Part 2: Writing scripts

Notes

Contents

Chapter 5: Geoprocessing using Python

5.1 Introduction

- This chapter describes the ArcPy site package. This allows for the close integration of ArcGIS and Python
- ArcPy modules, classes, and functions give access to all geoprocessing tools

5.2 Using the ArcPy site package

- Use the ArcPy site package!
- A site package is like a library of functions that add functionality to Python
- Site packages work like a module, but contains multiple modules as well as functions and classes
- ArcPy is organized in modules, functions, tools, and classes

5.3 Importing ArcPy

- You have to first start with importing the site package. Therefore, your geoprocessing code will begin with

```
import arcpy
```

- Once this is done, you can run all of the geoprocessing tools found in the standard toolboxes installed with ArcGIS
- There are numerous pre-installed modules, but some important ones are:
    - `arcpy.mapping` (a map automation module)
    - `arcpy.sa` (a map algebra module)
    - To import these, use the following syntax

```
import arcpy.mapping
```

- If you want to set a workspace, you could do the following, which makes our workspace a string variable:

```
import arcpy
arcpy.env.workspace = "C:/Data"
```

- By doing this, you can store and set or retrieve values of variables
- You usually don't need to use the entire module, so you can use the `from-import` statement to only import a portion:

```
from arcpy import env
env.workspace = "C:/Data"
```

- You can also give a module or part of a module a custom name (if you want)...it can make your code easier to read if you do it correctly
- The `from-import-as-*` the contents of the module are imported directly into the namespace, which means you do not need to add a prefix to the contents

```
from arcpy import env as *
workspace = "C:/Data"
```

5.4 Working with earlier versions of ArcGIS

- The ArcPy package was introduced for ArcGIS 10
- Before this introduction, the geoprocessing tools were accessed by using the `ArcGISscripting` module
- For example, if I was using ArcGIS 9.3, I would have to use:

```
import ArcGISscripting
gp = ArcGISscripting.create(9.3)
```

- Before ArcGIS 9.3, you have to use:

```
import ArcGISscripting
gp = ArcGISscripting.create()
```

- There is more detailed ways to getting at specific modules, but it is likely that I will not have to use them

5.5 Using Tools

- When using geoprocessing tools in Python, they are referred to **by name**, and not by what the "name" displayed in the ArcGIS toolbox
- Generally, the tool name is the same as you would normally see, but with no spaces
- You also need the toolbox alias (multiple different tools can contain the same name)
- Two ways of accessing a tool in a line of Python code:
    - call its corresponding function
    - call module that matches the toolboxes alias name

```
arcpy.<toolname_toolboxalias>(<parameters>)
```

```
arcpy.<toolboxalias>.<toolname>(<parameters>)
```

5.6 Working with toolboxes

- Again, all of the system toolboxes are available, but you would also have to call some of these. Ways to do this are:

```
import arcpy
arcpy.ImportToolbox("C:/Data/sampletools.tbx")
```

- This function references the actual file on the disk, not the name of the toolbox. Now, after importing a toolbox, the syntax for accessing a tool in Python is as follows:

```
arcpy.<toolname>_<toolboxalias>
```

- Once a particular tool is identified, the tool's syntax can be accessed from Python using the `Usage` function. For example,

```
import arcpy
tools = arcpy.ListTools("*_analysis")
for tool in tools:
print arcpy.Usage(tool)
```

- Another way to do the exact same thing is to use the `help` function:

```
print help(arcpy.Clip_analysis)
```

5.7 Using functions

- In ArcPy, all tools are functions, but not all functions are tools
- Some functions include:
  - Cursors
  - Describing data
  - Environment and settings
  - Fields
  - Geodatabase administration
  - General
  - General data functions
  - Getting and setting parameters
  - Licensing and installation
  - Listing data
  - Log history
  - Messaging and error handling
  - Progress dialog boxes
  - Raster
  - Spatial reference and transformations
  - Tools and toolboxes

5.8 Using classes

- The `SpatialReference` class contains coordinate system parameters. To work with these properties, you must instantiate them by initializing a new instance of a class:

```
arcpy.<classname>(parameters)
```

```
import arcpy
prjfile = "C:/Data/myprojection.prj"
spatialref = arcpy.SpatialReference(prjfile)
```

5.9 Using environment settings

- These are essentially hidden parameters that influence how a tool runs
- Each environment has a name and a label

5.10 Working with tool messages

- When running a tool, you get various messages. Some include:
    - Exact time when running the tool started and ended
    - The parameter values used to run the tool
    - Information about the progress in running the tool (information messages)
    - Warnings of potential problems in running the tool (warning messages)
    - Any errors that prevented the tool from running (error messages)
        - Severity = 0 (information message)
        - Severity = 1 (warning message)
        - Severity = 2 (error message)
- Some commands to use (self-explanatory):
    - GetMessage
    - GetMessageCount
    - GetMaxSeverity

5.11 Working with Licenses

- There are certain licenses that allow you to do certain things. If you don't have a particular license, you can't do any license-exclusive scripting!

5.12 Accessing ArcGIS Desktop Help

- This is the same as normal

Chapter 6: Exploring spatial data

6.1 Introduction

- This chapter describes several approaches to exploring spatial data

6.2 Checking for the existence of data

- You typically first have to specify input datasets on the tool dialog box by selecting a list of available layers or by browsing to the correct dataset
- This is done with the `Exists` function, which returns a Boolean value, indicating whether the element exists or not
- For example, this would look like:

```
arcpy.Exists(<dataset>)
```

- Or more specifically, to check if a shapefile exists, you would:

```
import arcpy
print = arcpy.Exists("C:/Data/streams.shp")
```

- Remember that when working in Python, keep two paths in mind:
    - System paths = paths recognized by the Windows OS
    - Catalog paths - paths that only ArcGIS recognizes
        - File geodatabase has a .gdb extension
        - Personal geodatabase has a .mdb extension
        - Enterprise geodatabase has a .sde extension

6.3 Describing data

- Each data type has properties that can be used to control the flow of a script
- The `Describe` function can be used to determine the properties of the input feature class, including the feature type (point, polygon, polyline, etc.)

    - Determine the feature type of a dataset before using it in a tool
- The syntax is:

```
import arcpy
<variable> = arcpy.Describe(<input dataset>)
```

- Running this returns an object that contains the properties of the dataset. These properties can be accessed using the `<object>.<property>` statement
- "Describe" can be used on datasets such as feature classes, shapefiles, rasters, and tables
- These properties that are created are organized into a series of property groups, one being `FeatureClass` which includes properties such as `shapeType`

6.4 Listing data

- I may (at some point) want to make an inventory of the available data so a script can iterate over the data during processing
- There are a lot of built-in functions in ArcPy that do this
- These functions create a Python list, which can contain strings and other data types
- Working with lists in Python usually requires `for` loops
- The ArcPy list functions include:
    - `ListFields`
    - `ListIndexes`
    - `ListDatasets`
    - `ListFeatureClasses`
    - `ListFiles`
    - `ListRasters`
    - `ListTables`
    - `ListWorkspaces`
    - `ListVersions`
- The syntax of these include:

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

- Wild cards (*) can be used to limit list names
- There are, of course, many optional parameters
- To create a list of rasters, use this syntax:

```
ListRasters ({wild_card}, {raster_type})
```

6.5 Using lists in `for` loops

- You can use "for" loops with raster data. For example:

```
import arcpy
from arcpy import env
env.workspace = "C:/Data"
tifflist = arcpy.ListRasters("", "TIF")
for tiff in tifflist:
    arcpy.BuildPyramids_management(tiff)
```

- The "ListRasters" function creates a list of TIFF tiles and the "for" loop iterates over each element in the list and builds pyramids for each
- This automates tedious tasks

6.6 Working with lists

- While Python lists can be made up of a specific data type, but the list can also be manipulated in Python using functions and methods of a Python list
- The number of feature classes in a workspace can be determined using the `len` function
- Lists can be sorted using the `sort` method
- Lists can be reversed using the `reverse` argument of the `sort` method

6.7 Working with tuples

- If you want to use a list without modifying the elements within it, you want to use a tuple
- Tuples are sequences of elements but are immutable
- These are just elements separated by commas
- Very similar to working with lists

6.8 Working with Dictionaries

- Dictionaries are a set of keys and their corresponding values
- Pairs are also called items in the dictionary
- A dictionary item consists of a key, followed by a colon, and then its corresponding value

- The entire dictionary is surrounded by curly brackets
- You can start off with an empty dictionary ({ }) and then add to it

```
>>> zoning = {}
```

- Items can then be added such as:

```
>>> zoning["RES"] = "Residential"
>>> zoning["IND"] = "Industry"
>>> zoning["WAT"] = "Water"
>>> print zoning
{"IND': "Industry", "RES": "Residential", "WAT": "Water"}
```

- Then, items can be modified using the same syntax. For example:

```
>>> zoning["IND"] = "Industrial"
>>> print zoning
{"IND': "Industrial", "RES": "Residential", "WAT": "Water"}
```

- Items can be deleted using square brackets and using the following keyword:

```
>>> del zoning["WAT"]
>>> print zoning
```

- The `keys` method returns a list of all the keys in the dictionary:

```
>>> zoning.keys()
```

- The `values` method returns a list of all the values in the dictionary:

```
>>> zoning.items()
```

- Generally, dictionaries are not common within ArcPy, but are used within `GetInstallInfo`

Chapter 7: Manipulating spatial data

7.1 Introduction

- This chapter introduces the ArcPy data access module `arcpy.da` which is used for working with data

7.2 Using cursors to access data

- Cursors are used to work with the rows in a table. A cursor is a database technology term for accessing a set of records in a table
- A cursor can be used to iterate over the set of records in a table or to insert new records into a table
- Records in a table are also referred to as rows
- There are three types of cursors
    - A search cursor is used to retrieve rows (SearchCursor)
    - An insert cursor is used to insert rows (InsertCursor)
    - An update cursor is used to update and delete rows (UpdateCursor)

Cursor Type	Method	Description
Search	next	Retrieves the next row
	reset	Resets the cursor to its starting position
Insert	insertRow	Inserts a row into the table
	next	Retrieves the next row object
Update	deleteRow	Removes the row from the table
	next	Retrieves the next row object
	reset	Reets the cursor to its starting position
	updateRow	Updates the current row

- Search and update cursors also support `with` statements
- You can also delete rows and cursors

7.3 Using SQL in Python

- Structured Query Language (SQL) is used to define one or more criteria based on attributes, operators, and calculations
    - Used in the Select by Attributes dialog box in ArcMap
- SQL queries can be carried out in Python using the `SearchCursor` function. The syntax is:

```
SearchCursor (in_table, field_names {where_clause}, {spatial_reference},
{fields}, {explode_to_points})
```

- There are other options such as the

```
AddFieldDelimiters(datasource, field)
```

- Another place where SQL is used is in the Select Tool

7.4 Working with table and field names

- It is important to have tables and fields unique names
- The `ValidateTableName` is a function which helps determine if the name for that workspace is available

```
ValidateTableName(name, {workspace})
```

- Fields cannot be added unless the name is valid, otherwise the script may fail during execution
- Similarly, we have
    - `ValidateFieldName`

7.5 Parsing table and field names

- The following function can be used to split the fully qualified name for a dataset into its components. The syntax is:

```
ParseTableName(name, {workspace})
```

- This returns a single string with the database name, the owner name, and table name, each seperated by a comma

7.6 Working with text files

- You can open files using:

```
open(name, {mode}, {buffering})
```

- If you want to do something other than just open the file, you must specify it:
  - r: read mode
  - w: write mode
  - +: read/write mode (added to another mode)
  - b: binary mode (added to another mode - for example, rb, wb, and others)
  - a: append mode
- If nothing is specified, then the read mode, r, is used by default
- To create a new file object, you can do something like:

```
>>> f = open("C:/Data/mytext.txt", "w")
```

- You can use numerous methods in order to manipulate a file. For example,

```
>>> f = open("C:/Data/mytext.txt.", "w")
>>> f.write("Geographic Information Systems")
>>> f.close()
```

- The `readline` method reads the next line from the text file and returns it as a string. If you continue with this, it successively reads the next line until the end
- It separates each line on a new "line" of whitespace, but includes a list with each line as an element, including a line break
- This can be useful if you only want part of a data table (i.e. only some of the rows and/or columns
- Files in Python are iterable

Chapter 8: Working with geometries

8.1 Introduction

- This chapter describes how to work with geometries

8.2 Working with geometry objects

- Each feature in a feature class contains a set of points that define vertices of the feature
- These points are accessed using geometry objects:
    - Point
    - Polyline
    - PointGeometry
    - MultiPoint
- These all return an array of point objects
- This is very time consuming, but there are shortcut commands if you wish to use them

8.3 Reading geometries

- For a polyline or polygon feature class, each point is represented at a vertex
- There can be numerous points for these feature classes as well, and there is also a shortcut to get particular points
- For these particular shortcuts, refer to pp. 161 - 163

8.4 Working with multipart features

- Feature class can have multiple parts, thus making them multipart features (for example, each island of the Hawaiian Islands)
- You can have several different point return types, such as:
    - Point, Polyline, Polygon, MultiPoint, and MultiPatch
- For this syntax, refer to pp. 165 - 166

8.5 Working with polygons with holes

- If a polygon has holes, it essentially has two rings - an outer ring and an inner ring (aka exterior ring and interior ring)
- In this case, both the interior and exterior ring points are returned

8.6 Writing geometries

- If you use the **insert and update cursor** option, you can create or update new features
- This is done using the `CreateFeatureClass` function
- For more on this, refer to pp. 169 - 171

8.7 Using cursors to set the spatial reference

- The spatial reference for a feature class describes the coordinate system, the spatial domain, and the precision
- It is typically set when the feature class is created
- Setting the spatial reference is not required, so sometimes you will find that it is not specified
- Spatial reference is applied to all features of the feature class
- You can set the spatial reference of a search cursor
- For the actual syntax on how to set the spatial reference, refer to pp. 172 - 173

8.8 Using geometry objects to work with geoprocessing tools

- Inputs for geoprocessing tools often consist of feature classes
- Sometimes, these feature classes do not exist, and need to be created
- Geometry objects can also be created directly as the output of geoprocessing tools

Chapter 9: Working with rasters

9.1 Introduction

- Rasters present a unique type of data
- This chapter is all about how to use the ArcPy functions to list and describe rasters
- ArcPy also includes the Spatial Analyst module (arcpy.sa), which fully integrates map algebra into the Python environment (makes scripting much more efficient)

9.2 Listing rasters

- The `ListRasters` function returns a Python list of rasters in a workspace. The syntax is

```
ListRasters({wild_card}, {raster_type})
```

- Wild_card is an optional parameter and can be used to limit the list based on the name of the rasters
- The optional raster_type parameter can be used to limit the list based on the type of raster (ex. JPEG or TIFF)
- Below is an example of ListRasters

```
import arcpy
from arcpy import env
Env.workspace = "C:/raster"
rasterlist = arcpy.ListRasters()
for raster in rasterlist:
    print raster
```

- The output would look like the following:

```
elevation
landcover.tif
tm.img
```

- The name of each raster is printed, potentially with a file extension
- The parameters of the ListRasters function can be used to filter the results, for example:

```
import arcpy
from arcpy import env
env.workspace = "C:/raster"
rasterlist  = arcpy.ListRasters("*", "IMG")
for raster in rasterlist:
    print raster
```

9.3 Describing raster properties

- Rasters can be described using the generic `Describe` function as already discussed (in Chapter 6)
- The Describe function returns the properties for a specified data element
- Properties are dynamic, meaning, the properties that are present depend on the data type being described
    - For example, when the Describe function is used on rasters, a generic set of properties is present in addition to specific properties that are unique to the specific raster element
- Three different raster elements can be distinguished:
    1. Raster dataset - a raster data model that is stored on disk or in a geodatabase. Raster datasets can be stored in many formats (TIFF, JPEG, IMAGINE, Esri GRID, and MrSID). Raster datasets can be single band or multiband
    2. Raster band - one layer in a raster dataset that represents data values for a specific range in the electromagnetic spectrum or other values derived by manipulating the original image bands. Many types of satellite images, for example, contain multiple bands
    3. Raster catalog - a collection of raster datasets defined in a table of any format, in which the records define the individual raster datasets that are included in the catalog. Raster catalogs can be used to display adjacent or overlapping raster datasets without having to combine them into a mosaic in one large file
- The following is an example syntax:

```
import arcpy
from arcpy import env
env.workspace = "C:/raster"
raster = "landcover.tif"
desc = arcpy.Describe(raster)
print desc.dataType
```

- The following are properties specific to raster datasets:
    - `bandCount` - the number of bands in the raster dataset
    - `compressionType` - the compression type (LZ77, JPEG, JPEG2000, or None)
    - `format` - the raster format (GRID, IMAGINE, TIFF, and more)

- `permanent` - indicates the permanent state of the raster: False if the raster is temporary, True if the raster is permanent
- `sensorType` - the sensor type used to capture the image

- An example of these are:

```
import arcpy
from arcpy import env
env.workspace = "C:/raster"
raster = "landcover.tif"
desc = arcpy.Describe(raster)
print desc.dataType
print desc.bandCount
print desc.compressionType
```

- The following are specific to raster bands:
  - `height` - the number of rows
  - `isInteger` - indicates whether the raster band is an integer type
  - `meanCellHeight` - the cell size in y direction
  - `meanCellWidth` - the cell size in the x direction
  - `noDataValue` - the NoData value of the raster band
  - `pixelType` - the pixel type, such as 8-bit integer, 16-bit integer, single precision floating point, and others
  - `primaryField` - the index of the field
  - `tableType` - the class name of the table
  - `width` - the number of columns
- For examples, see p. 181

9.4 Working with raster objects

- ArcPy also contains a Raster class that is used to reference a raster dataset
- A raster object can be created in two ways: (1) by referencing an existing raster on disk and (2) by using a map algebra statement. The syntax of the Raster class is:

```
Raster(inRaster)
```

- Below is how to create a raster object by referencing a raster on disk:

```
import arcpy
myraster = arcpy.Raster("C:/raster/elevation")
```

- Raster methods have only one method - `save`
- The save method makes the raster permanent (i.e., not temporary, by default)
- Syntax:

```
save({name})
```

9.5 Working with the ArcPy Spatial Analyst module

- The Spatial Analyst module carries out map algebra and other operations
- Very similar to the Spatial Analyst toolbox
- Can be more efficient than using the Spatial Analyst toolbox, however

9.6 Using map algebra operators

- Most of the map algebra operators are also available as geoprocessing tools under the Math toolset in the Spatial Analyst toolbox
- Using these operators in scripting can be more efficient
- This typically saves a large amount of code and is easier than using the typical Raster Calculator
- You can also use the raster calculator in Python:

```
RasterCalculator(expression, output_raster)
```

- Using this is only more efficient if you are using operators which are not already available in Python

9.7 Using the `ApplyEnvironment` function

- This function copies an existing raster and applies the current environment settings
- Syntax is as follows:

```
ApplyEnvironment(in_raster)
```

- For example, this allows you to change things like the extent or the cell size or to apply an analysis mask

9.8 Using classes of the `arcpy.sa` module

- This module contains a number of classes that are used for defining parameters of raster tools
- For example, there is the `Reclassify` tool, in which raster cells are given a new value (based on the reclassification table)
- For specifics of this tool, refer to pp. 188 - 190
- There are a variety of neighborhood objects. See pp. 190 - 192 for specifics

9.9 Using raster functions to work with NumPy arrays

- NumPyArrayToRaster - designed to work with very large arrays (scientific computing in Python)
- RasterToNumPyArray
- The two functions above are normal ArcPy functions
- May need to use the SciPy package