**Python Fundamentals**

Created by: Nathan Jennings

Created on: 09.15.2015

Updated on: 01.18.2016

**Overview**

This material and scripts can be used to help new Python programmers view and learn some of the fundamental concepts of Python. This material also complements the information in **Chapter 3** in *A Python Primer for ArcGIS, Workbook I*. In addition, the **Python Tutorial** on the **python.org** <https://docs.python.org/v.x/> (replace **v.x** with the appropriate Python version, e.g. 2.7) site shows some brief examples of most of these concepts among many others that one may find helpful during code development.

Python is a relatively easy programming language to learn and understand. A new programmer can start writing and implementing applications by knowing how to write a few basic programming constructions. The neophyte programmer can work through these examples to obtain grounding in Python fundamentals which will be at the core of almost all other code. Some ArcGIS (*arcpy*) fundamentals are also illustrated.

**Using and Modifying the Scripts**

The reader should open a script and read through the syntax to get a feeling for the code that is written. In some cases, the script will run “as is” since no GIS data or information is being processed. In other cases, where data is being processed, the reader will need to modify the appropriate workspaces and/or data paths as required to reference the **ArcPy\_Bytes/Data** folder. Currently, some of the scripts have data paths set to **“C:\PythonPrimer\ArcPy\_Bytes\Data”**. If the reader copies and unzips the **ArcPy\_Bytes.zip** file to **C:\PythonPrimer**, the scripts should work without modification. The sample scripts are found in the **ArcPy\_Bytes** folder; the data for some of the examples is found in the **ArcPy\_Bytes/Data** subfolder. NOTE: Make sure to modify the workspaces and data paths for the different scripts as required.

The Python IDLE environment is recommended to write and develop code. For those using ArcGIS, this can be found under **Programs—ArcGIS—Python v.x – IDLE**.

**Python Code Structure**

Program: **general\_python\_structure.py**

This program illustrates the general layout of a typical Python program. Almost all cases for the exercises and programs developed as part of *A Python Primer for ArcGIS*, only one *try* and one *except* block will be needed.

The program typically has a title and a space for some overall commentary about the program. Next, is a space for common variables that will be used throughout the program. Once these variables are defined, the bulk of the program will be written within the *try* block. Exception code (aka error handling code) will be written in the *except* block. See the **exception.py** file that contains the typical error handling code that is used in all of the programs and exercises within *A Python Primer for ArcGIS*. A *try* block must have an accompanying *except* block.

**Defining Variables**

Program: **variables.py**

In most cases variables will be defined as strings, numbers, lists, data paths, or workspaces. Data paths are just strings, but the string represents a folder path or a path to a geodatabase (e.g. geodatabase.gdb). Strings can define names of shapefile feature classes, geodatabase feature classes, images (rasters), names of attribute fields, specific attribute values, map document names, data frames, among many others.

Remember that “case” matters in variable names, so *aname* and *aName* represent different variables. Also, using a standard method of defining variables is often best coding practice. See Chapter 3 for a thorough discussion.

The **variables.py** file shows examples of common variable definitions typically found in many Python scripts (and especially in ArcGIS related scripts). The reader will note the use of *print* statements, a typical method used to help troubleshoot code. With simple *print* statements a code developer can easily review variable values used within the code. If the unexpected value appears, then this is an indication to the programmer that the code may need to be changed.

Review this script and then try writing some of your own variables. Make changes to variable definitions and re-run the code to see the new changes.

A simple *for* loop construction is provided to illustrate the use of variables and how the values can change (in this case, values within a Python list).

**Strings**

Program: **strings.py**

As mentioned above, strings are common structures in Python and many kinds of variables are often defined as strings. The **variable.py** script shows a few string variables. **Strings.py** shows a few more examples.

This program shows a few simple examples of defining and using strings.

The first section defines two separate variables and then concatenates (combines) them together

The next section shows a few examples of string variables being defined to a folder or a file geodatabase. Keep in mind that the variables are just names and even though they are defined as either a folder or a geodatabase, the “string” of the folder or geodatabase names is assigned to the variable. At this point, the programmer does not know if the folder or geodatabase exists.

The program continues by defining a variable to the name of a shapefile. The shapefile name and the datapath variable assigned to the folder are concatenated together in two different ways.

1. Using *os.path.join()* – this routine requires that the *os* module be imported (see the *import* line). The *os.path.join* routine is a simple way to combine multiple parts of a computer path together (i.e. concatenating different parts of a path such as a folder and a file name).
2. Simple string concatenation – this option has the same result as *os.path.join*.

A couple of *print* statements are written to simply write out the strings to the Python Shell.

The last section of the code uses strings to create a prospective query statement. Query statements are typically used in the feature selection (e.g. *SelectLayerByAttribute*) or cursor (*Search, Update*) routines and are often sources of coding problems to obtain the appropriate records from a database. The code shown in this example illustrates a typical query statement structure. Chapter 5 in *A Python Primer for ArcGIS, Workbook II* focuses on querying and selecting data. Chapter 6 focuses on cursor routines.

Note the syntax of the right side of the equal sign where “triple double” quotes are used. The entire right side of the query line is composed of strings concatenated together.

A number of “string functions” exist in Python. Examples include determining the length of a string, string parsing, string formatting (such as for use in formatting output text files), among others. A couple of these are illustrated at the end of the **stings.py** script. An example of formatting numbers as string is shown in the **numbers.py** script. See additional information for strings at the **python.org** site under “String Methods.”

The last part of the script shows some examples to “extract” parts of a string. Some of these might be useful when parsing path or file, feature class, table, or image names. There are also other Python methods to extract file root names from extensions and parse folders or directory structures. See the *os* module at the **python.org** site for more details.

**Numbers**

Program: **numbers.py**

Numbers are often used for math routines, counters, index values, lengths of strings, etc in Python. The **numbers.py** script shows a few basic examples of using numbers in Python. See additional information for numbers at the **python.org** site under “Numbers.”

The script begins with some simple variable definitions of numbers. Note that decimal places are used and will help determine the precision of the number and to be able to compute the correct values such as for division.

A list of numbers is provided with different levels of precision and then a loop is created to iterate over the list of numbers, computes a sum, and then reports an average. The loop also prints the number using a special string format routine that can be applied to numbers and can provide some standardization in cases where this is often required (such as with text file output that required standardized text formatting).

The program continues with an example of integers and shows a division result if both numbers are integers versus if one of the numbers is a floating point number (i.e. a decimal number).

The Python *range* routine is to generate a list of numbers and then also computes the sum and average.

Note the use of string concatenation and the *len()* routine so that the proper math can be performed. The *len()* provides the number of objects in a list (in this case, the count of numbers in the list so the average can be computed properly).

Complex numerical processing is beyond the scope of the Python Basics. The reader is encouraged to review *numpy* (**numpy.org**) and *scipy* (**scipy.org**) that contains higher level numerical and statistical computations. Components are these modules are embedding in some ArcGIS processing.

**Workspaces and Data Paths**

Program: **workspaces\_and\_datapaths.py**

ArcGIS workspaces and data paths (not an ArcGIS construct) are different and can impact how subsequent coding is created and processed.

Workspaces are ArcGIS constructs that are part of the *arcpy.env* Python module that act similar to a data path (such a *c:\folder\subfolder*….) but provide some additional options for geoprocessing, such as ArcGIS (*arcpy*) List routines. Any of the listing routines requires a “current workspace” to be set. See the ArcGIS Help on any of the list routines (such as feature classes, files, rasters, etc). If only a data path is referenced (i.e. without using *arcpy.env.workspace*), then ArcGIS list routines cannot be implemented (among other routines). Many ArcGIS routines do not require a workspace and a simple data path variable definition (to a folder or geodatabase) may suffice). Data paths can typically be defined for a variety of output folders or geodatabases if data needs to be copied or moved from one location to one or more different locations).

As mentioned above, a workspace (or “the current workspace”) will be defined as follows:

**arcpy.env.workspace** = *“<folder location to a subfolder or geodatabase>”*

In addition to setting a workspace, programmers often find the need to “concatenate” (combine) different parts of folders, subfolders, files, feature classes, etc. The Python *os.path.join* routine can be used. This construct is used throughout *A Python Primer for ArcGIS* and is also found in many of the ArcGIS and Python documentation sources. In general, this construct can be defined as follows:

datapath = *“<a string set to a folder>”* # for example

aVariable = **os.path.join**(*datapath, “<shapefilename.shp>”*) # for example

**Lists and Looping**

Program: **lists\_and\_looping.py**

This script reviews the basic concepts of Python and ArcGIS lists and looping structures such as the *for* and *while* loops. Looping structures are designed to “iterate” over numerous items in lists. Common examples in GIS are sets of records in a feature class or table, lists of feature classes, images, data sets, and fields. The reader has already seen an example of the *for* loop to iterate over lists of strings or numbers.

The script begins with a simple variable definition of a Python list of string values. A *for* loop structure then iterates over each value in the list and prints out the value to the Python Shell.

Each element in a Python list is accessed by using an index value which always begins with index value “0”.

In each looping routine, the loop structure condition must evaluate to “True” before the loop can be processed.

The general looping structures are shown below.

**for** *<a value>* **in** *<sequence>*: # the value must exist in the sequence to be true

# process the loop if the loop structure is true

**while** *<condition is true>*: # the condition must be true to process the loop

# process the loop if the loop structure is true

A couple of ArcGIS list routines are created that contain a list of feature classes or files. A *for* or *while* loop is then used to iterate of the list of values and prints the name of the feature class or file to the Python Shell.

**Conditional Statements**

Program: **if\_condition.py**

Conditional statements can be used to test different conditions. One can think of this as “making decisions” in the code.

if *<condition is true>*:

# process this block of code

elif *<a different condition is true>*:

# process this block of code

*<other elif statements can be added here>*

else:

# process this block of code

The Python script shows a couple of examples of using the *if..elif…else* construct. The user can change some of the variables and re-run the script to see the different changes.