Chapter 1: Python Basics

Notice to Students

Dear Students,

We understand that some of you may find it challenging to grasp the complete process at first. Please do not worry. It is perfectly normal to encounter difficulties when learning new concepts.

To help you succeed, we encourage you to recap what you have learned in your Basic Programming course from Year 2, Semester 2. Focus on understanding the basic programming syntax and workflow, such as loops and if statements.

By the end of Unit One, specifically at the end of Chapter I of this course, you should aim to:

- Understand the basic structure of a Python code.
- Identify the input, intermediate parts, and final output of the code.
- Find and correct errors in the code.

Remember, practice is key. Try to apply what you have learned and do not hesitate to ask questions if you need further clarification. We are here to support you.

Best regards,

Your instructor

1 Introduction to Scripting for Geospatial Data Analysis

1.1 Introduction

This session demonstrates the capabilities of scripting for geospatial data analysis and highlights some characteristics of the Python programming language. By the end of this session, you will understand the power of scripting in GIS workflows and why Python is a preferred language for GIS programming.

1.2 Session Objectives

The objectives of this session are to articulate in general terms what scripting can do for GIS workflows, explain why Python is selected for GIS programming, and demonstrate how to run code in Python, ArcGIS Python Window, and/or Visual Studio Code.

1.3 GIS and Scripting

At the core of GIS is geographic data analysis. This analysis often needs to be repeated across multiple fields, files, and directories, sometimes on a monthly or even daily basis, and it may need to be performed by multiple users. Scripting can significantly streamline these processes.

1.4 Use of Analysis Functions

GIS is used to answer complex spatial questions. For example, spatial analysis can help determine the land cover status of areas within proximity to selected streams. This involves identifying and mapping areas within a specified distance from the streams, analyzing the land cover types within these areas, and determining the proportion of different land cover types.

1.5 Scripting/Programming

Scripting increases productivity and facilitates sharing among users. It involves common data management activities such as reformatting data, copying files for backups, and searching database content. Scripting offers efficient batch processing, automated file reading and writing, running routine geo-processing tasks, and modeling.

1.5.1 Scripts

Scripts can access or modify GIS datasets and their fields and records, performing analysis at any of these levels. For example, a script can buffer a river and clip the land cover data to analyze the impact of proximity to water bodies.

1.5.2 Sample Script: Buffering a River and Clipping Land Cover in ArcGIS or QGIS

Here is a sample script for buffering a river and clipping the land cover in ArcGIS, focusing only on the land cover analysis for 2015:

1.5.2.1 ArcGIS Python Script

```
import arcpy

# Set environment settings
arcpy.env.workspace = "C:/hawassadatabase/data_vector"
arcpy.env.overwriteOutput = True

# Input data
river = "rivers.shp"
land_cover = "land_cover_2015.shp"

# Select river by attribute
river_name = "Wehsa River"
river_layer = arcpy.management.MakeFeatureLayer(river, "river_layer",
f"rivername = '{river_name}'")

# Buffer the river
buffer_output =
f"C:/hawassadatabase/analysisoutput/{river_name.replace(' ',
'_')}_buffer_1000m.shp"
arcpy.Buffer_analysis(river_layer, buffer_output, "1000 Meters")

# Clip the land cover with the river buffer
clip_output =
f"C:/hawassadatabase/analysisoutput/{river_name.replace(' ',
'_')}_buffer_1000m_land_cover_2015.shp"
arcpy.Clip_analysis(land_cover, buffer_output, clip_output)

print("Buffering and clipping completed successfully.")
```

1.5.2.2 QGIS Python Script (PyQGIS)

```
from qgis.core import (
    QgsVectorLayer,
    QgsProcessingFeatureSourceDefinition,
    QgsProcessingContext,
    QgsProcessingFeedback,
    QgsProject
)
import processing

# Set environment settings
data_vector_path = "C:/hawassadatabase/data_vector/"
output_path = "C:/hawassadatabase/analysisoutput/"

# Input data
river = data_vector_path + "rivers.shp"
land_cover = data_vector_path + "land_cover_2015.shp"

# Select river by attribute
river_name = "Wehsa River"
river_layer = QgsVectorLayer(river, "rivers", "ogr")
river_layer.setSubsetString(f"rivername = '{river_name}'")

# Buffer the river
buffer_output = output_path + f"{river_name.replace(' ',
'_')}_buffer_1000m.shp"
buffer_params = {
    'INPUT': river_layer,
    'DISTANCE': 1000,
    'OUTPUT': buffer_output
}
processing.run("native:buffer", buffer_params)

# Clip the land cover with the river buffer
clip_output = output_path + f"{river_name.replace(' ',
'_')}_buffer_1000m_land_cover_2015.shp"
clip_params = {
    'INPUT': land_cover,
    'OVERLAY': buffer_output,
    'OUTPUT': clip_output
}
processing.run("native:clip", clip_params)

print("Buffering and clipping completed successfully.")
```

1.6 Python and GIS

Python, created by Guido van Rossum, is an ideal programming language for GIS users for several reasons. Python is easy to learn, object-oriented, and widely supported. It is embraced by the GIS community and comes integrated with many GIS applications.

1.6.1 Why Python?

Python is easy to pick up, object-oriented, and has abundant resources for help. It is widely embraced by the GIS community and comes integrated with many GIS applications, making it a powerful tool for geospatial analysis.

1.6.2 Sample Data and Scripts for the Training

Sample data and scripts for this training session are available in the GIS lab. These resources will help you practice and understand the concepts discussed.

1.7 GIS Data Formats

GIS data can come in various formats, including GRID rasters, geodatabases, shapefiles, and PostgreSQL + PostGIS. Understanding these formats is crucial for effective data management and analysis.

1.8 Basic Benefits of Using Python in GIS (ArcGIS/QGIS)

1.8.1 Benefits

Using Python in GIS offers numerous benefits. It saves work time and money, automates redundant geoprocessing tasks, eliminates human error, and increases productivity. Python scripts can be run in toolboxes and scheduled for regular tasks, making them highly versatile. Importantly, no prior programming experience is required to start using Python for GIS.

1.8.2 Example: Producing Land Cover Maps with Different Buffer Distances and for Selected Rivers

To illustrate the benefits of using Python in GIS, consider the task of producing land cover maps with different buffer distances (e.g., 100, 200, 300, 500 meters) for selected rivers (e.g., Wehsa River, Werka River, Bele River) over different years (e.g., 2015, 2020, 2024). This involves:

1. **Buffering the Rivers**: Creating buffer zones around each river at specified distances.
2. **Clipping Land Cover Data**: Clipping the land cover data within these buffer zones.
3. **Analyzing Land Cover Types**: Analyzing the land cover types within each buffer zone for different years.

By scripting these tasks, you can automate the process, ensuring consistency and efficiency. Here is an example script for ArcGIS:

```
import arcpy

# Set environment settings
arcpy.env.workspace = "C:/hawassadatabase/data_vector"
arcpy.env.overwriteOutput = True

# Input data
rivers_shapefile = "rivers.shp"
land_cover_years = ["land_cover_2015.shp", "land_cover_2020.shp", "land_cover_2024.shp"]
buffer_distances = [100, 200, 300, 500]

# Extract rivers based on rivername attribute
river_names = ["Wehsa River", "Werka River", "Bele River"]

for river_name in river_names:
    # Select river by attribute
    river_layer = arcpy.management.MakeFeatureLayer(rivers_shapefile, "river_layer", f"rivername = '{river_name}'")

    for distance in buffer_distances:
        buffer_output = f"C:/hawassadatabase/analysisoutput/{river_name.replace(' ', '_')}_buffer_{distance}m.shp"
        arcpy.Buffer_analysis(river_layer, buffer_output, f"{distance} Meters")

        for land_cover in land_cover_years:
            year = land_cover.split('_')[2].split('.')[0]
            clip_output = f"C:/hawassadatabase/analysisoutput/{river_name.replace(' ', '_')}_buffer_{distance}m_land_cover_{year}.shp"
            arcpy.Clip_analysis(land_cover, buffer_output, clip_output)

print("Buffering and clipping completed successfully.")
```

This script automates the creation of buffer zones around each river at specified distances and clips the land cover data within these buffer zones for different years. This process can be repeated for different buffer distances and rivers, ensuring comprehensive spatial analysis.

1.9 Exercise

1.9.1 Questions based on the above script provided

These questions will help you understand the efficiency and accuracy benefits of scripting in GIS workflows compared to manual methods.

The questions based on the example "Producing Land Cover Maps with Different Buffer Distances and for Selected Rivers"

- **Number of Input Files**: How many input files are used in the script? List them.
- **Number of Analysis Functions**: Identify the analysis functions used in the script. How many distinct functions are there?
- **Number of Times the Functions are Run**: How many times is each analysis function executed in the script?
- **Advantages of Using the Script Compared to the Manual GUI Method**:
  - What are the advantages of using a script to perform GIS tasks compared to using the manual GUI method (e.g., clicking and opening each dialog for buffer, selecting files, entering parameters, and assigning output names)?
  - Discuss the potential for errors when using the manual GUI method, such as mixing up output names or incorrectly entering parameters. How does scripting help mitigate these errors?

1.9.2 Answer

The answers based on the example "Producing Land Cover Maps with Different Buffer Distances and for Selected Rivers"

- **Number of Input Files**:
- The script uses multiple input files:
  - *rivers.shp* (the shapefile containing river data)
  - *land_cover_2015.shp* (the shapefile containing land cover data for the year 2015)
  - *land_cover_2020.shp* (the shapefile containing land cover data for the year 2020)
  - *land_cover_2024.shp* (the shapefile containing land cover data for the year 2024)
- **Number of Analysis Functions**:
  The script uses two distinct analysis functions:
  - *Buffer_analysis* (to create buffer zones around the rivers)
  - *Clip_analysis* (to clip the land cover data within the buffer zones)
- **Number of Times the Functions are Run**:
  - The *Buffer_analysis* function is executed 12 times (3 rivers × 4 buffer distances).
  - The *Clip_analysis* function is executed 36 times (3 rivers × 4 buffer distances × 3 land cover years).
- **Advantages of Using the Script Compared to the Manual GUI Method**:
  - **Efficiency**: The script automates the entire process, saving significant time compared to manually opening dialogs, selecting files, entering parameters, and assigning output names for each operation.
  - **Consistency**: The script ensures that the same parameters and processes are applied consistently across all operations, reducing the risk of human error.
  - **Error Reduction**: Using a script minimizes the possibility of errors such as mixing up output names or incorrectly entering parameters, which can occur when using the manual GUI method.
  - **Reproducibility**: The script can be easily reused or modified for similar tasks, ensuring reproducibility of the analysis.
  - **Batch Processing**: Scripts can handle batch processing of multiple datasets or operations, which would be cumbersome and time-consuming to perform manually.

- **Scalability**: The script can be scaled to handle larger datasets or additional parameters without a significant increase in manual effort.

These answers should help students understand the practical benefits of scripting in GIS workflows compared to manual methods. If you need any further assistance or additional questions, feel free to ask!

1.10 Reading assignment

1.10.1 Explore the befits of programming in GIS

Activity:

- Conduct online research to find project methodologies and processes applied in real-world GIS projects. Summarize the methodologies and highlight how programming played a role in these projects. Proposed existing projects:
-
- Based on your research and coursework, propose a small project where programming in GIS could be beneficial. Outline the steps you would take and the expected outcomes.

Submission:

- Prepare a detailed report addressing the points above. Include references to your coursework, online resources, and any additional readings.

1.10.2 Recap the python syntax (from programming course / last semester)

- Based on the three codes below try to recap what you have learnt in the basic programing course in year

```
Import
Function
List
 = sign
Indention
Loop
Comment
Print
Keywords
```

- What are the differences among the three examples? Remember all the three codes produces the same results!

1.10.3 The three example codes

These examples are designed to calculate distances between pairs of UTM coordinates, specifically focusing on horizontal distance, surface (slope) distance, and vertical variation.
For details on coordinates, distances refer to the GIS courses and surveying!

1.10.3.1 Example 1: Using Functions and Additional Packages

```
# This example uses numpy and pandas to calculate distances and
display the results in a table.
import numpy as np
import pandas as pd

# Function to calculate horizontal distance
def horizontal_distance(x1, y1, x2, y2):
    return np.sqrt((x2 - x1)**2 + (y2 - y1)**2)

# Function to calculate surface (slope) distance
def surface_distance(x1, y1, z1, x2, y2, z2):
    return np.sqrt((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)

# Function to calculate vertical variation
def vertical_variation(z1, z2):
    return abs(z2 - z1)

# Sample UTM coordinates (x, y, z) for five paired points in Ethiopia
points = [
    (500000, 1000000, 1500, 500100, 1000100, 1600),
    (500200, 1000200, 1550, 500300, 1000300, 1650),
    (500400, 1000400, 1600, 500500, 1000500, 1700),
    (500600, 1000600, 1650, 500700, 1000700, 1750),
    (500800, 1000800, 1700, 500900, 1000900, 1800)
]

# Calculate distances and variations
data = []
for p in points:
    x1, y1, z1, x2, y2, z2 = p
    h_dist = horizontal_distance(x1, y1, x2, y2)
    s_dist = surface_distance(x1, y1, z1, x2, y2, z2)
    v_var = vertical_variation(z1, z2)
    data.append([x1, y1, z1, x2, y2, z2, h_dist, s_dist, v_var])

# Create a DataFrame
df = pd.DataFrame(data, columns=['X1', 'Y1', 'Z1', 'X2', 'Y2', 'Z2',
'Horizontal Distance (m)', 'Surface Distance (m)', 'Vertical Variation (m)'])

# Display the table
print("Example 1: Using Functions and Additional Packages")
print(df)
print("\n")
```

1.10.3.2 Example 2: Using Functions without Additional Packages

```
# This example performs the same calculations but without using numpy
or pandas.

# Function to calculate horizontal distance
def horizontal_distance_no_pkg(x1, y1, x2, y2):
    return ((x2 - x1)**2 + (y2 - y1)**2)**0.5

# Function to calculate surface (slope) distance
def surface_distance_no_pkg(x1, y1, z1, x2, y2, z2):
```

```
    return ((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)**0.5

# Function to calculate vertical variation
def vertical_variation_no_pkg(z1, z2):
    return abs(z2 - z1)

# Calculate distances and variations
data_no_pkg = []
for p in points:
    x1, y1, z1, x2, y2, z2 = p
    h_dist = horizontal_distance_no_pkg(x1, y1, x2, y2)
    s_dist = surface_distance_no_pkg(x1, y1, z1, x2, y2, z2)
    v_var = vertical_variation_no_pkg(z1, z2)
    data_no_pkg.append([x1, y1, z1, x2, y2, z2, h_dist, s_dist, v_var])

# Display the table
print("Example 2: Using Functions without Additional Packages")
print("X1\tY1\tZ1\tX2\tY2\tZ2\tHorizontal Distance (m)\tSurface
Distance (m)\tVertical Variation (m)")
for row in data_no_pkg:
    print("\t".join(map(str, row)))
print("\n")
```

1.10.3.3 # Example 3: Without Using Functions

```
# This example calculates the distances directly within the loop
without defining separate functions.

# Calculate distances and variations
data_no_func = []
for p in points:
    x1, y1, z1, x2, y2, z2 = p
    h_dist = ((x2 - x1)**2 + (y2 - y1)**2)**0.5
    s_dist = ((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)**0.5
    v_var = abs(z2 - z1)
    data_no_func.append([x1, y1, z1, x2, y2, z2, h_dist, s_dist,
v_var])

# Display the table
print("Example 3: Without Using Functions")
print("X1\tY1\tZ1\tX2\tY2\tZ2\tHorizontal Distance (m)\tSurface
Distance (m)\tVertical Variation (m)")
for row in data_no_func:
    print("\t".join(map(str, row)))
```

1.10.4 Explanation to the above codes

1.10.4.1 Explanation of Differences:

Example 1: Using Functions and Additional Packages

- Utilizes *numpy* for mathematical calculations and *pandas* for creating and displaying a DataFrame.
- Functions are defined to calculate horizontal distance, surface distance, and vertical variation.
- Results are displayed in a well-formatted table using *pandas*.

Example 2: Using Functions without Additional Packages

- Performs the same calculations as Example 1 but without using *numpy* or *pandas*.
- Functions are still defined for the calculations.
- Results are displayed using basic Python print statements.

Example 3: Without Using Functions

- Calculates the distances directly within the loop without defining separate functions.
- This approach is more straightforward but less modular and reusable.
- Results are displayed using basic Python print statements.

1.10.4.2 Explanation to the above codes

This section provides break down the general objective and the key components of the provided code examples. These examples are designed to calculate distances between pairs of UTM coordinates, specifically focusing on horizontal distance, surface (slope) distance, and vertical variation.

General Objective

The main goal of these scripts is to:

1. **Calculate Horizontal Distance**: The straight-line distance between two points in the horizontal plane (ignoring elevation).
2. **Calculate Surface (Slope) Distance**: The straight-line distance between two points considering the elevation difference.
3. **Calculate Vertical Variation**: The absolute difference in elevation between two points.

Key Components of the Code

Example 1: *Using Functions and Additional Packages*

- **Imports**: Uses *numpy* for mathematical operations and *pandas* for creating and displaying a DataFrame.
- **Functions**: Defines three functions to calculate horizontal distance, surface distance, and vertical variation.
- **Sample Data**: Provides sample UTM coordinates for five paired points in Ethiopia.
- **Calculations**: Uses the defined functions to calculate the distances and variations.
- **DataFrame**: Stores the results in a *pandas* DataFrame and prints it in a table format.

Example 2: *Using Functions without Additional Packages*

- **No Imports**: Does not use any additional packages.
- **Functions**: Similar to Example 1, but the calculations are done using basic Python operations.
- **Sample Data**: Same sample UTM coordinates as in Example 1.
- **Calculations**: Uses the defined functions to calculate the distances and variations.

- **Print Statements:** Displays the results using basic print statements.

Example 3: *Without Using Functions*

- **No Functions:** Performs the calculations directly within the loop.
- **Sample Data:** Same sample UTM coordinates as in the previous examples.
- **Calculations:** Directly calculates the distances and variations within the loop.
- **Print Statements:** Displays the results using basic print statements.

1.10.4.3 Explanation of the Calculations

1. **Horizontal Distance:**
   - Formula: (sqrt{(x2 - x1)^2 + (y2 - y1)^2})
   - This calculates the straight-line distance between two points in the horizontal plane.
2. **Surface (Slope) Distance:**
   - Formula: (sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2})
   - This considers the elevation difference, providing the actual distance over the terrain.
3. **Vertical Variation:**
   - Formula: (|z2 - z1|)
   - This gives the absolute difference in elevation between two points.

1.10.5 Recap for Basic Programming

- **Functions:** Encapsulate reusable code blocks, making the script modular and easier to maintain.
- **Loops:** Iterate over data points to perform repetitive calculations.
- **Basic Math Operations:** Use basic arithmetic operations and the square root function to perform distance calculations.
- **Data Structures:** Use lists to store and manage data points and results.
- **Print Statements:** Display results in a readable format.

These examples demonstrate how to apply basic programming concepts to solve a specific problem in geospatial analysis.

2 Beginning Python: syntax, variables

2.1 Overview

Before you can create GIS Python scripts, you need to understand where to write and run the code, and have a familiarity with basic programming concepts. This unit will guide you through these foundational elements.

2.2 What is this unit about?

This unit discusses Python development software for Windows operating systems. It covers:

- **Interactive Mode and Scripting**: Learn the difference between writing code interactively and scripting. Interactive mode allows you to test code snippets quickly, while scripting involves writing complete programs.
- **Running Scripts with Arguments**: Understand how to pass arguments to your scripts from the command line, which can make your scripts more flexible and powerful.
- **Fundamental Characteristics of Python**: Get to know the core features of Python that make it a powerful and easy-to-use programming language.

Some Fundamental Characteristics of Python

- **Comments**: Comments are used to explain what your code does. They are ignored by the Python interpreter and are meant for humans to read. In Python, comments start with the # symbol.
- **Keywords**: Keywords are reserved words that have special meaning in Python. Examples include *if*, *else*, *while*, *for*, *def*, and *return*. These cannot be used as variable names.
- **Indentation**: Python uses indentation to define code blocks. Proper indentation is crucial as it affects the flow and logic of your code.
- **Variable Usage and Naming**: Variables are used to store data. Python has rules for naming variables, such as starting with a letter or underscore and avoiding reserved keywords.
- **Traceback Messages**: When an error occurs, Python provides a traceback message that helps you understand what went wrong and where in your code the error occurred.
- **Dynamic Typing**: In Python, you do not need to declare the type of a variable. The type is determined at runtime, which makes Python flexible but requires careful handling of variable types.
- **Built-in Modules**: Python comes with a rich set of built-in modules that provide additional functionality. You can import these modules into your scripts to use their functions and classes.
- **Functions**: Functions are reusable blocks of code that perform a specific task. They help make your code modular and easier to maintain.
- **Constants**: Constants are variables that should not change once they are set. Python does not have built-in support for constants, but you can use naming conventions to indicate that a variable is a constant.
- **Exceptions**: Exceptions are used to handle errors gracefully. Python provides a robust mechanism for catching and handling exceptions using *try*, *except*, and *finally* blocks.

2.3 Unit Objectives - Practical

By the end of this unit, you should be able to:

- **Test Individual Lines of Code Interactively in a Code Editor**: Use an interactive Python shell or a code editor to test and debug individual lines of code.
- **Run Python Scripts in a Code Editor**: Write and execute complete Python scripts within a code editor.
- **Differentiate Between Scripting and Interactive Code Editor Windows**: Understand the differences and use cases for scripting versus interactive coding environments.
- **Pass Input to a Script**: Learn how to pass input parameters to your Python scripts from the command line or within the code.
- **Explain the Advantages of Using an Integrated Development Environment (IDE) Over a General-Purpose Text Editor**: Understand the benefits of using an IDE, such as syntax highlighting, code completion, and debugging tools.
- **Match Code Text Color with Code Components**: Recognize how code editors use color coding to differentiate between various components of the code, such as keywords, variables, and comments.

2.4 Where to Write Code

You can write Python code in any text editor. Save your files with a *.py* extension to indicate that they are Python scripts. Examples of text editors include Notepad, Sublime Text, and Visual Studio Code.

Integrated Development Environment (IDE) and Syntax

An IDE is a software application designed for computer programming. It provides a comprehensive environment that includes a code editor, debugger, and build tools. Syntax refers to the set of rules that define how to write code statements that the computer can interpret.

An IDE can:

- **Check Code Syntax**: Automatically check your code for syntax errors.
- **Highlight Special Code Statements**: Use color coding to highlight keywords, variables, and other code components.
- **Suggest Ways to Complete a Code Statement**: Provide code completion suggestions to help you write code faster and with fewer errors.
- **Provide Special Tools (Debuggers) to Investigate Errors in the Code**: Offer debugging tools to help you find and fix errors in your code.

2.4.1 Python Window

At the end of this section, we will briefly describe Python's IDE, focusing on its features and how it can help you write and debug Python code.

2.4.2 Visual Studio Code (VS Code)

Visual Studio Code is a popular standalone IDE that is widely used for Python development. It is known for its:

- **Cross-Platform Support**: Available on Windows, macOS, and Linux.
- **Lightweight**: Fast and responsive, even on older hardware.
- **Robust Architecture**: Built on a solid foundation that supports a wide range of programming languages and extensions.
- **IntelliSense**: Provides intelligent code completion, parameter info, quick info, and member lists.
- **Freeware**: Available for free, with a large community of users and contributors.

2.4.3 ArcGIS Python

ArcGIS includes a Python console that provides some IDE functionality. It allows you to:

- **Save Code**: Right-click and select "Save as" to save your scripts.
- **Load a Saved Script**: Right-click and select "Load" to open a previously saved script.
- **Run and Test Scripts Outside of ArcGIS Software**: Although it lacks some features of a standalone IDE, it is lightweight and useful for running and testing scripts outside of the main ArcGIS application.

2.4.4 QGIS Python

QGIS also includes a Python console, which serves as an interactive shell for executing Python commands. It features:

- **Python File Editor**: Allows you to edit and save your Python scripts.
- **Interactive Shell**: Provides an environment for running Python commands and testing code snippets.

2.4.5 Python IDE

For more detailed information on using the Python IDE, refer to the official Python documentation and help resources.

2.4.6 ArcGIS Python Window

Example: This Python Script Calls the Buffer (Analysis) Tool

- **simpleBuffer.py**: This script demonstrates how to call the Buffer tool from Python.

The ArcGIS Python Window is embedded in ArcMap. To use it:

1. Browse to *C:/pyScript* and open *simpleBuffer.py*.
2. Copy the last three lines of code from *simpleBuffer.py* into the ArcGIS Python Window.
3. Press the *Enter* key, and you'll see messages indicating that the buffering is occurring.
4. When the process completes, confirm that an output buffer file has been created and added to the map.
5. Confirm that you see a message in the Python Window giving the name of the result file.

You have just called a tool from Python, similar to running it from the ArcToolbox:

- **ArcToolbox > Analysis Tools > Proximity > Buffer**

2.5 Getting Started

Start the Code editor (the one you prefer)

To begin, start the code edior on your computer.

Open the Script

1. Open the script *describe_fc.py*.
2. Identify the several highlighted code components within the script.

The script *describe_fc.py* prints basic information about each feature class in a workspace.

To Try This Example, Follow These Steps:

1. **Start Code editor**: Launch the application.
2. **Open** *describe_fc.py* **in Visual Studio Code**: Navigate to the file and open it in the editor.
3. **You Do Not Need to Run the Code**: Simply examine the code and its components.

The Different Components of the Code

In the script, you will notice that different components of the code are displayed in different colors. These include:

- **Special Symbols (#)**: Used for comments.
- **Indentations**: Used to define code blocks.
- **Dots (.)**: Used to access methods and properties of objects.

2.6 Comments

Comments are information included only for human readers, not for the computer to interpret. Anything that follows one or more hash signs (#) on a line of Python code is a comment and is ignored by the computer.

Uses of Comments
Comments have several important uses:

- **Provide Metadata**: Include information such as the script name, purpose, author, date, usage (input), sample input syntax, and expected output.
- **Outline**: Help the programmer outline the code details and provide an overall workflow for the reader.
- **Clarify Specific Pieces of Code**: Even though good Python code is highly readable, comments can still be helpful to clarify specific parts.
- **Debug**: Assist the programmer in isolating mistakes in the code. Since comments are ignored by the computer, creating 'commented out' sections can help focus attention on another section of the code.

2.7 Keywords

Keywords are words reserved for special usage in Python. The script *describe_fc.py* uses several keywords, such as *import*, *print*, *for*, and *in*.

Here is a list of Python keywords:

```
and, del, from, not, while
as, elif, global, or, with
assert, else, if, pass, yield
break, except, import, print
class, exec, in, raise
continue, finally, is, return
def, for, lambda, try
```

2.8 Indentation

Indentation is meaningful in Python. It is used to define the structure and flow of the code. Notice that lines 19–27 in *describe_fc.py* are indented to the same position. This indicates that these lines are part of the same code block.

2.9 Built-in Functions

The script *describe_fc.py* uses the *print* keyword to print output to the Interactive Window. A function is a named set of code that performs a specific task. A built-in function is a function supplied by Python. You use it by typing the name of the function followed by the input arguments, usually inside parentheses.

2.10 Code Statement

A code statement that uses a built-in function has this general format:

functionName(argument1, argument2, argument3, ...)

Special Terminology Related to Functions

- **Calling Functions**: Executing a function by using its name followed by parentheses.
- **Passing Arguments**: Providing input for the function is referred to as passing arguments into the function. Parameters are the pieces of information that can be specified to a function.
- **Returning Values**: Functions can return values after execution.

Examples:

```
round(2.758, 2) returns 2.76
min(1, 2, 3) returns 1
```

2.11 Variables, Assignment Statements, and Dynamic Typing

Viewing *describe_fc.py* in Visual Studio Code also shows script elements in different colors. All objects in Python have a data type. To understand data types in Python, you need to be familiar with variables, assignment statements, and dynamic typing.

2.11.1 Variables

Variables are a basic building block of computer programming. A programming variable is a name that gets assigned a value.

Example:

```
FID = 145
```

This line of code is called an assignment statement.

2.11.2 An Assignment Statement

An assignment statement is a line of code (or statement) used to set the value of a variable. It consists of:

- The variable name (on the left)
- A value (on the right)
- A single equals sign in the middle.

2.11.3 Print

To print the value of a variable inside a script, you need to use the *print* function. This works in the Interactive Window too, but it is not necessary to use the *print* function in the Interactive Window. When you type a variable name and press the *Enter* key in the Interactive Window, Python prints its value.

Try:

```
inputData = 'trees.shp'
inputData
```

2.11.4 Variable vs Algebra Variable

A programming variable is similar to an algebra variable, except that it can be given non-numeric values. Therefore, the data type of a variable is an important piece of information.

How to Check the Data Type of a Variable

You can check the data type of a variable using the built-in *type* function.

#Examples:

```
FID = 124
type(FID)
inputData = 'trees.shp'
inputData
```

```
type(inputData)
# Try:
avg = 92
type(avg)
avg = 'A'
type(avg)
```

2.11.5 Variable 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. Variable names are case sensitive.

```
Example:
FID = 145
fid
```

Keywords cannot be used as variable names. For example:

```
print = 'inputData'
```

Python ensures that keywords are not used as variable names by reporting an error. However, Python does not report an error if you use the name of a built-in function as a variable name, which can cause unexpected behavior.

Example:

```
type(min)
min(1, 2, 3)
min = 5
min(1, 2, 3)
type(min)
```

2.12 Built-in Constants and Exceptions

Python has built-in constants and exceptions. No built-in names should be used as variable names in scripts to avoid losing their special functionality. Built-in constants include:

- *None*
- *True*
- *False*

Example:

```
type(True)
```

Built-in Constants and Exceptions in Use

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

2.13 Errors

You will often encounter *NameError*, *SyntaxError*, and *TypeError* exceptions.

- **NameError**: Usually occurs because of a spelling or capitalization mistake in the code.
- **SyntaxError**: Can occur for many reasons, but the underlying problem is that one of the rules of properly formed Python has been violated.
- **TypeError**: Occurs when the code attempts to use an operator differently from how it's designed. For example, code that adds an integer to a string generates a *TypeError*.

2.14 Standard (Built-in) Modules

When Python is installed, a library of standard modules is automatically installed. A module is a Python file containing related Python code. Python's standard library covers a wide variety of

functionality, including math functions, file copying, unzipping files, graphical user interfaces, and even Internet protocols for retrieving online data.

To use a module, you first use the *import* keyword.

2.15 Import Statement

The import statement can be applied to one or more modules using the following format:

```
import moduleName1, moduleName2, ...
```

Once a module is imported, its name can be used to invoke its functionality.

```
Example:
import math
math.radians(180)

# Output: 3.141592653589793
Example in ArcGIS:
import arcpy
```

2.16 Demonstration (download and install Python IDE)

Below are the free Python IDEs and online code editors with clear download links:

Free Python IDEs

1. **Visual Studio Code (VS Code)**
   - **Explanation**: A highly customizable code editor with support for Python through extensions. It offers debugging, syntax highlighting, code completion, and Git integration.
   - **Download & Installation**: Available for Windows, macOS, and Linux. Download from the official website: https://code.visualstudio.com/download
   - **Limitations**: Requires extensions for full Python support; can be resource-intensive with many extensions.
2. **PyCharm Community Edition**
   - **Explanation**: A powerful IDE specifically for Python, offering intelligent code completion, code inspections, and integrated debugging.
   - **Download & Installation**: Available for Windows, macOS, and Linux. Download from the JetBrains website: https://www.jetbrains.com/pycharm/download/
   - **Limitations**: Lacks some advanced features available in the Professional Edition; can be slow on older hardware.
3. **Spyder**
   - **Explanation**: An open-source IDE designed for data science. It includes features like an interactive console, variable explorer, and integrated documentation.
   - **Download & Installation**: Available for Windows, macOS, and Linux. Download from the official website: https://www.spyder-ide.org/
   - **Limitations**: Primarily geared towards data science; may not be suitable for general-purpose programming.
4. **Thonny**
   - **Explanation**: A beginner-friendly IDE with a simple interface, making it ideal for learning Python. It includes a debugger and step-through expression evaluation.
   - **Download & Installation**: Available for Windows, macOS, and Linux. Download from the official website: https://thonny.org/
   - **Limitations**: Limited features compared to more advanced IDEs; mainly suited for beginners.
5. **Jupyter Notebook**
   - **Explanation**: An open-source web application that allows you to create and share documents containing live code, equations, visualizations, and narrative text.
   - **Download & Installation**: Available for Windows, macOS, and Linux. Install via Anaconda or pip. Instructions on the Jupyter website: https://jupyter.org/install
   - **Limitations**: Not a full-fledged IDE; better suited for data analysis and machine learning tasks.

2.17 Demonstration - Free Online Python Code Editors

1. **Repl.it**

- **Explanation**: An online IDE that supports multiple programming languages, including Python. It offers collaborative coding, debugging, and hosting.
- **Website**: https://replit.com/
- **Limitations**: Limited offline capabilities; performance can vary based on internet connection.

2. **Google Colab**

- **Explanation**: A free Jupyter notebook environment that runs in the cloud. It's great for data analysis and machine learning.
- **Website**: https://colab.research.google.com/
- **Limitations**: Requires a Google account; limited to the resources provided by Google.

3. **PythonAnywhere**

- **Explanation**: An online Python development and hosting environment. It allows you to run Python scripts and web apps.
- **Website**: https://www.pythonanywhere.com/
- **Limitations**: Free tier has limited resources and features; requires an internet connection.

2.18 ArcPy (ArcGIS)

- **Explanation**: ArcPy allows you to use, automate, and extend desktop GIS functionalities in ArcGIS. It is particularly useful for geoprocessing tasks.
- **Download & Installation**: ArcPy is included with ArcGIS Desktop and ArcGIS Pro. You need to have ArcGIS installed to use ArcPy.
  - **ArcGIS Pro**: Download from the official website: https://www.esri.com/en-us/arcgis/products/arcgis-pro/overview
  - **ArcGIS Desktop**: Download from the official website: https://www.esri.com/en-us/arcgis/products/arcgis-desktop/overview
- **Limitations**: Requires a licensed copy of ArcGIS; not available as a standalone package.

2.19 QGIS with Python (PyQGIS)

- **Explanation**: PyQGIS allows you to automate tasks, create custom plugins, and perform geospatial analysis within QGIS.
- **Download & Installation**: QGIS is available for Windows, macOS, and Linux. Download from the official QGIS website: https://qgis.org/en/site/forusers/download.html
  - **Installation**: Follow the installation instructions on the website. Python is included with QGIS, and you can access the PyQGIS API directly within the QGIS environment.
- **Limitations**: None significant; fully open-source and free to use.

2.20 Practical: Opening existing python code

Use at least two different Software and one online version

2.21 Practical: Writing new python code

Use at least two different Software and one online version

2.22 Group work

2.22.1 Exploring code components

Open an existing python code and explore the components of the code, the objective of the code, the input and output of the code

2.22.2 Correcting an existing code with error

Open an existing code and run it
What error message displayed
Correct the codes

2.23 Demonstration

2.23.1 Simple Exercises

- Print "Hello, World!"
- Calculate the sum of two numbers
- Check if a number is even or odd

2.23.2 Intermediate Exercises

- Find the factorial of a number
- Print the Fibonacci sequence up to n terms

2.23.3 Complex Exercises

- Check if a string is a palindrome
- Sort a list of numbers
- Create a simple calculator

2.23.4 Print "Hello, World!"

```
# Simple Exercises
 Print "Hello, World!"
print("Hello, World!")
```

2.23.5 - Calculate the sum of two numbers

```
num1 = 5
num2 = 3
sum = num1 + num2
print("The sum is:", sum)
```

2.23.6 Check if a number is even or odd

```
num = int(input("Enter a number: "))
if num % 2 == 0:
    print("The number is even.")
else:
    print("The number is odd.")
```

2.23.7 Find the factorial of a number

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

num = int(input("Enter a number: "))
print("The factorial of", num, "is", factorial(num))
```

2.23.8 Print the Fibonacci sequence up to n terms

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        print(a, end=" ")
        a, b = b, a + b

terms = int(input("Enter the number of terms: "))
fibonacci(terms)
```

Complex Exercises- Check if a string is a palindrome

```
def is_palindrome(s):
    return s == s[::-1]

string = input("Enter a string: ")
if is_palindrome(string):
    print("The string is a palindrome.")
else:
    print("The string is not a palindrome.")
```

- Sort a list of numbers

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

numbers = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(numbers)
print("Sorted array is:", numbers)
```

- Create a simple calculator

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y == 0:
        return "Cannot divide by zero!"
    return x / y

print("Select operation:")
print("1. Add")
print("2. Subtract")
print("3. Multiply")
print("4. Divide")

choice = input("Enter choice(1/2/3/4): ")

num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

if choice == '1':
    print("Result:", add(num1, num2))
elif choice == '2':
    print("Result:", subtract(num1, num2))
elif choice == '3':
    print("Result:", multiply(num1, num2))
```

```
elif choice == '4':
    print("Result:", divide(num1, num2))
else:
    print("Invalid input")
```

2.24 Calculate distance between two points

To calculate the distance between two points in a 2D coordinate plane, you can use the distance formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Here's a Python code example to calculate the distance between two points:

```
import math

# Function to calculate distance
def calculate_distance(x1, y1, x2, y2):
    distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
    return distance

# Example coordinates
x1, y1 = 3, 4
x2, y2 = 7, 1

# Calculate and print the distance
distance = calculate_distance(x1, y1, x2, y2)
print("The distance between the points is:", distance)
```

This code defines a function calculate_distance that takes the coordinates of two points and returns the distance between them. You can replace the example coordinates with any other coordinates you need.

If you need to calculate the distance in a 3D space, the formula is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

And here's the Python code for that:

```
# Function to calculate distance in 3D
def calculate_distance_3d(x1, y1, z1, x2, y2, z2):
    distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)
    return distance

# Example coordinates
x1, y1, z1 = 3, 4, 5
x2, y2, z2 = 7, 1, 9

# Calculate and print the distance
distance_3d = calculate_distance_3d(x1, y1, z1, x2, y2, z2)
print("The distance between the points in 3D is:", distance_3d)
```

3 Basic Data Types

Overview

- All Python objects have a data type.
- Built-in Python data types are the building blocks for GIS scripts:
  - Integers
  - Floating point values
  - Strings

This unit uses examples to cover:

- Python numeric data types
- Mathematical operators
- String data types
- String operations and methods

Objectives

By the end of this unit, you will be able to:

- 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.

3.1 Numbers

Numeric Data Types

Python has four numeric data types:

- *int*
- *long*
- *float*
- *complex*

Example:

```
x = 2  # integer
y = 2.0  # float
print(type(x))  # <class 'int'>
print(type(y))  # <class 'float'>
```

Numerical Operators

- **Addition:** *7 + 2* results in *9*
- **Subtraction:** *7 - 2* results in *5*
- **Multiplication:** *7 * 2* results in *14*

Try retaining higher precision:

```
print(8 / 3)  # 2.6666666666666665
print(8.0 / 3)  # 2.6666666666666665
```

3.2 Strings

3.2.1 What Is a String?

- A set of characters surrounded by quotation marks.
- A variable assigned a string literal value is called a string variable.

Example:

```
inputData = 'trees.shp'
print(inputData)  # trees.shp
inputData = "trees.shp"
print(inputData)  # trees.shp
```

3.2.2 String Literals

- String literals in single or double quotes cannot span more than one line.
- A string literal with no closing quote raises a *SyntaxError*.

Example:

```
output = 'a b c'  # SyntaxError if not closed
output = """a b c d e f"""
print(output)  # a b c d e f
```

Line Continuation Character

- A backslash (\) embedded in a string at the end of a line allows a string to be written on more than one line while preserving the single line spacing.

Example:

```
output = 'a b c d e f \
g h i'
print(output)  # a b c d e f g h i
```

Numerical Characters Surrounded by Quotation Marks

- Considered to be string literals by Python.

Example:

```
FID = 145
print(type(FID))  # <class 'int'>
countyNum = '145'
print(type(countyNum))  # <class 'str'>
```

3.2.3 String Operations

GIS Python programming requires frequent string manipulation to deal with file names, field names, etc. You need to be familiar with:

- Finding the length of a string
- Indexing into a string
- Concatenating
- Slicing
- Checking for a substring in a string

3.2.4 Find the Length of Strings

Example:

```
print(len('trees.shp'))  # 9
data = 'trees.shp'
print(len(data))  # 9
```

Indexing into Strings

- Each character in a string has a numbered position called an index.
- Python uses zero-based indexing.

Example:

```
fieldName = 'COVER'
print(fieldName[0])  # C
print(len(fieldName))  # 5
# Attempting to use an invalid index number results in an IndexError
# print(fieldName[5])  # IndexError
print(fieldName[4])  # R
```

3.3 Slicing

3.3.1 Negative Indices

- Useful for getting the last character without checking the string length.

Example:

```
print(fieldName[-1])  # R
# Not possible to change the value of an individual character of a string with indexing
# fieldName[0] = 'D'  # TypeError
fieldName = 'DOVER'
fieldName = fieldName.replace('C', 'D')
```

```
print(fieldName)  # DOVER
```

3.3.2 Slice Strings

- Getting a substring which is only one character long.

Examples:

```
fieldName = 'COVER'
print(fieldName[1:3])  # 'OV'
print(fieldName[:3])   # 'COV'
print(fieldName[1:])   # 'OVER'

inputData = 'trees.shp'
baseName = inputData[:-4]  # Remove the file extension
print(baseName)  # 'trees'
```

- This approach assumes you know the file extension length.

3.3.3 Concatenate Strings

- Concatenation glues together a pair of strings.
- You use the same sign for addition, but it acts differently for strings.
  - The plus sign performs addition on numeric values and concatenation on strings.

Examples:

```
print(5 + 6)  # 11 (adding two numbers together)
print('5' + '6')  # '56' (concatenating two strings)

rasterName = 'NorthEast'
route = 'ATrain'
output = rasterName + route
print(output)  # 'NorthEastATrain'
```

3.3.4 Concatenate Strings with Numbers

- Both of the variables being concatenated must be string types, or you'll get a *TypeError*.

Example:

```
i = 1
rasterName = 'NorthEast'
# output = rasterName + i  # This will raise a TypeError
```

3.3.5 Type Conversion (Casting)

- To combine a string with a numeric value, cast the number to a string.
- Then the number is treated as a string data type for the concatenation operation.

Example:

```
i = 145
rasterName = 'NorthEast'
output = rasterName + str(i)
print(output)  # 'NorthEast145'
```

3.3.6 Use of Concatenation with Slicing

- Create an output name based on an input file name.

Example:

```
inputData = 'trees.shp'
baseName = inputData[:-4]
print(baseName)  # 'trees'

outputData = baseName + '_buffer.shp'
print(outputData)  # 'trees_buffer.shp'
```

3.3.7 Check for Substring Membership

- To check if a string contains a substring.

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

Example:

```
inputData = 'trees.shp'
baseName = inputData[:-4]
print(baseName)  # 'trees'

outputData = baseName + '_buffer.shp'
substring = 'buff'
print(substring in outputData)  # True
print(substring in inputData)  # False
```

Sequence Operations on Strings

Example:

```
exampleString = 'tuzigoot'
print(len(exampleString))  # 8
print(exampleString[2])  # 'z'
print(exampleString[:-4])  # 'tuzi'
print(exampleString + exampleString)  # 'tuzigoottuzigoot'
print('ample' in exampleString)  # False
```

3.4 Uses of string operators

3.4.1 Uses

- These string operations are powerful when combined with batch processing in GIS.
- They can be applied to other data types as well.
- Strings and other data types that have a collection of items are referred to as sequence data types.
- The characters in the string are the individual items in the sequence.

3.4.2 More Things with Strings

In some cases, you may need to:

- Replace special characters in a field name.
- Change a file name to all lower case.
- Check the ending of a file name.

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

String Methods

- Functions associated particularly with strings that perform actions on strings.
- The general format for dot notation looks like this:

4 Object-Oriented Programming (OOP)

4.1 Key Concepts

- **Object** and **method** are fundamental OOP terms.
- **Dot notation** is a specialized OOP syntax.
- Everything in Python is an object.

4.2 More on Objects

- Examples of objects include numbers, strings, functions, constants, and exceptions.
- In Python, most objects have accompanying functions and attributes, referred to as methods and properties.
- As soon as a variable is assigned a value, it becomes an object with methods.

4.3 Methods

- A method is a function that performs some action on an object.
- Methods are simply functions that are referred to as 'methods' because they are performed on an object.
- The terms **calling methods**, **passing arguments**, and **returning values** apply to methods in the same way they apply to functions.

Example: Calling the *replace* Method

- The variable *line* is assigned a string value.
- Dot notation is used to call the string method named *replace*.
- This example returns the string with commas replaced by semicolons:

```
line = '238998,NPS,NERO,Northeast'
print(line.replace(',', ';'))  # '238998;NPS;NERO;Northeast'
```

4.4 Dot Notation

- An object-oriented programming convention.
- Used on string objects (and other types of objects) to access methods and properties designed to work with those objects.

Explanation:

- Dot notation links the object method to the object.
- The object is the variable named *line*.
- The method (*replace*) takes two string arguments: the string to replace (a comma) and the replacement (a semicolon).
- String methods need to be used in assignment statements because strings are immutable. They do not change the value of the string object they are called on; instead, they return the modified value.

Example:

```
line = '238998,NPS,NERO,Northeast'
semicolonLine = line.replace(',', ';')
print(semicolonLine)  # '238998;NPS;NERO;Northeast'
```

Altering the Original Variable

- To alter the original variable (e.g., *line*), use it as the variable being assigned:

```
line = line.replace(',', ';')
print(line)  # '238998;NPS;NERO;Northeast'
```

Remarks

- Not all methods require arguments; however, even when you don't pass any arguments, you must use the parentheses.
- To change the letters of a string to all uppercase letters, use the *upper* method:

```
name = 'Delaware Water Gap'
name = name.upper()
print(name)  # 'DELAWARE WATER GAP'
```

4.5 Split and Join Methods

- The *split* and *join* methods are used in many GIS scripts.
- These methods involve another Python data type called a list.
- The *split* method returns a list of the words in the string, separated by the argument.

Example: *split* **Method**

- The *split* method looks for each occurrence of the forward slash (*/*) in the string object and splits the string at those positions. The resulting list has five items since the string has four slashes:

```
path = 'C:/geodb/data/xy1.txt'
print(path.split('/'))  # ['C:', 'geodb', 'data', 'xy1.txt']
```

- Have inverse functionality.
- The *split* method takes a single string and splits it into a list of strings, based on some delimiter.
- The *join* method takes a list of strings and joins them into a single string.

Example 1: *join* **Method**

```
numList = ['1', '2', '3']
animal = 'elephant'
print(animal.join(numList))  # '1elephant2elephant3'
```

Example 2: *join* **Method**

- The *join* method is performed on a string literal object, semicolon (*;*).
- The method inserts semicolons between the items and returns the resulting string:

```
pathList = ['C:', 'geodb', 'data', 'ch03', 'xy1.txt']
print('/'.join(pathList))  # 'C:/geodb/data/ch03/xy1.txt'
```

4.6 Integrated Development Environment (IDE) and String Methods

4.6.1 Using an IDE

- An IDE makes it easy to browse for methods of an object by bringing up a list of choices when you type the object name followed by a dot.

Example:

```
path = 'c:/geodb/data/river.txt'

print(path.endswith('txt'))  # True
```

Explanation

- When you type *path.* in an IDE, it will show a list of methods available for the *path* object, such as *endswith*, *startswith*, *replace*, etc.
- This feature helps you quickly find and use the methods you need without having to remember all of them.

5 File Paths and Raw Strings

5.1 Dealing with File Paths

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

5.2 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. When a backslash is followed immediately by a character in a string literal, it forms an escape sequence, and the backslash signals that the next character is to be given a special interpretation.

Examples:

- \n represents a new line.
- \t represents a tab.

Example:

print('X\t55.3') # X 55.3

The

- The *strip* method can be used to remove unwanted leading or trailing whitespace from a string. This is often useful when processing files.

Example:

```
dataRecord = '  aaaaa  '
print(dataRecord)  # '  aaaaa  '
dataRecord = dataRecord.strip()
print(dataRecord)  # 'aaaaa'
```

5.3 Escape Sequences in File Paths

- Escape sequences can lead to unintended consequences with file paths that contain backslashes.

Examples:

```
A = 'c:\terrain'
print(A)  # c: errain
dataPath = 'C:\t_river'
print(dataPath)  # C: _river

B = 'c:\nuse'
print(B)  # c:
```

5.4 use

Avoiding Problems with File Paths

1. **Use a forward slash instead of a backward slash:**
2. **Double the backslashes:**
3. **Use raw strings:**

- Placing an *r* just before a string literal creates a raw string. Python uses the raw value of a raw string, disregarding escape sequences.

6 Unicode Strings

```
Introduction to Unicode Strings
```

- When you start using ArcGIS functionality, 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, allowing for the representation of non-English characters, such as the Hindi alphabet.

Example:

```
dataFile = u'counties.shp'
print(dataFile)  # counties.shp
print(type(dataFile))  # <class 'str'>
Difference Between
```

- The difference between *str* and *Unicode* string data types in Python lies in the way the strings are encoded.
- The default encoding for Python *str* strings is based on the American Standard Code for Information Interchange (ASCII), which can only represent hundreds of characters.
- Unicode encoding can represent thousands of characters, making it suitable for non-English languages.
- In your GIS scripts, you can handle *Unicode* strings just like *str* strings.

7 Operations in Python:

7.1 Unicode Strings (Private Reading)

7.2 Using String Methods and Operations on Unicode Variables

Example:

```
dataFile = u'counties.shp'
# Does the string end with '.shp'? -> True
print(dataFile.endswith('.shp'))
# Does the string start with 'co'? -> True
print(dataFile.startswith('co'))
# How many times does 's' occur in the string? -> 2
print(dataFile.count('s'))
```

Output from Methods and Operations

- The output from methods and operations that return strings is Unicode when the input object is Unicode.

Example:

```
dataFile = u'counties.shp'
# Return an all caps string -> 'COUNTIES.SHP'
print(dataFile.upper())
# Index the 6th character in the string -> 'i'
print(dataFile[5])
# Concatenate two strings -> 'counties.shpcounties.shp'
print(dataFile + dataFile)
# 'counties.shp'
print(dataFile)
```

8 Printing Strings and Numbers

Using the

- The built-in *print* function is used frequently in scripting.
- In Python 3.x, parentheses are required around the arguments.

Example:

```
dataFile = 'counties.shp'
FID = 862
print(dataFile, FID)  # 'counties.shp 862'
# 'The first FID in counties.shp is 862 !'
print('The first FID in', dataFile, 'is', FID, '!')
```

8.1 Linking Expressions to be Printed

8.1.1 Using Commas

- Commas can be used to separate multiple expressions in a *print* statement.

Example:

```
dataFile = 'counties.shp'
FID = 862
print(dataFile, FID)  # 'counties.shp 862'
# 'The first FID in counties.shp is 862 !'
print('The first FID in', dataFile, 'is', FID, '!')
```

8.1.2 Using Concatenation

- Concatenation uses a plus sign to join strings. However, it introduces its own complications, such as spacing issues.

Example:

```
dataFile = 'counties.shp'
FID = 862
# This will raise a TypeError because FID is an integer
# print('The first FID in' + dataFile + 'is' + FID + '!')
# 'The first FID in counties.shp is 862!'
print('The first FID in' + dataFile + ' is ' + str(FID) + '!')
```

8.1.3 Using String Formatting

- The *format* method is performed on a string literal object, which contains placeholders for variables and gets the variables as a list of arguments.
- Placeholders are inserted into string literals as numbers within curly brackets (*{0}*, *{1}*, *{2}*, etc.).
- The numbers refer to the zero-based index of the arguments in the order they are listed.

Example:

```
dataFile = 'counties.shp'
FID = 862
# 'The first FID in counties.shp is 862!'
print('The first FID in {0} is {1}!'.format(dataFile, FID))
```

9 Lists

9.1 Objectives

- 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.
- Check script syntax.
- Interpret traceback error messages.

9.2 Introduction to Lists

- A Python list 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:

Example:

- A Python list of field names from a comma-separated value ('.csv') file containing wildfire data:

What is Inside?

- All of the items in the *fields* list are strings, but a list can hold items of varying data types.

Examples:

```
stateData = ['Ethiopia', ['Hawassa', 'Shashemene'], 18809888]
exampleList = [10000, 'a', 1.5, 'b', 'banana', 'c', 'cusp']
# Create an Empty List
dataList = []
```

9.3 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.

Examples:

```
exampleList = [10000, 'a', 1.5, 'b', 'banana', 'c', 'cusp']

# Length
print(len(exampleList))  # 7

# Indexing
print(exampleList[6])  # 'cusp'

# Slicing
print(exampleList[2:4])  # [1.5, 'b']

# Concatenation
print(exampleList + exampleList)  # [10000, 'a', 1.5, 'b', 'banana',
'c', 'cusp', 10000, 'a', 1.5, 'b', 'banana', 'c', 'cusp']
```

```
# Membership
print('prune' in exampleList)  # False
```

Modify List Items

- The operations work on list items just like they work on string characters.

Example:

```
exampleList[0] = 'prune'
# Modifying the first item in the list.
print(exampleList)
# ['prune', 'a', 1.5, 'b', 'banana', 'c', 'cusp']
```

9.4 List Methods

- List objects have a specific set of methods associated with them, including:
  - *append*
  - *extend*
  - *insert*
  - *remove*
  - *pop*
  - *index*
  - *count*
  - *sort*
  - *reverse*

Example:

```
fireIDs = ['238998', '239131', '239135', '239400']
newID = '239413'
fireIDs.append(newID)  # Changing the list in-place.
print(fireIDs)  # ['238998', '239131', '239135', '239400', '239413']
```

Example:

```
fireTypes = [16, 13, 16, 6, 17, 16, 6, 11, 11, 12, 14, 13, 11]
countResults = fireTypes.count(11)
print(countResults)  # 3
```

9.5 The Built-in

Examples:

```
print(list(range(9)))  # [0, 1, 2, 3, 4, 5, 6, 7, 8]
print(list(range(5, 9)))  # [5, 6, 7, 8]
print(list(range(0, 9, 2)))  # [0, 2, 4, 6, 8]
```

9.6 Copying a List

- In-place methods like *reverse* and *sort* alter the order of the list.

Example:

```
fireIDs = ['238998', '239131', '239135', '239400']
fireIDs.reverse()
print(fireIDs)  # ['239400', '239135', '239131', '238998']
```

9.6.1 Shallow Copy

Example:

```
a = list(range(1, 11))
b = a  # Shallow copy
a.reverse()
print(a)  # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
print(b)  # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

9.6.2 Deep Copy

Example:

```
a = list(range(1, 11))
b = list(a)  # Deep copy
a.reverse()
print(a)  # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
print(b)  # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

10 Reading and Writing Files

10.1 Data Sources

- Text and other sources of data can be read and written using file operations in Python.

10.2 Open

- The *open* function returns a file object and is most commonly used with two arguments: *open(filename, mode)*.

Example:

```
f = open('workfile', 'w')
```

10.2.1 File Operations

Using the

- The first argument is a string containing the filename.
- The second argument is another string containing a few characters describing the way in which the file will be used.

File Modes

- *'r'* - Read mode: Opens the file for reading.
- *'w'* - Write mode: Opens the file for writing (an existing file with the same name will be erased).
- *'a'* - Append mode: Opens the file for appending; any data written to the file is automatically added to the end.
- *'r+'* - Read and write mode: Opens the file for both reading and writing.

Optional Mode Argument

- The mode argument is optional; *'r'* will be assumed if it's omitted.

Text Mode

- Normally, files are opened in text mode, meaning you read and write strings from and to the file, which are encoded in a specific encoding.
- When reading, the default is to convert platform-specific line endings (\r\n or \r) to just \n.
- When writing, the default is to convert occurrences of \n to platform-specific line endings.

10.2.2 Using the open

- It is good practice to use the *with* keyword when dealing with file objects.
- The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.
- Using *with* is also much shorter than writing equivalent try-finally blocks.

Example:

```
with open('workfile') as f:
    read_data = f.read()
print(f.closed)  # True
```

10.3 Closing Files

- If you're not using the *with* keyword, you should call *f.close()* to close the file and immediately free up any system resources used by it.

Example:

```
f = open('workfile', 'r')
# Perform file operations
f.close()
```

10.4 Writing to Files

- Warning: Calling *f.write()* without using the *with* keyword or calling *f.close()* might result in the arguments of *f.write()* not being completely written to the disk, even if the program exits successfully.

Methods of File Objects

- Assume that a file object called *f* has already been created.

10.5 Reading Files

- To read a file's contents, call *f.read(size)*, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode).
- If the end of the file has been reached, *f.read()* will return an empty string (''').

Example:

```
f = open('workfile', 'r')
print(f.read())  # Reads the entire file
f.close()
```

10.5.1 Reading Lines

- *f.readline()* reads a single line from the file; a newline character (*n*) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline.

Example:

```
f = open('workfile', 'r')
print(f.readline())  # 'This is the first line of the file.\n'
print(f.readline())  # 'Second line of the file\n'
f.close()
```

10.5.2 Looping Over a File

- For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code.

Example:

```
with open('workfile', 'r') as f:
    for line in f:
        print(line, end='')
```

Reading All Lines

- If you want to read all the lines of a file into a list, you can use *list(f)* or *f.readlines()*.

Example:

```
with open('workfile', 'r') as f:
    lines = f.readlines()
print(lines)
```

Writing to a File

- *f.write(string)* writes the contents of the string to the file, returning the number of characters written.

Example:

```
with open('workfile', 'w') as f:
    f.write('This is a test')
```

11 Decision-Making and Describing Data

11.1 Introduction

- Scripts routinely need to perform different operations based on some deciding criteria.
- The decision may be very simple or it may be more complex.

11.2 This Unit Presents

- The Python syntax for decision-making.
- Conditional expressions.
- ArcGIS tools that make selections.
- The *arcpy* Describe method.
- Handling optional input.
- Creating directories.

11.3 Objectives

- Implement *IF*, *ELSE IF*, and *ELSE* structures in Python.
- Explain when to use only an *IF* block, when to use an *ELSE 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.

11.4 Conditional Constructs

- If some condition is true, some action is taken.
- In Python, conditional constructs begin with the *if* keyword.
- This keyword, followed by a condition, followed by a block of indented code, makes up the simplest conditional construct.

11.5 Syntax:

```
if condition:
    code_statement(s)
```

- If the condition is true, the indented block of code is executed. Otherwise, it is skipped.

Example 1

```
if speciesCount < 500:
    print('Endangered species')
```

Example 2

```
speciesCount = 250
if speciesCount < 500:
    print('Endangered species')
```

- This example has no contingency plan. If the species count is 500 or higher, no special action is taken. It uses only one decision-making code block.

Example 3

- When you want to execute two different actions depending on if the condition is true or false, add an *ELSE* block after the *IF* block.

11.6 Example: Check for valid polygon areas.

```
if area > 0:
    print('The area is valid.')
else:
    print('The area is invalid.')
```

11.7 Boolean Operators

- Python comparison operators (x and y are objects such as numbers or strings):
  - *x < y* # x is less than y
  - *x > y* # x is greater than y
  - *x <= y* # x is less than or equal to y
  - *x >= y* # x is greater than or equal to y
  - *x == y* # x is equal to y
  - *x != y* # x is not equal to y

11.8 Logical Operators

- Python logical operators (x and y are objects, often Boolean expressions):
  - *x and y* # True if both x and y are true.
  - *x or y* # True if either x or y or both are true.
  - *not x* # True if x is false.

11.9 Membership Operators

- Membership operators (x is any object and y is a sequence type object):
  - *x in y* # True if x is in y.
  - *x not in y* # True if x is not in y.

11.10 Comparison Operators

- < and > symbols
- <= and >= operators

11.11 Example:

Print the ID numbers of highways and rivers.

```
if classType == 'major highway':
    print('Highway-', FID)
elif classType == 'river':
    print('River-', FID)
```

Example: Print the ID number for all class types.

```
if classType == 'highway':
    print('Highway-', FID)
elif classType == 'river':
    print('River-', FID)
else:
    print('Other-', FID)
```

11.12 Equality vs. Assignment

- One common mistake is using = when == is intended.
- The single equals sign (=) is an assignment operator, not a comparison operator.
- To set *x* equal to five, use *x = 5*.
- But to compare *x* to five, use *x == 5*.

Logical Operators

- Logical operator keywords *and*, *or*, and *not* are used to encode logical conditions.

Example:

- The following code prints the file name if it has a *.shp* or *.txt* extension:

```
fileName = 'park.shp'
if fileName.endswith('.shp') or fileName.endswith('.txt'):
    print(fileName)
```

- The following code prints file names that do not have a *.csv* extension:

```
if not fileName.endswith('.csv'):
    print(fileName)
```

12 Decision-Making and Looping

12.1.1 Compound Conditional Expressions

- Expressions that use *and* and *or* are called compound conditional expressions because they combine two or more conditions.
- The expressions on either side of the keywords *and* and *or* are evaluated independently. Sometimes this doesn't correspond to how we would normally formulate a condition in natural language.

Example:

- Suppose we want to print the species name only when the species is salmon or tuna.
- A complete comparison expression is needed on both sides:

12.2 Membership Operators

Example 1:

```
if classType == 'major highway' or classType == 'river' or classType
== 'stream' or classType == 'bridge':
    print(classType, '-', FID)
else:
    print('Other-', FID)
```

Example 2:

```
specialTypes = ['highway', 'river', 'stream', 'bridge']
if classType in specialTypes:
    print(classType, '-', FID)
else:
    print('Other-', FID)
```

Example 3:

- Conversely, it can be used to identify only those items that are not in the list:

13 Looping for Geoprocessing

13.1 Unit Objectives

- Implement *WHILE*-loops and *FOR*-loops in Python.
- Identify the three key iterating variable components in a *WHILE*-loop.
- Explain how Python *FOR*-loops work.
- Repair infinite loops.
- Call a geoprocessing tool in a *WHILE*-loop to vary a numerical parameter.
- Automatically generate numerical lists.
- Loop with the *range* function.
- Branch and loop within loops.
- List the files in a directory.
- Geoprocess each file in a list.
- Correct indentation errors.

13.2 Looping Syntax

- Python has two structures for performing repetition: *WHILE*-loops and *FOR*-loops.

Example: Basic

```
dFiles = ['data1.shp', 'data2.shp', 'data3.shp', 'data4.shp']
for currentFile in dFiles:
    print(currentFile)
print('I'm done!')
```

13.3 Example: Use a

Code:

```
# Purpose: Create a set of line features from a set of point
features
import arcpy
arcpy.env.workspace = 'C:/geodb/data/'
```

```
arcpy.env.overwriteOutput = True
theFiles = ['data1.shp', 'data2.shp', 'data3.shp', 'data4.shp']
for currentFile in theFiles:
    # Remove file extension from the current name.
    baseName = currentFile[:-4]
    # Create unique output name. E.g., 'data1Line.shp'.
    outName = baseName + 'Line.shp'
    arcpy.PointsToLine_management(currentFile, outName)
    print('{0} created.'.format(outName))
```

Explanation:

- This script creates line features from point features using the *arcpy.PointsToLine_management* tool.
- It iterates over a list of shapefiles, removes the file extension, creates a new output name, and then processes each file.

13.4 Example:

Code:

```
# Purpose: Buffer a park varying buffer distances from 1 to 5 miles.
import arcpy
arcpy.env.workspace = 'C:/geodb/data/'
arcpy.env.overwriteOutput = True
inName = 'park.shp'
for num in range(1, 6):
    # Set buffer distance based on num ('1 miles', '2 miles', …)
    distance = '{0} miles'.format(num)
    # Set output name ('buffer1.shp', 'buffer2.shp', …)
    outName = 'buffer{0}.shp'.format(num)
    arcpy.Buffer_analysis(inName, outName, distance)
    print('{0} created.'.format(outName))
```

14 Directories in Python:

Directory Inventory and Handling Optional Input

Using

- The *os.path.exists* function returns *True* if it determines that the file passed as an argument exists.

Example:

```
import os
theDir = 'C:/geodb/data/pics'
# os.listdir returns a list of the files
theFiles = os.listdir(theDir)
for fileName in theFiles:
    print(os.path.exists(fileName))
```

Specifying the Full Path

- To specify the full path, join the directory and file name:

```
fullName = os.path.join(theDir, fileName)
print(os.path.exists(fullName))
```

Example: Use

Code:

```
import os
theDir = 'C:/geodb/data/pics'
theFiles = os.listdir(theDir)
for fileName in theFiles:
    fullName = os.path.join(theDir, fileName)
```

```
    print(fullName, os.path.exists(fullName))
```

14.1 Example: Print Modification Time of Files

- This example demonstrates how to use *os.path.join* inside a loop to create full path file names and print their modification times.

Code:

import os

```
theDir = 'C:/geodb/data/pics'
theFiles = os.listdir(theDir)
for fileName in theFiles:
    fullName = os.path.join(theDir, fileName)
    if os.path.exists(fullName):
        modTime = os.path.getmtime(fullName)
        print(f'{fullName} was last modified at {modTime}')
```

Handling Optional User Input

- Sometimes scripts need to handle optional user input. This can be done using default parameter values or checking if an input is provided.

Example:

```
def process_file(file_path=None):
    if file_path is None:
        file_path = 'default.txt'
    with open(file_path, 'r') as file:
        data = file.read()
    print(data)
```

Creating Directories

- Creating directories can be done using the *os.makedirs* function, which creates a directory recursively.

Example:

```
import os
output_dir = 'C:/geodb/output'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
print(f'Directory {output_dir} created.')
```

Safely Creating Output Directories

- It is important to check if a directory exists before creating it to avoid errors.

Example:

```
import os
output_dir = 'C:/geodb/output'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
else:
    print(f'Directory {output_dir} already exists.')
```

End of unit 1