement to add new values.

An empty dictionary favDict is created, then the script loops through the topics, asks about each topic in turn and populates the dictionary inside the loop.

### Dictionaries and Cursors

Another common use of dictionaries is to store GIS data gathered by arcpy cursors.

Example18.5 finds the median area of the polygons in a shapefile and then determine the IDs of any polygons whose areas are close to the median area.

The script collects the polygon areas of a shapefile in a dictionary.

Then the built-in numpy module is used to calculate the median of the dictionary value list.

A final loop through the dictionary items identifies the polygons with area measurements in the range of plus or minus 400 square feet of the median area.

## Discussion

Dictionaries provide a way to store a mapping between one set of values and another set of values.

Like strings, lists, and tuples, they are a built-in Python sequence data type that has methods and special operations.

Items are accessed via a key, not by indexing.

The 'dict' dictionary data type is not ordered (though Python does have other variations of dictionaries that are ordered).

A dictionary can be hard coded or populated dynamically.

The syntax for adding items uses an assignment statement with the dictionary name and key in square braces on the left and a value on the right.

If the key is not already in the dictionary, a new item is added.

If, on the other hand, the key is already in the dictionary, the item with this key is overwritten.

Dictionaries with value lists are often useful.

In this case, list syntax and methods must be used to modify the lists.

Common applications of lists include replacing multi-way conditional constructs (if/elif/elif...) and storing data collected from user input or with arcpy cursors.

Care must be taken to use thein keyword as appropriate when populating a dictionary, since overwriting can occur.

Handling these situations depends on the application.

If the input already has unique keys (such as FIDs) no checking may be needed.

But if collisions can occurs, thought should be given to how to handle these.

Should the script keep the first instance of that key? Should it keep the last one? Should it use a list and store every occurrence? The examples in this chapter can be used to help with these decisions.

## Key Terms

'dict’ data type

Key

Value

Item

Access by key vs. indexing multi-way conditional construct

Dictionary methods (keys, values, items)

numpy module

## Exercises

# Reading and Writing Text Files

Abstract

GIS data often comes in formats that can’t be imported into ArcGIS or read with arcpy cursors without modification.

Automating these modifications can be a significant time saver if you have numerous fi les that require the same changes.

The arcpy cursors discussed in Chapter 17 are specialized for reading and writing attribute tables that conform to a prescribed format, such as dBASE files, shapefiles, and GRID rasters.

This chapter discusses Python file handling functions that can read and write text fi les, regardless of format.

The chapter begins with Python file handling syntax and then uses GIS examples to demonstrate common tabular text file modifications.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Write and read text files.

• Use the Python with structure for reading and writing.

• Separate the parts in a line of a tabular data file.

• Modify a tabular data file.

• Differentiate between the Python working directory and the arcpy workspace.

• Handle IOError exceptions.

• Avoid file locking.

• Write a fi le that preserves Python data structures.

## Working with file Objects

The first step to reading or writing a fi le is to call the built-in open function which returns a file object.

The' file' data type has a number of methods for reading and writing a fi le.

The following line of code creates a file object named 'f' that allows you to read a file:

The open function requires one argument, the name of the fi le.

The file object can be created for reading, writing, or both.

The second argument is an optional argument for setting the mode.

The mode describes how the fi le will be used: 'r' for reading text, 'w' for writing text, 'r+' for both reading and writing ASCII text files.

Other modes can be used to handle binary (non-text) file formats, but the examples in this chapter focus on reading and writing text fi les ('r' and 'w'), as these are the most common usage scenarios.

Depending on the mode, the file object returned by the open function has either read or write methods.

In the code above, ‘poem.txt’ was opened in read mode.

There are several methods for reading all or part of a fi le (See Table19.1).

The read method reads the entire file and returns it as a string:

Run the code above, then browse to ‘poem.txt’ and open it to view the contents in a text editor, such as ‘Wordpad’.

The fi le contains five lines of text, a Pythonic verse inspired by Lewis Carroll’s poem, ‘Jabberwocky’.

The string printed by Python contains a carriage return escape sequence ('\n') at the end of each line of text.

Add another line to the poem and try to save the fi le.

If you have run the fi le opening code, you will receive a message that the fi le is in use by another application and it cannot be saved.

Like arcpy cursors, Python file objects lock the file.

To release the fi le, you must use the close method.

There should be one close statement for each open statement to prevent locking issues.

The close method does not require any arguments, but parentheses must be used, since it is a method.

Use the following statement to close the file object and release the lock:

Similar to the process of reading a book (open->read->close), when reading or writing a file, the code should open the fi le, read or write, and then close the file.

If a book is left open, it can be splattered with soup.

If a file is left open, it will be locked and can cause errors or data corruption.

### The WITH Statement

The WITH statement is an alternative to closing the file explicitly.

The usage of WITH statements for file objects is similar to the usage described in Chapter 17 for arcpy cursors.

Place the open function call between the with and as keywords; Place the file object variable name after the as keyword; Complete the statement with a colon.

Indent the code that deals with the file object.

The file object is only usable inside this indented code block.

Dedented code signals the end of the block.

The following two lines of code, are equivalent to the following three lines of code, except that the with block ensures that locks are released even if an exception is thrown while attempting to read the fi le.

When a with statement is used, the fi le close method does not need to be called.

Upon exiting the with block, the fi le is automatically closed.

Attempting to call the read method outside of this block returns a value error, since the file is automatically closed when the code exits the with code block:

The open->read->close approach is handy for Interactive Window examples since the read feedback is immediate and the explicit close statements make the scope of the file object obvious.

Otherwise, the with statement is a good technique.

This chapter uses both approaches.

### Writing Text Files

### Safe File Reading

### The os Working Directory vs. the arcpy workspace

When you’re writing geoprocessing scripts and setting the arcpy workspace, it’s easy to make the mistake of assuming this will work for Python file objects as well.

The built-In open function is not part of the arcpy package, so the full path file name may need to be used even if the arcpy workspace is set.

For example, the following code sets the arcpy workspace and then attempts to open ‘poem.txt’ which resides in the workspace:

### Reading Text Files

The read method returns the entire contents of the fi le as a string with lines separated only by carriage return characters.

For many applications, you may want to read each line, one at a time, using the readline method or a FOR -loop.

The readline method reads the next line in the fi le.

When the file is first opened, the “next” line is the first line of the file:

Calling readline followed by a FOR -loop is often used to handle header lines separately from table records.

This approach is used again in upcoming examples on handling fi le content.

This section used read, readline, and FOR -loops to read a file.

Though other approaches exist, these are sufficient for the most common applications.

Table19.1 summarizes these methods and variations.

## Parsing Line Contents

Once you read a line of a text fi le with a script, you need to parse the contents, in other words, break them down into usable parts.

Each line is returned as a single string with a trailing carriage return.

Depending on the text fi le contents, parsing the text may involve stripping trailing whitespace, splitting the string into a list of items, casting the string to a numeric value and so forth.

Several common parsing operations are demonstrated here.

Suppose, for example, you want to find the sum of a line in ‘report.txt’.

Though the fi le contains tab delimited numbers, the readline function always returns a single string.

In the following example, readline returns a string containing numbers and escape sequences:

The tabs appear as whitespace when the print statement is used because it interprets them, but the line variable is a string data type containing all the numbers and tabs within one string.

Now that the line contents are stored in a variable, you can use the string split method to separate the numbers.

By default, the split method separates the string contents at the white spaces, so no argument is needed.

The following code splits the values into a list:

### Parsing Field Names

Example19.2 hard-codes the index of the fields (0 and 1) in the split line containing those fields.

If the field name is known, but the position of the column is not, you can use the list index method, which returns the index of the first occurrence of a value in the list:

The ‘fieldIndex.py’ script in Example19.3 calls a getIndex function that strips the trailing whitespace from the field names string, splits the string into a list, and then finds the index of the field based on its name ‘Label’.

Since ‘Label’ is the second field in the ‘cfactors.txt’ data fi le, the script finds that the field named ‘Label’ has index 1.

## Modifying Text Files

For many applications you will need to modify files.

Reading and writing to the same file using 'r+' mode is not usually the best solution for this, because managing existing content and modifications simultaneously can be complicated.

A simpler approach is to read the original file, make modifications to the content within the script, then write the modified content to another file.

The script can then replace the original file with the new one, if desired.

This flow would be as follows:

### Pseudocode for Modifying Text Files

### Working with Tabular Text

Some common modification tasks, such as removing lines or columns, take a slightly different approach.

To demonstrate these functions, we’ll use the raw output from a device that tracks eye movement and records the points on a map where the eye fixates.

The raw eye tracking data is not designed for use in ArcGIS, so it needs to be modified for import into ArcGIS and visualized as a shapefile.

Some entries need to be deleted.

The arcpy update cursor function has a deleteRow method for this purpose, but when the data is not ArcGIS compatible this method can’t be used.

The Python native file objects do not have a delete function.

However, deletion can be achieved by omission.

The usual flow is to read a line of the input fi le, modify the line, and then write the line to the output fi le.

For lines that are to be deleted, you can read the line from the input, but not write it to the output.

The internal position cursor needs to read each line in the input file to step through it methodically, but not all the lines that are read need to be written to the output!

ArcGIS expects the first line of a table to be a set of field names;

However, tables are often preceded by metadata, so removing header lines is a common task.

The first two lines of the ‘eyeTrack.csv’ file contain metadata about the conditions under which eye movements were tracked.

These lines must be deleted so that the MakeXYEventLayer management tool can be used to bring the points into ArcGIS.

To strip these lines from the modified file, ‘removeHeader.py’, in Example 19.5 simply reads these lines, but does not write them to output.

Deleting header lines boils down to reading but not writing the first few lines to the output file.

Sometimes other lines need to be deleted.

For example, we also need to remove lines in ‘eyeTrack.csv’ where the fixation point-of-gazexory values (FPOGX and FPOGY fields) are not positive;

These represent invalid readings.

The pseudocode for removing delimited records based on a condition is as follows:

In short, you check the condition and only write the line when the condition is true.

Example19.6 shows a code snippet from sample script ‘removeRecords.py’.

This code only writes table lines to the output fi le that have values greater than zero for the fields FPOGX and FPOGY.

This has the effect of deleting the lines with non positive values for either of these fields.

This script calls the getIndex function defined in Example19.3 and then when it encounters each line, it splits the line, and gets the float values of the columns of interest using the field indices.

It then checks the required condition (positive for both columns) and writes the line only if the condition is met.

Several columns in the eye tracking data contain non-numeric values.

These col umns need to be removed entirely.

Since thefile object reads each line individually and the tabular line contents can be split into lists, dealing with columns involves list operations.

To remove columns, you need to remove the corresponding item in each line’s list.

ThegetIndex function defi ned in Example19.3 can be used again to determine the index of a column.

Then the stringpop method can be used to remove that index:

The pop removes the item with the specified index.

It returns the value of the item it removed, but we’re not using that value in this application, so we can just discard it.

Instead, we are using the resulting list which has one less item.

Care must be taken to remove the intended items.

When pop is used, the indices of the rest of the items coming after that one in the list are decremented by one.

Trying to pop indices 2 and 4 in that order results in an index error:

Eight columns in the ‘eyeTrack.csv’ file need to be removed.

Example19.7 lists these fields in the removeFields variable.

This code snippet from sample script ‘removeColumns.py’ reads the line containing the field names, determines the indices of these unwanted columns, then calls are moveItems function.

This function sorts the indices in reverse order and then pops the unwanted values off a list that is derived from a delimited string.

The removeItems function is called once for the field names line and then it is called again for each record in the table.

The function splits the line string into a list, removes the unwanted items in the list and then rejoins the modified list into a delimited string.

This shortened string is returned to the caller and the script writes the modified line to the file.

You can combine functionality such as removing lines and removing columns of a tabular text file within a single script by working from top to bottom in a text file, one line at a time, parsing and modifying data along the way.

The ‘eyeTrackSHP.py’ sample script in the exercises deletes rows and columns and performs a transformation from an eye tracking reference system to a geographic reference system.

In Figure19.1, the eye fixation point shapefile is overlaid on the world map that the viewer was viewing as the eye movements were recorded.

## Pickling

The standard file object write function requires a string argument.

Suppose your scripts generate a number of dictionaries and lists that you want to save in a file for further analysis.

This means converting other data types before writing them to file.

For example, to write a number in a file, you need to first cast it to a string;

To write a list to a file, you need to represent it as a string by joining the elements.

If instead, you want to preserve the original data types within a file, you can use a built-in Python module named pickle.

Pickling allows you to write and read any Python data type.

The file itself contains information that encodes the data type.

When opened in Wordpad, the encoding doesn’t look the same as the Python syntax for these types, but Python can read them using the pickle module.

The pickle methods for writing and reading are named dump and load.

The dump method can be used when a fi le is opened for writing.

It takes two arguments, the object to be written, which can be any data type, and a file object created in write mode.

The following code dumps a float and then a list into the ‘gherkin.txt’ file:

## Discussion

This chapter discussed file object methods and techniques for parsing and modifying files.

file objects have additional methods, such as seek and writelines , but most file handling can be accomplished using only the read ,write ,read line , and FOR -loop methods discussed here.

Using the WITH statement to open fi les provides a convenient protection against locking.

If a WITH statement is not used to open the fi le, the close method must be called before the fi le is released.

If an exception occurs before the close method is called, the fi le will remain locked and you may need to type a call to the close method in the Interactive Window.

The close method can be used along with the WITH structure, but it is not necessary.

The built-in file object read functions always read fi le content as strings and fi le content must be written as strings.

This means that most of the parsing and modification work in fi le handling involves string operations—casting when numbers are involved and converting to lists when the lines are part of a table.

Pickling is a useful alternative for writing fi les that are only meant to be read by Python;

Whereas, generic Python file objects provide techniques for manipulating the data into a format that can be imported into GIS software.

## Key Terms

The' file' data type

The built-in open function

The file read, readline, write, and close methods

The WITH statement for opening files.

The for line in f reading approach

The IOError exception

Current working directory vs. arcpy workspace

Parse

The open/read/parse/modify/write/close/replace workflow

The shutil module

The list pop method

The pickle module

## Exercises

# Working with HTML and KML

Abstract

GIS tasks often involve working with HTML and KML files.

You may need to generate an HTML page showing the results of some GIS analysis or you may need to parse the data attributes in the description that pops up when you click on a KML element in Google Earth.

HTML and KML files are simply text files that use tags to delineate elements within the data.

Chapter 19 discussed how to read and write text files with Python;

This chapter explains some additional techniques for working with HTML and KML formats.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Identify basic HTML tags, such as links, images, and text formatting.

• Set HTML tag attributes.

• Create an HTML fi le with text formatting, links, and embedded images.

• Identify KML tags that define geographic features.

• Write HTML and KML with Python.

• Download and save Web site contents with Python.

• Extract fi les from Zip and KMZ archives.

• Parse HTML and KML with the Python.

• Convert KML files to ESRI shapefiles with Python.

## Working with HTML

HTML, which stands for Hyper Text Markup Language is a popular format for Web pages.

This section provides a brief introduction to the format of these fi les.

Readers who are interested in learning more should search for interactive HTML tutorials online which allow you to enter HTML code and view the results.

HTML is a language for encoding web content.

Files containing HTML code have ‘.html’ or ‘.htm’ file extensions.

HTML is not a programming language like Python, but rather a language that uses tags to create web pages.

A tag is a set of characters surrounded by angle brackets, such as<html> and<body>, as shown in Example20.1.

This simple HTML file is named ‘elephant1.html’.

### Specifying Links

### Embedding Images

### HTML Lists

### HTML Tables

### Writing HTML with Python

HTML can be used to present GIS analysis results.

Since HTML is simply a text file containing HTML tags, you can use Python fi le handling to write HTML.

There are specialized packages for writing HTML which are not covered in this book;

Instead, the upcoming examples show how to do basic HTML file writing with standard Python file handling, familiar from Chapter 19.

The Python script in Example20.2 hard-codes a string variable mystr with HTML code, opens a file for writing, and writes the string to fi le.

The triple quotes are wrapped around the string literal value to preserve the line breaks in the HTML code.

As the HTML file becomes more complex, breaking the HTML code into three parts (beginning, middle, and end) can make it easier to manage the string contents.

Example20.3 breaks the HTML code from Example20.2 into three parts and writes each part to file.

Example20.3 also uses string formatting to dynamically generate the HTML content based on user input (the resulting Web page is shown in Figure20.4).

### Parsing HTML with BeautifulSoup

By reading and parsing markup language content with Python, you can access linked or embedded GIS data.

The previous section showed that HTML pages can be generated using Python file objects in write'w' (write) mode.

HTML can also be read with Pythonfile objects in'r' (read) mode; However, the challenge comes in deciphering the HTML, i.e., parsing the content.

To simplify this process, it’s worth learning a few things about a module that supports markup language parsing.

Python does have a built-in module for HTML parsing (named HTMLParser ), but the non-built-in module namedBeautifulSoup , in refer ence to a Lewis Carroll verse, is easier to use for high-level tasks such as fi nding all the links in a page.

The BeautifulSoup module has methods and objects for reading tags and their attributes and content.

Online documentation explains how to download and install the latest version of BeautifulSoup , but for consistency, a stand-alone version which consists of just one Python fi le named ‘BeautifulSoup.py’ is included with the sample scripts for this chapter.

Use this module while you’re learning the basics, so that you don’t get caught up in the installation procedures.

Code may need minor changes to use the latest installable versions.

The given module doesn’t require any special installation to be imported, but since it is not a built-in Python module, you have to make sure Python looks in the right directory.

Handle this just as you handle importing user-defined modules.

To import BeautifulSoup , first import the sys module and append the path for BeautifulSoup to the path list, then use a separate import statement to import BeautifulSoup :

## Fetching and Uncompressing Data

Python can also enable you to fetch web content (retrieve it from a Web site), so that you can automatically harvest online GIS content that is made available through Web site links.

### Fetching HTML

### Fetching Compressed Data

### Expanding Compressed Data

## Working with KML

Now that you know something about HTML and writing/reading it with Python, you’ll also be able to work with other markup languages.

Keyhole Markup Language (KML) fi les are of particular interest for GIS work, since this is a markup fi le format for geographic information.

Like HTML, KML uses tags wrapped around content to encode data in a text fi le.

KML fi les contain tags relating to places with geographic positions.

They are designed to be viewed in Google Earth (Figure20.5) and other GIS.

ArcGIS can import these datasets into shapefiles for geographic analysis, using the ArcGIS KML to Layer (Conversion) tool, but this tool does not necessarily import all of the information encoded in the fi le.

Geographic features are imported, but important attributes may not be imported.

Since KML is a markup language, Python scripting with BeautifulSoup can be used to solve this problem and parse the data.

Then ArcGIS cursors can be used to build tables from the parsed data.

### The Structure of KML

### Parsing

### Converting KML to Shapefile

## Discussion

This chapter briefly introduced two markup languages, illustrated writing and reading markup files with standard built-in Python file objects and functions, and showed how to use Python to fetch and consume Web content.

The BeautifulSoup module for parsing markup languages was used to parse HTML and KML content.

Finally, insert cursors were used to import parsed KML into a shapefile.

One difficulty in parsing markup fi les is that there can be errors in the markup code.

Also, since the content in markup files varies, writing code to step through with the debugger to watch soup Tag objects and their contents is a pragmatic approach for developing scripts with BeautifulSoup.

Once you see the tag contents (such as the KML description tag list contents), you can refi ne your code to consume the information you need.

This code may need to be highly customized to handle your data, which explains the difficulty in creating a general tool that handles any HTML or KML content.

Though this chapter only used HTML and KML examples, these approaches can be adapted for other markup language files, such as XML files.

## Key Terms

## Exercises

# Classes

Abstract

Classes are the central construct for object-oriented programming (OOP).

Knowing how to design your own objects will deepen your understanding of Python and help with learning other OOP languages.

Object-oriented programming requires a paradigm shift from functional programming, which essentially works by grouping related steps into reusable functions.

In functional programming, to make an omelet, you get six eggs and repeat break\_egg until all the eggs are broken.

In contrast, OOP revolves around the objects involved in the problem.

Methods and properties belong to object classes;

Making an omelet involvesEgg objects.

The Egg objects can have a shell property and a break\_shell method and you could make each egg object break its shell.

It may sound strange at first, but thinking about objects you have already been using will help.

When you create a list, you’re creating an instance of a list object.

Then you apply list methods to it, such as, mylist.sort() ) .

Similarly, you have used geoprocessing objects, such as Describe objects which have properties such as dataType and dataPath .

Throughout this book you have used objects, methods, properties, and dot notation.

All of these are aspects of OOP.

Up to this point, you haven’t designed your own objects;

In this chapter you’ll learn the syntax to do so.

The examples in this chapter demonstrate basic OOP syntax and some of the benefits.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• List some benefits of object-oriented programming.

• Explain the terms ‘object’, ‘class’, ‘property’, and ‘method’.

• Define a class and create an object instance.

• Create class properties and methods.

• Modify object properties and call object methods.

• Call a class defined in a user-defined module.

## Why Use OOP?

User-defined functions increase code reusability.

However, they have their limitations.

Functions don’t store information like variables do.

Every time a function is run, it starts afresh.

Functions and variables are often closely related to each other and need to interact with each other.

As an example, suppose you are studying park maintenance and utilization using a dataset of hiking trails.

Each trail has an identification number.

You could store the trail IDs in a list as follows:

As the database is updated, you may need to modify the trail list.

To do so, you could append or remove IDs from the list:

However, each land trail has additional information associated with it, such as the vegetation classification for the trail.

How can you keep track of the vegetation?

You might think of using a dictionary of trail IDs and vegetation classifications, as in the following example:

Then you might need another dictionary for the trail lengths, and other dictionaries for the trail surfaces, the benches, the trail maintenance organizations, the manmade structures near each trail, the grade, and so forth.

Functions need to access various trail data by passing in arguments and using the trail IDs to identify the value of each attribute for that trail.

For example, a function calculating the costs for maintaining a trail needs to know the trail length and vegetation classification.

The function, calculateCost, defined in Example21.1 passes these as arguments.

An upcoming example will show a different way to do this using OOP.

As the code grows, it becomes difficult to keep track of which functions need which variables and how everything relates.

With all of these variables separate but depending on the same set of properties, maintaining the variables correctly can grow cumbersome.

Suppose you need to remove the trail with ID number 5.

Each variable that references that trail must be modified to reflect the removal.

The trail 5 entry for the vegetation dictionary, the length dictionary, the volunteer dictionary, and so forth must all be deleted.

This process could be made easier by grouping these trail attributes with the trail.

Then when a trail needs to be added, updated or removed, the trail information is all together in one place.

Object-oriented programming does this by grouping variables and related functions so the functions can act on the variables and interact with each other more smoothly.

Classes group closely related functions and variables together.

They also provide a convenient way to work with a group of objects that have common attributes, such as a set of trails that each have a length, a vegetation classification, and a volunteer organization.

A class allows you to design a basic trail and each time you need to create a new trail, you can just specify the values of its attributes.

This concept of grouping functions and variables related to a particular type of item is the key principle of OOP.

Classes are the container for these related functions and variables.

You design objects by creating an object template, called a class.

Which acts as a blueprint for the object.

The class specifies functions and variables associated with the object.

In this way, the class structure provides a means for grouping related functions and variables.

## Defining a Class

To defi ne a Python class, use the class keyword followed by the class name and a colon.

The contents of the class are indented to indicate that they are a related block of code.

Python class names usually use upper camel case capitalization.

The format for a Python class is as follows:

Example21.2 shows a Trail class definition.

Think of a class definition as a blueprint.

It isn’t creating an object itself;

It simply describes how to make one.

You can create multiple instances of objects from the blueprint.

The Trail class has three properties, ID, length , and vegetation and it defines three functions or methods .

The term ‘method’ is used to refer to functions that are defined within a class.

The Trail class contains the methods, \_\_init\_\_ (a special method), calculateCost, and reportInfo.

The argument list for all three of these methods starts with a special variable named self.

Next we will discuss how to create an object from the class blueprint and the special role of the \_\_init\_\_ method and the self variable in creating objects.

## Object Initialization and Self

Once the Trail class is defined, you can create a Trail object, a specific instance of a Trail with a particular ID, length, and vegetation classification.

The syntax for creating an object instance looks similar to calling a function that returns a value.

You call the class, using the class name followed by parentheses containing arguments and store the return value in a variable, as in Example21.3.

If you attempt to run Example21.3 without running Example21.2 beforehand, you will receive a NameError, since Trail is not yet defined.

When you call a class, the \_\_init\_\_ function inside the class is automatically invoked.

The \_\_init\_\_ method for the Trail class sets the values of self.ID, self.length , and self.vegetation .

The terms \_\_init\_\_ and self need some additional explanation.

* \_\_init\_\_ is a special method which is automatically called when an object of that class is created.

It is used to create an initial state for the object (similar to a constructor in other languages such as C++ and Java).

Use one of these inside your class to initialize object properties.

Note that the name \_\_init\_\_ requires two underscores on the front and two on the back; otherwise, it will not be recognized as the initialization method.

* self, as the word suggests, refers to the particular instance of the object itself.

This means that when you set self.vegetation with an assignment statement inside of \_\_init\_\_ , the vegetation property for the current object is initialized.

You should use self as the first parameter in the list of any function defined within a class.

This allows you to refer to object properties within other methods without passing them as arguments.

For example, calculateCost uses self.vegetation and self.length without passing these into this method.

Though you could use a name other than self as a placeholder for the first position in the parameter list, it is best to conform to convention so that your code is consistent with others.

When a class is invoked, the caller must provide arguments for the argument list in the \_\_init\_\_ method.

When the Trail class is invoked in Example21.3 , the caller provides three arguments.

The argument list should correspond to the \_\_init\_\_ argument list, except for self which is passed implicitly into all class methods.

Since the self variable is passed implicitly, only three, not four, arguments are needed to create a Trail object.

The three arguments, an identification number, a length, and a vegetation classification, must be specified in this order in the class call.

The \_\_init\_\_ method constructs an instance of the object and initializes its properties.

The class returns an object instance, which can be saved using an assignment statement.

Example21.3 saves the returned object in a variable named myTrail.

## Using Class Objects

A class blueprint can define object properties and methods.

An instance of aTrail object has three properties (ID ,length , andvegetation ), and two methods (calculateCost andreportInfo ) in addition to the \_\_init\_\_ method.

Notice the differences between the class method,calculateCost , in Example 21.2 and the stand-alonecalculateCost function defi ned in Example21.1 .

Example21.1 uses three parameters (trail\_ID ,vegetation andlength ); Whereas, Example21.2 only usesself as a parameter.

The class method does not need to use a trail ID number to access its vegetation classification or the length, because the Trail object stores this information about itself.

It can simply refer to self.vegetation and self.length.

Variables that are assigned a value within the class \_\_init\_\_ method asself.variableName (such as self.

length and self.vegetation ) are objectproperties orattributes .

General Python documentation uses the term ‘attribute’, but ArcGIS Resources documentation uses the term ‘property’.

This book generally uses the term ‘property’, for consistency with the arcpy documentation.

This is the same terminology we used earlier to discussarcpy object properties, such as the dataType property of the arcpy Describe object.

Using properties and methods from user-defi ned classes has the same format as using other objects.

Once a Trail object has been created, you can access its properties and methods using dot notation.

The following code creates a Trail object, prints the length and vegetation properties and runs the calculateCost method:

## Where to Define a Class

## Classes Within a Separate User-Defined Module

## Discussion

## Key Terms

The class keyword

Object-oriented programming

Functional programming

A class versus an object instance

Instantiating an object instance

Properties

Methods

The self argument

The \_\_init\_\_ method

## Exercises

# User Interfaces for File and Folder Selection

Abstract

Running a script via an IDE requires some expertise, particularly when the script takes arguments.

When you create scripts to share with others, your target audience may not be familiar with how to run a Python script.

To make it easier, you can create a graphical user interface, a more intuitive means for the user to provide information for the script than command line arguments.

Graphical user interfaces (GUIs) allow the user to interact with the script via windows labeled with explicit instructions.

A Python script that uses GUIs to collect arguments can be run by double-clicking on it in Windows Explorer, avoiding an IDE altogether.

Developing highly sophisticated custom GUIs is an advanced programming skill.

However, you can generate certain GUIs with just a few lines of Python code.

This chapter covers some easy to program GUIs for entering text or browsing to files.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Get user input with graphical user interfaces.

• Get user input to open and save fi les.

• Customize interfaces using optional arguments.

## A Simple Interface with raw\_input

The built-inraw\_input function launches a rudimentary GUI for text entry.

When the script reaches araw\_input command, the user is prompted for input with a dialog box like the one shown in Figure22.1 .

A dialog box is a type of GUI that uses a window to gather information from the user or inform the user of some thing, so called because it facilitates communications (a dialog) between the com puter and the user.

The dialog box in Figure22.1 has two buttons, a label, and a text box.

The label says ‘Enter a workspace’.

The white recessed box is called a text box.

This allows the user to type text which will be returned to the script that called the raw\_input function.

The raw\_input command in Example22.1 is the code that generates the GUI in Figure22.1 .

Theraw\_input function takes one argument, a string message to be displayed as the text box label (the argument in Example22.1 is ‘Enter a work space:’ and this same message is displayed in the dialog box label).

You can place any string message here, but the label is usually used to prompt the user to enter a value.

The response typed by the user in the text box is returned to the script when the user selects the ‘OK’ button;

This response is the return value of the raw\_input function.

In Example22.1 , the return value is being stored in arcpy.env.work space .

When the user types ‘C:/gispy’ in the text box and presses ‘OK’, the value of arcpy.env.workspace is set to ‘C:/gispy’.

The raw\_input function return value is always a string, even if the user enters a number.

You can cast numerical entries (This is similar to handling numeric sys.argv values).

The raw\_input function generates a simple general purpose GUI that can be used in a variety of situations with some limitations.

One limitation is that the user is allowed to enter anything in the text box, so your script may need to anticipate common errors.

For example, if the script casts a number garnered from the user, it can include code to handleTypeError exceptions.

As a second example, there may be a typo in the workspace path.

You can use error handling for this as well, but we’ll discuss a GUI that constrains the input to avert this particular mistake altogether by using a specialized GUI.

File names are such a common GIS scripting input need that it’s worth learning about a few specialized GUIs for this purpose.

The remainder of the GUIs discussed in chapter are designed for handling fi le and directory interactions.

## File Handling with tkFileDialog

This built-intkFileDialog module is one of several tk (or tool kit) modules that provide functionality for interfaces.

The tkFileDialog module functions generate file dialog boxes for browsing to files.

This eliminates the need to type file paths and enforces path accuracy.

The appearance of the fi le dialog boxes is controlled by the operating system.

For example, the screen shots shown in this chapter are generated by Windows 7.

Windows 8 file dialogs may look different, but the functionality is the same.

The file dialogs come in several different flavors for opening files, saving files, and choosing directories.

There are several functions that get fi le or directory names from the user.

Others functions return file objects open for reading or writing.

Optional arguments can be used to control the appearance and behavior of the file dialogs.

The upcoming sections discuss functions for getting file names and file objects, and the optional arguments for these functions.

### Getting File and Directory Names

File paths are a common input for scripts.

The tkFileDialog method *askopenfilename* creates a fi le browsing GUI, like the dialog box in Figure22.2.

When execution reaches the *askopenfilename* call in Example22.2, this dialog is launched.

When the user selects a fi le and clicks the ‘Open’ button, the full path name of the file is returned to the script.

The assignment statement in Example22.2 stores the return value in the fc variable.

The script prints the file name, calls the arcpy Describe method, and uses the Describe object to print the data type of the input file.

Optional arguments for askopenfilename allow the dialog to be customized.

The two options used in Example22.2 are setting the dialog box title and the type of fi les displayed by the browser.

Here we only want the user to select a shapefile, so we specify this as the fi le type—more about these options in a moment.

The asksaveasfilename method is similar to the askopenfilename method in that it returns a file name.

However, since it’s asking the user for a name for saving a file, it warns the user when a chosen name already exists.

The user can cancel or overwrite the existing file.

Calling this method doesn’t actually create the fi le you name.

Rather it returns a string fi le name that the script can use for some output it is creating.

The return value from asksaveasfilename is used as an output fi le name in the following code which makes a copy of the input file:

Directory paths are another input that scripts often require.

Example22.3 uses the tkFileDialog module method, askdirectory, to set the geoprocessing workspace.

The initialdir option sets the initial directory.

The dialog box cre ated by the code is shown in Figure22.3.

The three methods described so far return file path names as strings.

Additional methods discussed in this chapter return file objects.

First though, we’ll take a closer look at the optional arguments.

### Options

tkFileDialog method options allow you to customize the appearance and behavior of the dialog boxes.

These are specified askeyword arguments.

All, none, or a subset of the options may be used.

Throughout this book, we’ve been working with positional arguments.

Positional arguments are ones which must be provided in the order specified in the function signature.

Unlike positional arguments, the ordering of keyword arguments does not matter.

Instead of order, assignment statements are used to specify which options are being set;

The option name goes on the left of the equals sign and the value on the right.

Arguments must still be separated by commas (e.g., the askdirectory call in Example22.3 uses two options, initialdir and title;

The assignment statements for these two options are separated by a comma).

The options for the methods discussed in this chapter are listed in Table22.1.

#### File-Handling Options

Most of the options affect how the dialogs filter the files.

The title option is the only option strictly for appearance; It specifi es a string to display as the title of the dialog box (Compare the titles in Figures22.2 and22.3 to the title option settings in Examples22.2 and22.3 ).

This section demonstrates the behavior of the fi le- handling options, filetypes ,initialdir ,initial file , and multiple for the askopen filename method and the mustexist option for the askdirectory method.

The code snippets in this section assume that the tkFileDialog module has already been imported (The snippets are available in the sample script named ‘fileDialogOptions.py’).

File Types

Initial Directory

Initial File

Multiple Files

Existing Directories

### Opening Files for Reading and Writing

## Discussion

This chapter presented some simple GUI techniques, including theraw\_input function and severaltkFileDialog methods.

Theraw\_input function returns a string version of the characters entered by the user.

ThetkFileDialog method provide several types of functionality.

Several distinctions between the fi le dialog methods determine which method to select for use in a script.

One distinction is between those that return the names of files and those that return file objects.

ThetkFileDialog method names that end with ‘filename’ return the names of the fi les;

These should be used when the script does not need to open the fi le for reading or writing.

Theaskdirectory method also falls into this category (it returns a directory name).

ThetkFileDialog method names that end with ‘file’ returnfile objects open for reading or writing.

When you call these methods, you need to use the file objectclose command or a WITH state ment to avoid locking.

Another distinction is between those that are designed for input and output fi les.

The methods names that start with ‘askopen’ are for getting an input fi le to read or use the fi le contents in some way.

The method names that start with ‘asksaveas’ are for allowing the user to set output fi le names or create an empty fi le for writing.

Table22.2 summarizes the distinctions between thetkFileDialog methods discussed in this chapter.

The GUI implementations in this chapter do have some limitations.

Only one function is called at a time.

To collect both an input directory and an output fi le name, you need to call two methods sequentially and the GUIs will appear in succession.

The raw\_input method can handle any text input, but all values are returned as strings.

When you collect a number withraw\_input you need to cast it.

Also, raw\_input provides no validation.

For example, when you ask for a number, the user can enter something other than a number, so you would have to check that in your script.

When writing code for GUIs, there is a tradeoff between simplicity and customizability.

There are a number of popular Python packages for building Python GUIs, such as Tk, wxWidgets, and pyQT, but these involve a steep learning curve.

ArcGIS provides an alternative middle ground with a relatively easy to learn tool for building GUIs that are more complex than those presented in this chapter, though less flexible than stand-alone custom Python GUIs.

Chapter 23 presents this ESRI alternative using ArcGIS toolboxes to build the interfaces.

## Key Terms

Graphical user interface (GUI)

Dialog box

File dialog box

Text box

The built-in raw\_input function

filename vs.file methods

askopen vs.asksaveas methods

positional arguments

keyword arguments

## Exercises

# ArcGIS Python GUIs

Abstract

The graphical user interfaces (GUIs) discussed in the previous chapter create file browsing interfaces with simple Python calls to the tkFileDialog module.

Some applications may require additional input (other than files/directories).

Or you may want to create one GUI dialog that accepts multiple input parameters.

If the application is meant to be run in an environment where ArcGIS is installed, Script Tools and Python toolboxes provide a solution for building GUIs with these characteristics.

‘Script Tool’ is an ESRI term for an ArcGIS construct that resides within a custom toolbox and points to an underlying Python script.

By using a Script Tool, you can create a custom GUI that looks similar to the built-in ArcGIS tool GUIs.

You can also add a button to one of the ArcGIS menus, so that users can launch the tool with one button click.

Python toolboxes are another way to create GUIs that look just like Script Tool GUIs.

A Python toolbox is a text fi le containing Python classes to define the toolbox and tools.

Script Tools are a good way to learn about the GUI options available; Python toolboxes are an efficient way to develop tools, once you understand these options.

This chapter introduces Script Tools and then steps through the various customization techniques.

Last, Python toolboxes are discussed.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

• Create an ArcGIS graphical user interface for a Python script.

• Specify GIS data type interface parameters.

• Set types, default values, direction, multiplicity, and other properties of parameters.

• Enable a user to input hand-digitized points, lines, or polygons features.

• Create a toolbar button to launch the GUI.

• Add output to a map.

• Set the symbology of an output parameter.

• Display progress as processes run.

• Implement dynamic behavior in a GUI.

• Build a Python toolbox GUI.

• Explain the difference between a Script Tool and a Python toolbox tool.

## Creating a Script Tool

A Script Tool can be thought of as a wrapper for running a Python script;

It points to a Python script, passes user input into the script, runs the script, and receives output from the script.

The workflow for building a Script Tool involves creating a Python script and a toolbox first, then using the Script Tool wizard to set up the Script Tool.

Script Tools reside within custom ArcGIS toolboxes.

A custom toolbox is one created by a user, not one of the built-in ArcGIS toolboxes (In Figure23.1, practice.tbx’ is a custom toolbox).

You create a toolbox, then add a Script Tool to it.

This launches a Script Tool wizard.

The wizard steps you through setting up the Script Tool properties, including a list of parameters which are used to pass arguments to the Python script.

Once created, the Script Tool appears in ArcCatalog in the table of contents under its toolbox.

The icon for a Script Tool looks like a scroll (In Figure23.2, ‘printTextFiles’ is a Script Tool).

### Printing from a Script Tool

### Making a Script Tool Button

### Pointing to a Script

Before we create GUIs with Script Tools, it’s important to understand a few things about Script Tools and their Python scripts.

Figure23.2 shows the Script Tool in ArcCatalog, but if you browse to the same directory (‘C:\gispy\sample\_scripts\ch23\toolboxes’) in Windows Explorer, you see toolbox files, but no Script Tool file.

Script Tools do not appear in Windows Explorer; They are stored as part of a ‘.tbx’ file.

To see evidence of this, check the current file size of the ‘practice.tbx’ file (~6 KB).

Add a dummy Script Tool to the toolbox (right-click on ‘practice.tbx’ in ArcCatalog and choose Add > Script…).

Accept all the defaults (Just click ‘Next’, ‘Next’, ‘Finish’ without browsing to a script, etc.).

Now check the size of the ‘practice.tbx’ fi le again (~7 KB).

It’s larger because it is now storing the additional Script Tool.

A Script Tool is part of the toolbox.

To share a Script Tool with someone else, you need to give them the toolbox; the Script Tool will be visible to them when they view the toolbox in ArcCatalog.

Since the Script Tool is just a pointer to a script, you must also give them the script.

You may be wondering, what it means to say the Script Tool is a ‘pointer to a script’?

And will the Script Tool still point to the script when the toolbox is moved?

To explore the relationship between Script Tools and scripts try some experiments, as described in the following steps:

Take care when you share a Script Tool, so that the user doesn’t encounter Error 000576.

Maintain a relative path between the Script Tool and the script and set the Script Tool to use relative paths.

Open the ‘printTextFiles’ Script Tool properties and select the ‘General’ tab.

The ‘Store relative path names’ checkbox should be checked.

This means we can move the Script Tool as long as we move the script to the same relative location.

Relative to ‘example.tbx’, ‘textPrinter.py’ is one directory up out of ‘toolboxes’ and then one step down into a ‘scripts’ directory.

Remember that moving the Script Tool is accomplished by moving the toolbox.

To see what happens when you don’t maintain a relative path with the script try the following:

## Creating a GUI

Now that you know how to create Script Tools and how the tools relate to underlying scripts, you’re ready to use Script Tools to create GUIs.

When you set up a Script Tool, the last pane of the Script Tool wizard contains a table for parameters.

For the ‘printTextFiles’ example, we left this table empty.

This is why this Script Tool says ‘This tool has no parameters’ when it is launched.

In this section, we’ll use the Script Tool wizard parameter pane to set up some parameters.

This pane contains two boxes, one for the parameters and one for the parameter properties (Figure23.5).

A parameter is added by specifying a display name and data type in the parameter table.

Parameter properties can be adjusted for each parameter.

This section steps through an example that adds some simple parameters.

Sections 23.2.1 and 23.2.2 contain details on parameter data types and properties.

When items are added to the parameter table, the Script Tool generates a widget for the parameters in the list.

User interface elements (e.g., text boxes, buttons, check boxes, combo boxes, and list boxes) are commonly referred to as widgets.

Widgets help the user make input choices by constraining the way input is accepted.

For example, they may help the user browse to a file or select amongst several mutually exclusive choices.

### Using Parameter Data Types

### Using Parameter Properties

Parameter properties provide additional tailoring for the input widgets.

The box at the bottom of the parameters input pane (see Figure23.5 ) controls these properties.

Take a look at the parameter list for ‘01\_optionalParam’ in the ‘propertyExamples’ toolbox (Right-click on ‘01\_optionalParam’ in ArcCatalog, select ‘Properties…’ and then select the parameters tab).

Double click on one of the parameter names in the top box.

An @ symbol appears to the left of its name to show that it is selected:

When you select a parameter in the top box, the parameter properties box updates to display the properties for that particular parameter.

You may not have noticed this, since many data types start out with the same property values.

A few data types do have unique default values auto-populated.

To see this, click on ‘Cell Size’ and then ‘Compression’ in the ‘01\_optionalParam’ parameter list and watch the parameter ‘Default’ value change from ‘MAXOF’ to ‘LZ77’ (these are names of the default algorithms for these data types).

Table23.2 lists the parameter properties with descriptions.

This section discusses the parameter properties examples in the ‘propertyExamples.tbx’ toolbox.

#### Type

The ‘Type’ property designates a parameter as ‘Required’, ‘Optional’, or ‘Derived’.

By default, parameters are ‘Required’.

A dot appears next to required parameters on the GUI until they are filled in.

The tool will not run unless required parameters are specified.

Optional parameters can be left blank.

Script Tool ‘01\_optionalParam’ takes one required argument (an areal unit) and four optional arguments (‘Cell Size’, ‘Compression’, ‘Double’, ‘Feature Class’ types).

It points to ‘reportSTargs.py’.

When the tool is run with the areal unit set to ‘8 SquareKilometers’ without specifying any of the optional arguments, it reports the following:

When a tool is run with an optional parameter left blank, the default value for that data type is used.

Some data types have a special default value (e.g., MAXOF and LZ77 for cell size and compression data types).

Other data types, such as Double, Feature class, and String, have a generic default value, a hash sign (# ).

The user doesn’t see the hash sign, but this is what the underlying script receives.

The Python script needs to check for the hash sign when handling these types of optional parameters.

Script Tool ‘02\_optionalParam’ takes one required argument (a base) and one optional argument (a power).

This tool points to ‘exponentiator.py’ which calculates the base number raised to a power (e.g., if the input is 5 and 2, the tool prints ‘5.0 raised to the 2.0 is 25.0’).

If the optional argument is omitted, the script uses a power of 1.

To handle the optional argument, the underlying script checks for a hash sign:

With this approach, the user must provide hash signs for the ‘optional arguments’ when running the script outside of the Script Tool, else the script will raise an IndexError exception.

Required arguments should be placed at the beginning of the parameter list (with optional parameters at the end).

An example of the third parameter type, ‘Derived’, will be discussed in the next section, as this is closely related to the ‘Direction’ property.

#### Direction

The ‘Direction’ property designates a parameter as either ‘Input’ or ‘Output’.

By default, parameters are ‘Input’.

Input values are information the script needs to perform its tasks, such as a dataset or workspace to use.

The Script Tool examples so far in this chapter have only used input parameters.

Input parameters can be required or optional, but not derived.

The output direction is used for Script Tool parameters that represent output generated by the tool.

This may be one or more datasets created by the tool.

It may be a modified preexisting dataset (e.g., a dataset with a new fi eld added, as shown in an upcoming example, ‘14\_derivedObtainedFrom’).

It may be a Boolean.

It may be numerical values resulting from Script Tool calculations.

Any types of results generated by standard ArcToolbox tools could be output from a custom Script Tool.

Like standard tools, custom Script Tools can be used as tools in ModelBuilder models.

The output (Boolean, numerical results, dataset, and so forth) would then be passed along to the output ovals in the models.

Output parameters can be required, optional, or derived.

Output parameters with a ‘Required’ or ‘Optional’ type allow the user to set the name of new output datasets that will be created by the Script Tool.

The ‘03\_requiredOutput’ Script Tool has two parameters, one required input feature class and one required output feature class.

It points to a script named ‘copier.py’ that makes a copy of the first argument and names it as specified by the second argument, as in the following code:

arcpy.Copy\_management(sys.argv[1], sys.argv[2])

When a tool with output data parameters is run in ArcMap, the geoprocessing output can be automatically added to the table of contents.

Go to Geoprocessing menu > Geoprocessing options and check ‘Add results of geoprocessing output to the display’.

Run the ‘03\_requiredOutput’ on any input feature class.

When the tool run is completed, the output copy of the input feature class should be automatically displayed on the map.

The combination of ‘Required’ type with ‘output’ direction can only be used for output that will be created by the script, not for modifications to existing data (if you try to select an existing dataset, the GUI will raise an error or warning).

Also, it should only be used when you want to allow the user to select the location and name of the output.

If the output is a modification of existing data or if the script itself determines the output name and location, output should be a ‘Derived’ type parameter.

Input can’t be derived, when you select ‘Derived’ in the ‘Type’ property, the ‘Direction’ property is automatically set to ‘Output’.

To display derived output results, you need to use an arcpy method named SetParameterAsText.

Script Tool ‘04\_derivedOutput1’ has only one parameter, a derived output.

This tool points to script ‘buffer1.py’ (Example23.4 ) which buffers a hard-coded shapefile and passes the output buffer fi le name back to the Script Tool using the following statement:

arcpy.SetParameterAsText(0, outputFile)

TheSetParameterAsText method takes two arguments, a number and a string.

The number specifies the Script Tool parameter index.

The second argument is a string representing the name of the output.

In this case, it’s the output fi le name.

TheSetParameterAsText method does not count the script path name, rather it only counts the parameters in the Script Tool list (using zero-based indexing).

In this example, the Script Tool has only one parameter, so the index for this derived output parameter is zero.

#### Multivalve

#### Default or Schema

#### Environment

This property can be used to set the default value of a parameter to an environment setting.

For example, you can use the current workspace, the scratch workspace, the output coordinate system, and so forth, of the map document to set the default value for a parameter.

To use this, leave the default value blank and instead select an environment setting from the drop-down list in the ‘environment’ property.

The ‘10\_environ’ Script Tool sets the environment property of its only parameter to the scratch workspace.

If this tool is run from ArcCatalog inside of ArcMap within the ‘featureSetExample’ map document, the default value for the ‘Workspace’ parameter is set to ‘C:\gispy\data\ch23’, because this is the value of the scratch workspace environment setting for this map.

#### Filter

The filter property restricts the values that can be entered.

There are six types of filters and the type of filter that can be used depends on the parameter’s data type as listed in Table23.3.

Other data types only give ‘None’ as a choice for the ‘Filter’ property.

To use this property, click on the parameter name, then click in the filter box in the parameter property list.

When you select the fi lter type, a GUI is launched to guide you in specifying the constraints.

Script Tools ‘11\_filterValueList’ and ‘12\_filterRangeList’ use value list and range filters.

Script Tool ‘11\_fi lterValueList’ takes a string argument, a United States region.

To help the user select a valid table, the regions are given in a value list.

Double click on the parameter and then on its filter property to see the ‘List of Values’ (Figure23.10).

These values are simply typed into the list;

Add another item to the list and you’ll see it when you run the tool.

The value list creates a combo box (Figure23.11).

This tool points to ‘regional.py’ (Example23.9) which prints the states in the input region.

If ‘New England’ is selected, the output is:

#### Obtained from

The essential idea of the ‘Obtained from’ property is to use information from one parameter to generate or constrain the value of another parameter.

It can be used for input or derived type parameters.

In Script Tool ‘13\_inputObtainedFrom’, the ‘Field’ values in the second parameter are obtained from the input feature class (Figure23.14 ).

When you run the tool the ‘Field name’ combo box choices change based on the ‘Input feature class’ selection (Figure23.15 ).

This property can only be set for certain parameter data types, ‘Field’, ‘SQL expression’, ‘Linear Units’, ‘Coordinate System’, and a few others.

You can also use this ‘Obtained from’ property to handle output data that is derived from modifying an input file.

For example, input data might be modified by an update cursor or a field may be added, as mentioned in the discussion on the parameter ‘Type’ property.

Script Tool ‘14\_derivedObtainedFrom’ points to ‘com bineFields.py’ (Example23.10 ) which adds a new field to the input dataset by combining two fields.

The tool lists five parameters: an input file, two fields to combine, and a new field name (Figure23.16 ).

The fifth parameter is a derived output parameter obtained from the input fi le;

It corresponds to the updated version of the input dataset.

If this tool is run in ArcMap, the updated dataset is added to the map (or if that dataset is already a map layer, the layer is updated).

#### Symbology

The symbology property can be used to set the visual representation of output parameters.

To do so, you need to use ArcMap to predefine the desired symbology and export the layer as a layer fi le (with an ‘.lyr’ extension).

Then you can set the symbology property of the output parameter to the location of the layer fi le.

When the output is created, it will be added to the map and displayed with the same symbology as the layer fi le.

Script Tool ‘15\_symbology’ points to a script called ‘feature 2point.py’ (Example23.11) which takes an input polygon layer and calls the ‘FeatureToPoint’ tool to find the centroid of each layer.

The derived output feature class parameter is set to the resulting output.

Its ‘Symbology’ property is set to a layer file with star point symbols (‘C:\gispy\data\ch23\training\starPoints.lyr’).

Figure23.17 shows the star symbology output when the tool is run on the United States layer in ‘symbologyPropExample.mxd’.

This is one of several ways in which symbology can be applied via Python.

The ‘ApplySymbologyFromLayer’ (Data Management) tool can also be called from Python.

There are also methods within thearcpy mapping module, discussed in the upcoming chapter.

All of these methods require a training dataset that has the desired symbology pre-configured.

## Showing Progress

While a Script Tool performs lengthy processes, the user may be uncertain about the progress.

To avoid this, the tool can communicate updates as progress occurs.

There are several arcpy progress bar methods designed to provide custom feed back as the tool works.

When a Script Tool is run, the geoprocessing window shows an oscillating progress bar (a trail of boxes moving to and fro) to let the user know that some progress is being made (Figure23.18 ).

The geoprocessing dialog title shows the name of the Script Tool (e.g., ‘deleteFiles’) and below that a label tells the user what the tool is doing.

The default label states that the Script Tool is being executed (e.g., ‘Executing deleteFiles’).

With the arcpy SetProgressor method, you can modify the behavior of this progress indicator.

There are two progress modes:

Figure23.20 shows the geoprocessing window after these three lines of code are run.

The third and fourth argument in theSetProgressor method specify a minimum and maximum value for the progressor position.

An optional fi fth argu ment can be used to specify an interval other than one for each step.

The label is updated when ‘SetProgressorLabel’ is called.

The percentage shown on the bar is updated whenSetProgressorPosition is called.

The percentage is calcu lated as (s \* interval) \* 100%)/maximum (or 100%, whichever is smaller) wheres starts at the minimum value and gets incremented by the interval each time SetProgressorPosition is called.

Usually whenSetProgressor is called, the minimum is set to zero and the maximum is set to the total number of steps being tracked.

In this example, there are four fi les being deleted, so maximum is set to four.

In Figure23.20 , the bar shows 25% progress because the SetProgressorPosition has been called once and the maximum is four (1/4 = 25%).

The ‘defaultProgressor’ Script Tool in the ‘progressorExamples’ toolbox points to ‘defaultProgressor.py’ (Example23.12 ), which uses the default progress bar and updates the labels.

This tool is based on the ‘deleteFiles’ tool (discussed earlier in the chapter) that deletes fi les as specifi ed by the user.

A few lines of code have been added for the progress commands.

The script initializes the default progressor label (arcpy.SetProgressor('default', message) ) and then updates the label inside the deletion loop (arcpy.SetProgressorLabel('Deleting {0}'.format(d)) ).

For demonstration purposes, the built-intime module is used to stall the script progress so that the labels persist long enough to be read.

The sleep method suspends code execution for the given number of seconds.

If the number of steps are known at the outset, as in our ‘deleteFiles’ example, which gathers a list of fi les to delete, each accomplishment can be reported as a percentage of the overall progress.

The number of fi les to be processed (e.g., deleted, buffered, copied, etc.) can be used as the total number of steps.

The ‘stepProgressor’ Script Tool points to ‘stepProgressor.py’ (Example23.13 ), which is identical to default Progressor.py’, except it demonstrates using the ‘step’ progress mode.

The script initializes the step progressor using four arguments, the mode (‘step’), the label message, the minimum step value (0), and the maximum step value (len(data) ).

The script deletes each fi le in a list, so the total number of fi les, the length of this list, is used as the maximum step value.

Inside the deletion loop, the progress label is updated (arcpy.SetProgressorLabel('Deleting {0}'.format(d)) ) and the bar position is updated (arcpy.SetProgressorPosition() ).

Again, calls to the sleep method are only for demonstration purposes and can be removed for practical applications.

## Validating

### The ToolValidator Class

## Python Toolboxes

### Setting Up Parameters (getParameterInfo)

### Checking for Licenses (isLicensed)

### Validation (updateParameters and updateMessages)

### Running the Code (execute)

### Comparing Tools

## Discussion

Script Tools and Python toolboxes provide a convenient means for creating graphical user interfaces for ArcGIS scripts.

Though they have some limitations, they are well-suited for making GUIs to collect ArcGIS data types.

Script Tools are a good starting point for learning to use parameter data types and their properties.

Python toolboxes facilitate a more streamlined workflow for creating tools by using Python code, instead of a tool wizard, to specify parameter properties.

Setting up validation is also more efficient in a Python toolbox tool.

When you modify the code, you can save it, refresh the toolbox, and see the results.

You can leave the saved ‘.pyt’ file open while testing the tool and you can simply edit it again as needed.

Each time you edit Script Tool validation code, you need to open the tool properties, select the ‘Edit’ button on the validation tab, edit and save the code, apply the changes, close the tool properties, and then run the tool.

When you make changes to the code in Python toolboxes, you do need to refresh the file in ArcCatalog to see the changed behavior.

A red X on a tool icon in ArcCatalog means the tool won't run.

When you see an X on a Script Tool, it often means that the path to a script was not set as relative or the script has been moved to a different relative position.

If this is the case, you can update the ‘Script’ path to fix the tool (and check ‘Use relative pathnames, if desired).

Python toolboxes help to avoid this common mistake since they define the toolbox and tools in one file.

If the code in a Python toolbox contains an error, a red X will appear as well.

For certain kinds of errors, a ‘Why’ tip will appear on the tool.

You can select this tip to learn more about the error (see ‘zzz.pyt’ for an example).

ArcGIS hosts a downloadable tool for exporting Script Tools to Python tool boxes, but similar to exporting ModelBuilder models to Python scripts, the user needs to have a good understanding of Python to refine the resulting ‘.pyt’ file for full functionality.

The template along with sample Python toolboxes provide a good starting point for creating new Python toolboxes.

Graphical user interfaces should usually be added near the end of a GIS project, once the programming functionality has been developed and tested.

GUIs are convenient for users, not for developers.

## Key Terms

Script Tool

Geoprocessing Window

Widget

Combo box

ToolValidator class

Validation

Internal validation

Python toolbox

Parameter object

Value object

## Exercises

# Mapping Module

Abstract

Suppose you want to take a screen shot of dozens of maps to insert into a report.

Or suppose you’ve been reorganizing your data and you want to ensure that you haven’t broken the data paths on a large collection of maps.

Or suppose you’ve carefully selected a symbology for a layer and you’d like to apply this symbology to other maps that use that data.

With the arcpy package mapping module, you can automate these and other tasks related to map documents and their layers.

The mapping module is a script that resides within the arcpy package installed with ArcGIS.

This chapter discusses how to use the mapping module to work with map documents, data frames, layers, symbology, and map layout elements.

Chapter Objectives

After reading this chapter, you’ll be able to do the following:

* Describe the limits and capabilities of thearcpy mapping module.
* Explain the hierarchy of map documents, data frames, and layers.
* Use MapDocument, DataFrame , and Layer properties and methods.
* Differentiate betweenarcpy mapping code for the ArcMap Python window andarcpy mapping code for stand-alone scripts.
* List the data frames in a map.
* List the layers in a map.
* Save a map document.
* Save a copy of a map document.
* Export a map as an image.
* Move map layers within the table of contents.
* Remove layers from the map.
* Add layers to a map.
* Modify the symbology of a layer.
* Modify map layout elements.

Suppose you want to take a screen shot of dozens of maps to insert into a report.

Or suppose you’ve been reorganizing your data and you want to ensure that you haven’t broken the data paths on a large collection of maps.

Or suppose you’ve carefully selected a symbology for a layer and you’d like to apply this symbology to other maps that use that data.

With the arcpy package mapping module, you can auto mate these and other tasks related to map documents and their layers.

The mapping module is a script that resides within the arcpy package installed with ArcGIS.

This chapter discusses how to use the mapping module to work with map documents, data frames, layers, symbology, and map layout elements.

To understand the mapping module, it’s helpful to keep in mind the hierarchical structure of ArcMap.

ArcMap opens map documents, containing data frames, and the date frames have layers.

The layers themselves point to geographic data stored on a computer disk.

The three central classes in the mapping module are the MapDocument, DataFrame, and Layer classes, which correspond to the structure of ArcMap.

The mapping classes and functions are designed for managing existing map documents (not for creating new ones) and for automating map publication.

The module provides access to the DataFrame and Layer objects with ListDataFrames and ListLayers methods that return lists of objects.

The properties of these objects can then be modified.

There are also a set of methods for exporting maps in image formats or PDF (portable document format).

Some examples in this chapter use the ArcMap Python Window.

This is a departure from other chapters which work strictly with stand-alone IDEs.

The ultimate goal of this chapter still aims for writing stand-alone scripts, ones that you can run outside of ArcMap.

However, because the mapping module manipulates map document properties, it’s useful to test mapping module code statements within an open map document.

We’ll start in the PythonWin Interactive Window and then move to the ArcMap Python Window as we explore the MapDocument, DataFrame, and Layer classes.

## Map Documents

### Map Name or 'CURRENT' Map

### MapDocument Properties

### Saving Map Documents

## Working with Data Frames

## Working with

### Moving, Removing, and Adding Layers

### Working with Symbology

## Managing Layout Elements

## Discussion

In this chapter, we used the arcpy mapping module to modify map, data frame, and layer properties, to add, move, and delete layers, and to modify symbology and layout elements.

These operations involve working with map documents, data frames, layers, symbology, and layout element object properties and methods.

Most mapping scripts begin by creating a MapDocument object, listing the DataFrame objects, and listing the Layer objects.

To explore these objects further, reference the 'CURRENT' map document in the ArcGIS Python window.

When porting code to stand-alone applications, be sure to reference the full path file name of the map document and be aware of data locking conditions.

This chapter demonstrated many of the mapping module capabilities, but not all.

If you are working with temporal data attributes, you may want to investigate additional mapping module functionality, such as the DataFrameTime and LayerTime classes that enable dynamic map displays.

One final tip, when you're using the mapping module extensively, it may be helpful to print the alphabetical lists of mapping classes, functions, and constants found in the ArcGIS Resources help pages.

## Key Terms

MapDocument objects

DataFrame objects

Layer objects

'CURRENT' vs. map name

RefreshActiveView method

Layer file (.lyr extension)

SymbologyType property

Layer object symbology class

Layout element objects

## Exercises

#

Several online sights have excellent video or interactive tutorials for beginning programmers.

These can be used in tandem with the chapters of this book.

For example, as you read Chapter 2 on Python variables, you can watch the New Boston Python 'Variables' video for reinforcement.

The online resources listed as 'References' are not meant to be read straight through; Instead, these can be searched for key Python or ArcGIS terms.

The 'ArcGIS Resources' site has Python help that is specific to ArcGIS; the others contain general GIS Python help or general Python language help.

Forums are another useful resource for programmers.

If you're having a problem with your script, someone else has probably had that problem.

Search online forums for related posts and post questions yourself.

It is well worth the time it takes to sign up for an account.