Automating tasks in ArcGIS - Python

1. Introduction

Geoprocessing tasks can be time intensive since they are often performed on a number of different datasets, on large datasets with numerous records, on a number of operations on same, different datasets or on the kind of repetitive tasks. Scripting is an efficient way for automating geoprocessing tasks with more flexibility. It allows the execution of simple processes (a single tool) or complex processes (multitool tasks with validation). In addition, scripts are recyclable, meaning they can be data nonspecific and used again.

ArcGIS provides development environments such as Visual Basic, Python, and high level programming language which allow you to easily automatic your task and customize your application.

In this lab, we are going to discuss the use of Python script in ArcGIS environment

Some reasons for programming in a GIS are:

- To calculate values needed for analyses
- To perform repetitive tasks
- To create customized GIS interfaces for special purposes
- To automate complex GIS modeling
- Spatial data manipulation etc.

---

2. Python Basics

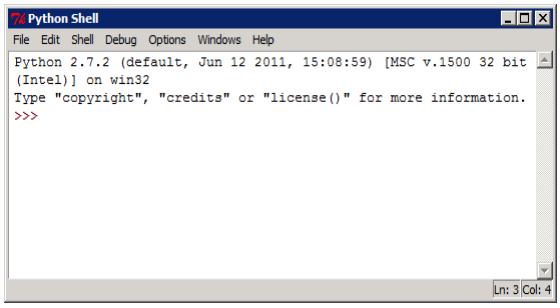
Before working on any ArcGIS script, first you need to take a closer look the basics of Python language.

"Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales"

You can run python in either interactive mode by using Python Interpreter in PythonShell, an Integrated IDE environment, which allows to program, test, debug, and execute of Python code, with the resulting messages printed to the window or run a python script file that contains many python statements.

- Start Python by clicking Start->All Programs->ArcGIS->Python2.7->IDLE(PytonGUI). The following window is popped up and ready for input commands.

">>>" is the command prompt in Python



- Type in the following code.

```
>>> a = 10
>>> b = 20
>>> print a + b
```

The first line means the value of 10 is assigned to variable **a** and 20 is assignment to **b**. The last line print out the result **a+b**

*Q. What is the result of **a+b** ?*

- Type in the following code.

```
>>> a = "This is a test!"
>>> a
```

*Q. What is the value of variable **a** ?*

Here **a**, **b** work as variables that reference to values. Variable name must start with a letter and can be followed by any number of letters, digits, or underscores. Variables are dynamic binding, which means that you can assign different type of values to the same variable at different time.

For example:

```
>>> a = 25
>>> a = "This is a test!"
>>> a
```

The final content of the variable **a** is "This is a test", a text string not the number 25

Using Python as a calculator

- Type in a math expression.

```
>>> 2 + 2      #This is a comment
>>> (50-5*6)/4
```

Q. What is the result? The sign # indicates this is a comment line and will be ignored when executing the script

The equal sign ("=") is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5*9
>>> width * height
```

Q. What is the result?

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0 # Zero x, y and z
>>> x
>>> y
>>> z
```

Q. Result?

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 3 * 3.75 / 1.5
>>> 7.0 / 2
```

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes or double quotes. Try the following code:

```
>>> print 'spam eggs' 'spam eggs'
>>> print 'doesn\'t'
>>> print "doesn't"
>>> print '"Yes," he said.'
>>> print "\"Yes,\" he said."
>>> print '"Isn\'t," she said.'
```

Python operators			
Operator	**Name**	**Example**	**Output**
+	Addition	4+5	9
-	Subtraction	8-5	3
*	Multiplication	4*5	20
/	Division	19/3	6
%	Remainder	19%3	1
**	Exponent	2**4	16

String literals can span multiple lines in several ways. Continuation lines can be used, with a backslash as the last character on the line indicating that the next line is a logical continuation of the line:

```
>>> hello = "This is a rather long string containing\n \
```

several lines of text just as you would do in C.\n"

```
>>> hello
>>> print hello
```

Basic Python structure

- **Comments** (**#** at beginning of the line)
- **Line continuation** (\ at the end of a line)
- **Variable** is a name that represents a value:
    - Is bound to a value

- Value is not stored in variable, variable just refers to (points to) a value stored somewhere in memory

- **Expression** represents values (is something)

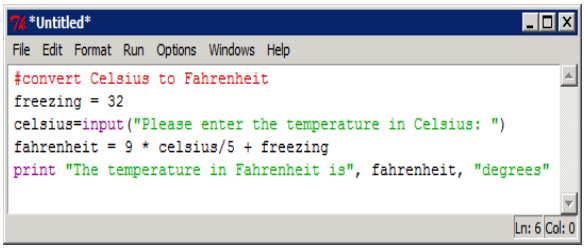
```
>>> 2+2 (resulted 4)
>>> "My string"
```

- **Statement** is an instruction (*does/changes* something)

```
>>> x = 2 (assignment)
>>> print x
```

As mentioned before, you can run python from a script file. Now let's try to write a very simple program.

- Create a new folder **python** in your geog413 folder
- From the python shell window, click File ->New Window in python shell window to open a code window.
- Type in the following code and save it as **cl-fa.py** under your **python** folder. This is a very simple program code to convert Celsius to Fahrenheit



There is a default color coding schema. The lines showing in red are comment lines, reserve key words (command) show in purple, the text string is in green.

- In Python code window, click Run->Check Module. This allows you to validate your module to check if there is any syntax error. If there is any error such, it will be highlighted in the code window, which makes it easy to correct it.
- Click Run -> Run Module to run the program.
- Watch the Python shell window, you will be asked to provide input temperature in Celsius
- Type in 35

Please enter the temperature in Celsius: **35**

You will have the result as the following:

The temperature in Fahrenheit is 95 degrees

Now create another python script to calculate the area of a shape and save the file to **cal_area.py** in your python folder. The program should allow user to select the shapes they want to calculate the area. Two shapes are available to be calculated: Rectangle and Circle.

- Copy and Paste the following code to a Python code window and save it as **cal_area.py.** Make sure the code have the correction indentation as Python use the indentation to define the program block

```
# Area calculation program
print "Welcome to the Area calculation program"
print "---------------------------------------"
print ""
# Print out the menu:
print "Please select a shape:"
print "1 Rectangle"
print "2 Circle"
# Get the user's choice:
shape = input(">")
# Calculate the area:
if shape == 1:
     height = input("Please enter the height: ")
     width = input("Please enter the width: ")
     area = height * width
     geometry = "Rectangle"
else:
     radius = input("Please enter the radius: ")
```

```
    area = 3.14 * (radius * radius)
    geometry = "Circle"
print "\nThe area of ", geometry, " is ", area, "\n"
```

- Check the module and run this program. You will get the result similar as the following. The text in red is users' input.

```
Welcome to the Area calculation program
---------------------------------------
Please select a shape:
1 Rectangle
2 Circle
> 1
Please enter the height: 2
Please enter the width: 3

The area of the Rectangle is 6
```

In this program we use a python statement called IF statement. IF statement allows you to control the data flow based on a condition. Here the condition is **shape** whose value is from user's input. If the value of SHAPE is 1 (the condition is true), then the program will execute the statements of the indented part below the condition **shape == 1**, otherwise the program will execute the statements below the **else** statement. Indentation is important in Python. It indicates the level of execution.

The operator "==" is a local boolean operator (comparison), which compare whether two values are equal. The following table shows the boolean operators used in Python for comparison.

Boolean operators	
Operators	**Function**
<	less than
<=	less that or equal to
>	greater than
>=	greater than or equal to
!=	not equal to
<>	not equal to (alternate)
==	equal to

Now let's add four more lines as the following (red) in to this script to make this program run a loop to calculate the area depend on user's desire.

```
# Area calculation program
print "Welcome to the Area calculation program"
print "---------------------------------------"
print ""

calculate=1
while calculate != 0:
        # Print out the menu:
        print "Please select a shape:"
        print "1 Rectangle"
        print "2 Circle"
        # Get the user's choice:
        shape = input(">")
        # Calculate the area:
        if shape == 1:
                height = input("Please enter the height: ")
                width = input("Please enter the width: ")
                area = height * width
                geometry = "Rectangle"
        else:
                radius = input("Please enter the radius: ")
                area = 3.14*(radius*radius)
                geometry = "Circle"
        print "\nThe area of the", geometry, " is ", area, "\n"
        calculate = input("Do you want to calculate a area for another shape?(1-Yes|0-No):")
print "\nThank you for using our software!"
```

- Run this program. The result should look similar to the following.

```
Welcome to the Area calculation program
---------------------------------------
Please select a shape:
1 Rectangle
2 Circle
```

```
> 2
Please enter the radius: 5
The area of Circle is 78.5
Do you want to calculate another shape? (1 - Yes | 0 - No): 1
Please select a shape:
1 Rectangle
2 Circle
> 1
Please enter the height: 4
Please enter the width: 6

The area of the Rectangle is 24

Do you want to calculate another shape? (1 - Yes | 0 - No): 0
Thank you for using our software!
```

Write a python script to calculate the average and sum of a list of numbers (10 numbers). The script should allow user to input the numbers and print out the average and sum.

3. Using Python script in ArcGIS

Python can be used in ArcGIS to perform many tasks, automate your work, customize ArcGIS and develop new application etc. All geoprocessong tools can be used in Python with the Geoprocessing object,

By definition, Python scripts are modules, which is Python's highest level of code organization. Modules serve several purposes;

- The most obvious is the ability to save code to a reusable form, a script.
- Scripts can avoid duplication of code by using programs contained in other modules.
- Using the import statement, a program can import any number of functions, variables, or both contained in another module.

A module creates a natural grouping and naming structure, which eliminates ambiguity when using commonly named programs. Each time you create a script, you will import one or more modules except you are doing very simple programming as we did in the previous section.

You can write a Python script to execute and make use of a geoprocessing service in multiple ways. The primary way to execute a script is to make use of ArcPy. ArcPy has built-in methods to connect to, execute, and handle the result from the service. The majority of scripts will connect to and use geoprocessing services through ArcPy.

ArcPy (often referred to as the ArcPy site package) provides Python access for all geoprocessing tools, including extensions, as well as a wide variety of useful functions and classes for working with and interrogating GIS data. Using Python and ArcPy, you can develop an infinite number of useful programs that operate on geographic data.

Scripts should be well commented. Each script should contain a heading section that describes what this script does, who created it, and when it was created, along with comments throughout the script itself that explain what it is doing. Use the pound symbol (#) at the beginning of a line to indicate a comment, which is a text string that should not be interpreted by Python. All text following a pound symbol will be ignored by the interpreter.

Set default script editor

- First we will set up the default script editor as Python.
- In ArcCatalog/ArcMap, click Geogprocessing->Geoprocessing Options
- Set the Editor to C:\Python27\Lib\idlelib\idle.pyw

Whenever you edit your python script in ArcGIS, it will automatically open it with python editor

- Create a folder **trim1** under your **python** folder
- In PythonShell window, click the File -> New Window to open a code window.
- Click the File menu and click Save As. Name the script **multi_clip.py** and save it in your **python** folder.

- Add the following lines to code window

```
# Import modules
import arcpy, os
from arcpy import env

#Set input workspace
env.workspace = "L:/labs/geog413/data/python/trim1_utm"

#Set the local variables
clipFeature = "L:/labs/geog413/data/geoprocessing/pg_bndy.shp"

#Set the output workspace
outWorkspace = "K:/geog413/python/trim1/"
```

The first three lines import the **os**, **arcpy** modules and **env** module in **arcpy** to the script. The accessing of ArcGIS geoprocessing tools is done through **arcpy** object.

This script statement creates a geoprocessing object, which will have the following arguments so it can be used in a generic fashion. The statements define and set variables based on the user-defined values passed to the script at execution:

- An input workspace defining the set of feature classes to process
- A feature class to be used by the geoprocessing tool.
- An output workspace where the results of the geoprocessing tool will be written

Add the following error-handling statement and geoprocessor call to the code window:

```
try:
   #Get a list of the featureclasses in the workspace folder
   trimList = arcpy.ListFeatureClasses()
```

Python enforces indentation of code after certain statements as a construct of the language. The **try** statement defines the beginning of a block of code that will be handled by its associated exception handler, or **except** statement. All codes within this block must be indented. Python uses **try/except** blocks to handle unexpected errors during execution. Exception handlers define what the program should do when an exception is raised by the system or by the script itself. In this case, you are only concerned about an error occurring with the geoprocessor, so a **try** block is started just before you start to use the object. It is good practice to use exception handling in any script using the geoprocessor so its error messages can be propagated back to the user. This also allows the script to exit gracefully and return informative messages instead of simply causing a system error.

The geoprocessor's **ListFeatureClasses** method returns an enumeration of feature class names in the current workspace. The workspace defines the location of your data and where all new data will be created unless a full path is specified. The workspace has already been set to the first argument's value.

- Add the following code:

```
   #Loop through the list of feature classes and clip the feature class
   for trim in trimList:
       #Set output feature class.
       outFeatureClass = outWorkspace + trim.split('_')[0] + "_clip.shp"

       #Clip each feature class in the list
       arcpy.AddMessage("Clipping " + trim + "   ...")
       print "Clipping " + trim + "   ..."
       arcpy.Clip_analysis(trim, clipFeature, outFeatureClass, "")
except:
    arcpy.AddMessage(arcpy.GetMessages(2))
    print arcpy.GetMessages(2)
```

When there are no more names in the enumeration, a null or empty string will be returned. Python will evaluate this as false, which will cause the while loop to exit. The next value from the enumeration must be retrieved before the end of the indented block so it is evaluated correctly at the start of the loop.

The **Split** method is used to split a string using the separator specified in the bracket. The [0] after the split indicate the first element will be returned.

The **os** module's path object is used to manipulate the clip feature class's path, so only the feature class name is evaluated in an expression, not the entire path. The Clip tool is accessed as a method of the geoponics, using the various string variables as parameter values.

- Save the script by clicking the Save button on the Standard toolbar.

View the completed script

```
##Script Name: Clip_Multiple_Features.
##Description: Clips one or more shapefiles.
##Created By: Name here.
##Date: Date here.

# Import modules
import arcpy, os, string
from arcpy import env

#Set the input workspace
env.workspace = "L:/labs/geog413/data/python/trim1_utm"

#Set the local variables
clipFeature = "L:/labs/geog413/data/geoprocessing/pg_bndy.shp"

#Set the output workspace, replace the path with your own local folder
outWorkspace = "K:/geog413/python/trim1/"

try:
   #Get a list of the featureclasses in the workspace folder
   trimList = arcpy.ListFeatureClasses()

   #Loop through the list of feature classes
   for trim in trimList:
       #Set output feature class
       outFeatureClass = outWorkspace + trim.split('.')[0] + "_clip.shp"

       #Clip each feature class in the list
       arcpy.AddMessage("Clipping " + trim + "   ...")
       print "Clipping " + trim + "   ..."
       arcpy.Clip_analysis(trim, clipFeature, outFeatureClass, "")
except:
    arcpy.AddMessage(arcpy.GetMessages(2))
    print arcpy.GetMessages(2)
```

The **except** statement is required by the earlier **try** statement; otherwise, a syntax error will occur. If an error occurs during execution, the code within **except** block will be executed. In this case, any message with a severity value of 2, indicating an error, will be added to the geoprocessor in case the script is used as the source of a tool. All error messages are also printed to the standard output in case the script is run outside a tool.

- Save the script
- Run the script by clicking the Run->Run the Model.
- In ArcCatalog, check your python directory, you should have ***tctrl_clip.shp, trivr_clip.shp, troad_clip.shp*** and ***tlake_clip.shp*** there.

You can also get a script from the model you built in ModelBuilder by converting the model to script. Let's convert the model for reprojection and clipping you built in previous lab.

- Navigate to your geoprocessing\model folder and find the "**Reprojection & Clipping**" tool under **pg_tools** toolbox.
- Right-click the tool and choose Edit to open the Editing window.
- Click Model->Export->To Python script and give the **proj_clip.py** for the script name under your python folder.
- Open this script in Python script window, here you can examine the script code. You may need some editing if your model does not have proper parameters setting.

4. Building a customized tool

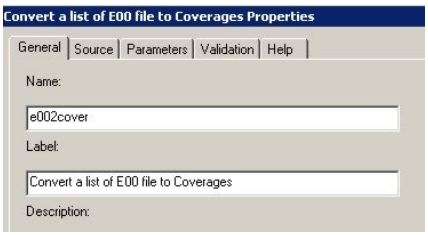
You can build a customized tool with python scrip and add it to ArcToolbox for regular use just like using other ArcToolbox.

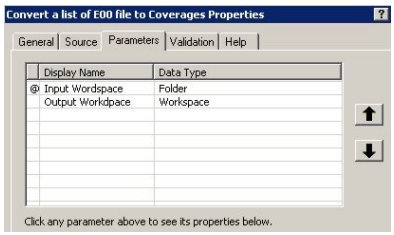
- Open a new python window and copy and paste the following code. Save the file to **e002cover.py** under your python folder.

```
# ---------------------------------------------------------------------------
# Name:  e00-cover.py
# Purpose: Convert a list of E00 files to ArcInfo coverages
# Command line: e00-cover.py <input_workspace> <output_workspace>
# <input workspace>: the directory that have all E00 files
# <output workspace>: the directory where all output data go

# Parameter Setting:
#       Workspace:            Workspace
#       Output Workspace      Workspace
# ---------------------------------------------------------------------------
# Import modules
import arcpy, os
from arcpy import env

env.overwriteOutput = True

# Define workspace as input parameters
env.workspace = arcpy.GetParameterAsText(0)
output_workspace = arcpy.GetParameterAsText(1)

# Get a list of the E00 files from the input_workspace...
list_files = arcpy.ListFiles("*.e00")
arcpy.AddMessage("\nConvert E00 files to ArcInfo coverages ... ")

# Loop through each item in the list to import e00 file to coverage

count = 0
if len(list_files) > 0:   #check if the list is empty
     for each in list_files:
            count = count + 1
            in_e00 = env.workspace + os.sep + each
            out_cover = each.split('.')[0]
            arcpy.AddMessage("Converting " + each + " to coverage ... ")
            arcpy.ImportFromE00_conversion(in_e00, output_workspace, out_cover)

arcpy.AddMessage("\n" + str(count) + " E00 files imported to ArcInfo coverages\n")
```

You probably notice that there are two lines above in bold face. This allows for taking user input when running the script.

- In ArcCatalog, go to your Python folder. Right-click the folder and choose New Toolbox. Name the toolbox as MyTools
- Right-click on the MyTools and choose Add Script. A window pops up allowing for setting the parameter
- Give the "e002cover' for Name and "Convert a list of E00 files to Coverage" for Label. Click Next button





- Here you need to specify where the script file is located. Click the open folder button and navigate to your python folder and double click the script file **e002cover.py**. Click Next again.
- Now you can set the script argument that will take user's input (Parameters tab)
- Click the first empty place under Display Name and type in "Input Workspace". Click the corresponding place for under Data Type and choose Folder from the dropdown list. This define the data type of the parameter "Input Workspace"

- Now set the second parameter "Output Workspace" as Workspace
- Click OK to finish. You will have a tool (Convert a list of E00 files to Coverages) listed under your MyTools
- Now double-click the tool to run it. A window pops up allowing for parameters.

Now you can specify the input workspace and output workspace.

- Set the Input Workspace to L:\labs\geog413\data\python\e00
- Set the Output Workspace to K:\geog413\python.
- Click OK. When finished, you will have four coverage dataset converted from E00 file.
- Examine your **python** folder, you should have a list of coverage datasets there (***contours, lakes, rivers, roads***).

Here is another example to merge a list of datasets into a new dataset.

Examine the folder at L:\labs\geog413\data\python\trim1. You should see that there are four folders (93g010, 93g020, 93g009, 93g019) that indicate four map sheets and each folder has a list of datasets.

Now we will merge four datasets from each folder to produce four merged dataset for ***tlake, troad, trivr,*** and ***twtrl*** in your local folder.

Copy the following text into a new python window and save it to **merge.py**

```
# ---------------------------------------------------------------------------
# Version: ArcGIS 10.1
# Functionality: Merge a list of datasets to a new dataset
# Description: merge.py <input_workspace> <output_workspace>
# <input workspace>: the directory that have all E00 files
# <output workspace>: the directory where all output data go

# Parameter Setting:
#       Workspace:              Workspace
#       Output Workspace        Workspace
# ---------------------------------------------------------------------------

# Import modules
import arcpy, os
from arcpy import env

env.overwriteOutput = True

# Define workspace as input parameters
env.workspace = arcpy.GetParameterAsText(0)
output_workspace = arcpy.GetParameterAsText(1)

# Define local variables
sheet_list = ['93g009', '93g010', '93g019', '93g020']
data_list = ['tlake.shp', 'trivr.shp', 'troad.shp', 'twtrl.shp']

for data in data_list:
        arcpy.AddMessage("\nMerging " + data + '...')
        merge_string = []
        out_data = output_workspace + os.sep + data

        for sheet in sheet_list:
             dataset = env.workspace + '/'+ sheet + '/'+ data
             arcpy.AddMessage(" Adding " + sheet + '/'+ data)
             merge_string.append(dataset)

        #Merging datasets
        arcpy.Merge_management(merge_string, out_data)
```

- Add a Script under **MyTools**
- Set the Name to **merge-data** and "Merge a list of datasets into a new dataset" for Label
- Set the Script source to the file ***merge.py*** from your python folder
- Set parameters "Input workspace" with Folder type and "Output workspace" with Folder type
- Run this tool

- Set the Input Workspace to L:\labs\geog413\data\python\trim1and Output Workspace to K:\geog413\python.
- Click OK button to run the tool. When it is done, you will have four merged datasets: ***tlake.shp, troad.shp, trivr.shp*** and ***twtrl.shp*** under your python folder.

Write a script to automate the raster surface interpolation in Statistical Analysis lab

- First in ArcMap Join the station attribute table (**climate_data.dbf**) to station and export it as ***station_attr.shp*** in your local folder. Use this layer as the input point feature.

The following script will take the point feature (***station.shp)*** as the input point feature and take the workspace you specified as the output workspace which is the location for the output datasets

```
# Name: climate.py
# Description:  Interpolate a series of point features onto a rectangular raster
# Requirements: 3D Analyst & Spatial Analyst Extension
# Parameter Setting:
       InPoint Feature:       Feature Layer
       Output Workspace       Workspace

# Import system modules
import arcpy, string, os
from arcpy import env
from arcpy.sa import *

# Check out the ArcGIS Spatial Analyst extension license
arcpy.CheckOutExtension("3D")
arcpy.CheckOutExtension("Spatial")

# Set input and environment variables
inPoint = arcpy.GetParameterAsText(0)       # The point feature must come with interpolation field
env.workspace = arcpy.GetParameterAsText(1) # Where the resulting data will be
env.overwriteOutput = True

# Set local variables
cellSize = 200
splineType = "REGULARIZED"
weight = 0.1

# Define the data fields that server as Zfield (interpolation)
mean = ["mean_71_75", "mean_76_80", "mean_81_85", "mean_86_90"]
rain = ["rain_71_75", "rain_76_80", "rain_81_85", "rain_86_90"]
snow = ["snow_71_75", "snow_76_80", "snow_81_85", "snow_86_90"]
prec = ["prec_71_75", "prec_76_80", "prec_81_85", "prec_86_90"]

all_fields = mean + rain + snow + prec

# Loop through each data field and generate the surface for each five-year
for field in all_fields:
    zField = field
    out_data = field.split('_')[0] + field.split('_')[2]

    # Execute Spline
    arcpy.AddMessage("\nGenerating surface for " + field + "  ...")
    arcpy.Spline_3d(inPoint, zField, out_data, cellSize, splineType, weight)
# Generating overall statistics
for item in ["mean", "rain", "snow", "prec"]:
    if item == "mean":
        name = "temp"
    else:
        name = item
    for stat in ["MEAN", "MAXIMUM", "MINIMUM", "RANGE"]:
        arcpy.AddMessage ("\nGenerating overall statistics " + name + " " + stat + " ...")
        outCellStat = CellStatistics([item + "75", item + "80", item + "85", item + "90"], stat, "NODATA")
        outCellStat.save(name + "_" + stat)
```

- Save the script as **climate.py**. Add the script to MyTool and give a try to run it. (Note: make to set correct environments)

The following is the python script for the case study (Wildlife Habitat) in 300 lab

- Create a folder **mmf** under your **python** folder
- Read the following script carefully, replace the path of workspaces (input and output) and run the script.
- Check the datasets you have in your output workspace.

```
# ---------------------------------------------------------------------------
# wildlefe_habitat.py
# Locate the potential wildlife area
# Parameter Setting:
#      Workspace:         Workspace
#      Output Workspace:  Workspace
# ---------------------------------------------------------------------------
# Import system modules
import arcpy, os
from arcpy import env

# Set the input/output workspace
env.workspace = arcpy.GetParameterAsText(0)   # Where the data used for analysis
out workspace = arcpy.GetParameterAsText(1)   # Place for the resulting data

# Local variables for input data
forest = arcpy.env.workspace + "/forest.shp"
rivers = arcpy.env.workspace + "/rivers.shp"
soils = arcpy.env.workspace + "/soils.shp"

# Local variables for output data
factor1 = out workspace + "/factor1.shp"
factor1_50m = out_workspace + "/factor1_50m.shp"
rivers250m = out_workspace + "/rivers250m.shp"
factor1_500m = out_workspace + "/factor1_500m.shp"
factor2 = out workspace + "/factor2.shp"
factor3 = out workspace + "/soil selected.shp"
factor12 = out_workspace + "/fact12.shp"
Wildlife_Habitat_Area = out_workspace + "/final.shp"

# Process: SELECT Soil - Mg, -Ct
arcpy.Select_analysis(soils, factor3, "\"SOIL_SY1\" = 'Mg' OR \"SOIL_SY1\" = 'Ct'")

# Process: SELECT, HTCL_IN>1 AGECL_IN>=3 CRNCLCL=5,6,7 SPCS1="S"
arcpy.Select_analysis(forest, factor1, "\"HTCL_IN\">1 AND \"AGECL_IN\">=3 AND ( \"CRNCLCL\" =5 OR
\"CRNCLCL\"=6 or \"CRNCLCL\"=7 ) AND \"SPCS1\" = 'S'")

# Process: BUFFER50m on selected forest
arcpy.Buffer analysis(factor1, factor1 50m, "50 Meters", "FULL", "ROUND", "ALL", "")

# Process: BUFFER250m on rivers
arcpy.Buffer_analysis(rivers, rivers250m, "250 Meters", "FULL", "ROUND", "ALL", "")

# Process: BUFFER500m on selected forest
arcpy.Buffer analysis(factor1, factor1 500m, "500 Meters", "FULL", "ROUND", "ALL", "")
# Process: INTERSECT to produced factor2
arcpy.Intersect_analysis(rivers250m + ";" + factor1_500m, factor2, "ALL", "", "INPUT" )

# Process: UNION
arcpy.Union_analysis(factor1_50m + ";" + factor2, factor12, "ALL", "", "GAPS")

# Process: INTERSECT fact1, fact2 & fact3
arcpy.Intersect_analysis(factor3 + ";" + factor12, Wildlife_Habitat_Area, "ALL", "", "INPUT")
print "The End"
```

The following script is doing the zonal statistics analysis for overlapped polygon zones

```
#-------------------------------------------------------
# zonal stat.py
# Perform zonal statistics for overlapped polygon zones
# Parameter Setting:
#       Zone Layer:          Feature Layer
#       Zone Field           Field
#       Statistics Raster:   Raster Layer
#      Output Location:     Workspace
#      Output Table:         String
#-------------------------------------------------------

#Import modules
```

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

#--- Define function ------------------
# function: Get unique values from a field
def getUniqueValue(inputTable, field):
    valueList = []    # array to hold list of values collected
    valueSet = set()  # set to hold values to test against to get list
    rows = arcpy.SearchCursor(inputTable) # create search cursor

    # iterate through table and collect unique values
    for row in rows:
        value = row.getValue(field)

        # add value if not already added and not current year
        if value not in valueSet:
            valueList.append(value)

        # add value to valueset for checking against in next iteration
        valueSet.add(value)

    # return value list
    valueList.sort()
    return valueList

#--- Main Program ------------------
#Set up the environment
arcpy.env.overwriteOutput = True
arcpy.CheckOutExtension("Spatial")

# Get input variable from user

zoneLayer = arcpy.GetParameterAsText(0)
zoneID = arcpy.GetParameterAsText(1)
statLayer = arcpy.GetParameterAsText(2)
outWorkspace = arcpy.GetParameterAsText(3)
tableName = arcpy.GetParameterAsText(4)

arcpy.env.workspace = outWorkspace
outTable = os.path.join(outWorkspace, tableName)
tempTable = "temp_" + tableName
featureLayer = "featureLayer"

arcpy.MakeFeatureLayer management(zoneLayer, featureLayer)

# Get the total number of zones (zoneLayer)
result = arcpy.GetCount_management(zoneLayer)
count = int(result.getOutput(0))

# looping through each zone
uniqueValue = getUniqueValue(zoneLayer, zoneID)

for buff in uniqueValue:
    arcpy.AddMessage("Calculating the statistics for " + zoneID + ": " + str(buff))
    where = '"' + zoneID + '"' + ' = ' + str(buff)
    arcpy.SelectLayerByAttribute management(featureLayer, "NEW SELECTION", where)
    if buff == 0:
        arcpy.AddMessage("    Generateing statistics table ...\n")
        outZSaT = ZonalStatisticsAsTable(featureLayer, zoneID, statLayer, outTable)
    else:
        outZSaT = ZonalStatisticsAsTable(featureLayer, zoneID, statLayer, tempTable)
        arcpy.AddMessage("    Appending the result ...\n")
        arcpy.Append management(tempTable, outTable)

arcpy.Delete_management(tempTable)
```