Exercise 6

Exploring spatial data

Exercise data

Exercise data for this book can be downloaded from links.esri.com/PythonPro3rdEditionData. This is a link to the ArcGIS® Online group called *Python Scripting for ArcGIS Pro – 2024 (Esri Press)*. The data for exercise 6 is posted as a zip file called PythonScripting_Ex06_Data.zip. Download this file and extract it to a folder of your choice. The instructions use a folder called C:\PythonPro, but you can use a different folder provided you update any paths.

Check for the existence of data

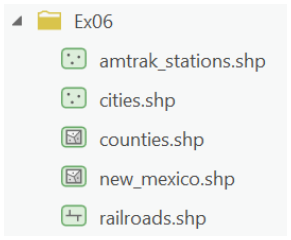
Before starting to work with the exercise data, you will preview the data in ArcGIS® Pro. When starting a new project in ArcGIS Pro, you can leave the home folder, default geodatabase, and default toolbox to their original settings or use the exercise folder—e.g., C:\PythonPro\Ex06. Saving your work is optional.

1. **Start ArcGIS Pro and click Start Without a Template to create a new blank project.**

2. **Make sure the Catalog pane is visible by clicking Catalog Pane on the View tab. Dock the Catalog pane to the right side of the ArcGIS Pro interface.**

3. **Create a new folder connection to the location of the exercise data by right-clicking Folders > Add Folder Connection and navigating to the folder—e.g., C:\PythonPro\Ex06.**

4. **Examine the contents of this folder.**



Notice that there are five shapefiles, including point, polyline, and polygon shapefiles.

5. **Save the project as Exercise 6 to the C:\PythonPro\Ex08 folder. Then, close ArcGIS Pro.**

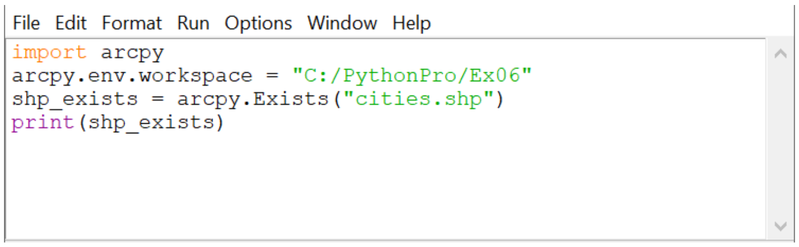
6. **Start IDLE.**

7. **Create a new script file, and save your script as shape_exists.py to the C:\PythonPro\Ex06 folder.**

8. **Enter the following lines of code:**

```
import arcpy

arcpy.env.workspace = "C:/PythonPro/Ex06"

shp_exists = arcpy.Exists("cities.shp")

print(shp_exists)
```

The script file in IDLE looks like the figure.



```
import arcpy
arcpy.env.workspace = "C:/PythonPro/Ex06"
shp_exists = arcpy.Exists("cities.shp")
print(shp_exists)
```

9. **Save and run the script.**

   Running the script returns a value of `True`.

Note: If your result is `False`, double-check the paths, and confirm in ArcGIS Pro that the shapefile of interest resides in the same folder used in the script.

10. **Modify the script by replacing "cities.shp" with "CITIES.SHP".**

11. **Save and run the script.**

Running the script returns the value of `True`. Python, for the most part, is case sensitive and this applies to strings as well. One of the exceptions is path and file names, so "cities.shp" is the same as "CITIES.SHP" and "C:/ PYTHONPRO/ EX06" is the same as "c:/pythonpro/ex06".

Checking for the existence of data is commonly used in stand-alone scripts.

12. **Modify the script as follows:**

```
import arcpy

arcpy.env.workspace = "C:/PythonPro/Ex06"

fc = "cities.shp"

newfc = "cities_copy.shp"

if arcpy.Exists(fc):

    arcpy.CopyFeatures_management(fc, newfc)
```

13. **Save and run the script.**

14. **Start ArcGIS Pro and open Exercise 6.**

15. **In the Catalog pane, confirm that the new feature class has been created, and then delete it.**

The script determines whether the input feature class exists and runs the geoprocessing operation accordingly. If the input feature class does not exist, nothing happens. The script uses variables for the input and output feature classes, which makes the rest of the script more reusable.

16. **Close IDLE.**

Describe the data

The `da.Describe()` function can be used to determine properties of datasets and other inputs into geoprocessing tools.

1. **In ArcGIS Pro, make sure the Catalog pane is open.**
2. **Add the five feature classes in the C:\PythonPro\Ex06 folder to a new map.**
3. **Open the Python window.**
4. **Run the following code:**

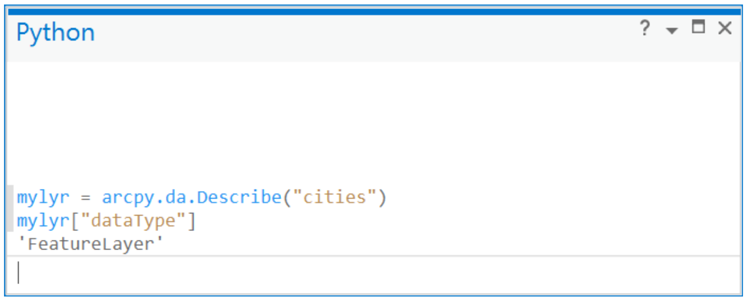
```
mylyr = arcpy.da.Describe("cities")
```

The `da.Describe` function returns a dictionary with items that describe the input dataset.

5. **Run the following code:**

```
mylyr["dataType"]
```

Running the code returns `'FeatureLayer'` as the data type.



```
mylyr = arcpy.da.Describe("cities")
mylyr["dataType"]
'FeatureLayer'
```

The same result can be accomplished with a single line of code:

```
arcpy.da.Describe("cities")["dataType"]
```

However, often you must access more than a single property, so assigning the result from `da.Describe()` to a variable makes it easier to reuse.

You can also work with datasets on disk.

6. **Run the following code:**

```
myshp = arcpy.da.Describe("C:/PythonPro/Ex06/cities.shp")

myshp["dataType"]
```

Running the code returns `'ShapeFile'`. This is the same shapefile but now accessed as a dataset on disk instead of as a feature layer in the active map. You will examine a few more properties in the next steps.

7. **Run the following code:**

```
myshp["datasetType"]
```

The result is `'FeatureClass'`.

8. **Run the following code:**

```
myshp["catalogPath"]
```

The result is `'C:/PythonPro/Ex06/cities.shp'`.

9. **Run the following code:**

```
myshp["file"]
```

The result is `'cities.shp'`.

10. **Run the following code:**

```
myshp["shapeType"]
```

The result is `'Point'`.

Most properties consist of strings or Boolean values, and simply accessing the property prints its value. Some properties, however, consist of objects, and objects can have many properties, which must be accessed separately.

For example, accessing the `spatialReference` property of the feature class returns a `SpatialReference` object.

11. **Run the following code:**

```
myshp["spatialReference"]
```

The result is like the following:

```
<SpatialReference object at 0x5f2a7898[0x4573f8b0]>
```

Notice that the code returns a reference to an object. The reference value will vary with every session, and on its own, this reference value is not so useful. However, the object has many properties, which can each be accessed individually.

12. **Run the following code:**

    ```
    myshp["spatialReference"].name
    ```

    The result is `'GCS_North_American_1983'`.

13. **Run the following code:**

    ```
    myshp["spatialReference"].type
    ```

    The result is `'Geographic'`.

14. **Close ArcGIS Pro.**

    There is no need to save your work.

List data

Describing data typically works with a single element, such as a feature class. List functions can be used to work with many types of elements, including feature classes, rasters, tables, and fields.

1. **Start IDLE.**

2. **Create a new script file, and save your script as list_data.py to the C:\PythonPro\Ex06 folder.**

3. **Enter the following lines of code:**

```
import arcpy

arcpy.env.workspace = "C:/PythonPro/Ex06"

fclist = arcpy.ListFeatureClasses()

print(fclist)
```

Note: The preceding lines of code just as easily could be run using the Python window in ArcGIS Pro. In general, switching between the Python window and a stand-alone editor such as IDLE is a matter of preference. However, writing scripts in an editor is typically preferred as your scripts get longer and when you want to be able to make changes to your scripts in the future.

4. **Save and run the script.**

The list of feature classes is printed to the interactive interpreter, as follows:

```
['amtrak_stations.shp', 'cities.shp', 'counties.shp', 
'new_mexico.shp', 'railroads.shp']
```

Once a list of elements is obtained, a `for` loop can be used to iterate over the elements of the list and carry out a specific task. For example, the `da.Describe()` function can be used to access properties of each of the feature classes in a workspace.

Modify the script as follows:

```
import arcpy
arcpy.env.workspace = "C:/PythonPro/Ex06"
fclist = arcpy.ListFeatureClasses()
for fc in fclist:
    fcdesc = arcpy.da.Describe(fc)
    dtype = fcdesc["dataType"]
    name = fcdesc["name"]
    stype = fcdesc["shapeType"]
    print(f"{dtype} {name} has shapetype {stype}")
```

5. **Save and run the script.**

Running the script prints the date type, name, and shape type of each feature class to the interactive interpreter, as follows:

```
ShapeFile amtrak_stations.shp has shapetype Point
ShapeFile cities.shp has shapetype Point
ShapeFile counties.shp has shapetype Polygon
ShapeFile new_mexico.shp has shapetype Polygon
ShapeFile railroads.shp has shapetype Polyline
```

The same approach can be employed to use the list with geoprocessing tools.

6. **Save your script as list_copy.py to the C:\PythonPro\Ex06 folder.**

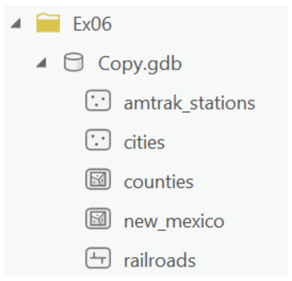
7. **Modify the script as follows:**

```
import arcpy

import os

ws = "C:/PythonPro/Ex06"

fgdb = "Copy.gdb"

arcpy.CreateFileGDB_management(ws, fgdb)

arcpy.env.workspace = ws

fclist = arcpy.ListFeatureClasses()

for fc in fclist:

    fcname = arcpy.da.Describe(fc)["baseName"]

    newfc = os.path.join(ws, fgdb, fcname)

    arcpy.CopyFeatures_management(fc, newfc)
```

8. **Save and run the script.**

9. **Start ArcGIS Pro and open Exercise 6.**

10. **In the Catalog pane, confirm that the new file geodatabase is created and that the five shapefiles are copied as feature classes in the Ex06 folder**



When copying shapefiles to a geodatabase, the file extension .shp must be removed. The script therefore uses the `baseName` property of the feature class rather than the default name for the name of the shapefile. The script uses the `os.path.join()` function to create the correct path, consisting of the workspace, the new file geodatabase, and the name of the feature class.

Note: Some scripts use a statement such as `rstrip(".shp")` to remove the file extension, but this inadvertently can remove additional letters from the name. Using the `baseName` property is therefore more robust.

11. **Close ArcGIS Pro.**

There is no need to save your work.

List functions exist for several different types of elements, including fields. Next, you will create a script for listing the fields of the cities.shp shapefile.

12. **In IDLE, create a new script file, and save your script as list_fields.py to the C:\PythonPro\Ex06 folder.**

13. **Enter the following lines of code:**

```
import arcpy

arcpy.env.overwriteOutput = True

arcpy.env.workspace = "C:/PythonPro/Ex06"

fieldlist = arcpy.ListFields("cities.shp")

for field in fieldlist:

    print(field.name + " " + field.type)
```

14. **Save and run the script.**

The `ListFields()` function returns a list of field objects. The properties of these objects are used to print the name and type of the fields. Running the script prints a list of the names of the fields in the feature followed by their type. The result is as follows:

```
FID OID

Shape Geometry

CITIESX020 Double

FEATURE String

NAME String

POP_RANGE String

POP_2000 Integer
```

```
FIPS55 String

COUNTY String

FIPS String

STATE String

STATE_FIPS String

DISPLAY SmallInteger
```

15. **Close IDLE.**

As you have been working through the exercises, you probably have encountered some unexpected errors. A simple typo can cause a script not to run properly. Even when all typos are fixed, you may continue to run into errors. One of the most common error messages is like the following:

```
ERROR 000725: Output Feature Class C:\<folder>\<file>
already exists.
```

What happens often is that you run a script, and then modify it and try running it again. The script then tries to overwrite existing files that resulted from the earlier execution of the script, and the script fails. To overcome these types of errors, you can add the following line to your script:

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

This line of code makes it possible for the script to overwrite existing files. Even with this statement, however, error messages of this type may continue. If you are running a stand-alone script from IDLE and are also using ArcGIS Pro to examine the results, there may be a shared lock on the data, preventing it from being overwritten. This is a common error. To overcome error messages related to a shared lock on the data, close ArcGIS Pro and run the script again. When the script is finished running, you can open ArcGIS Pro to examine the results.

Manipulate lists

Lists are widely used in batch geoprocessing. Lists can be manipulated in several different ways.

1. **Start ArcGIS Pro and create a new blank project.**

2. **Open the Python Window and run the following code:**

```
arcpy.env.workspace = "C:/PythonPro/Ex06/Copy.gdb"

fclist = arcpy.ListFeatureClasses()

print(fclist)
```

Running the code prints a list of feature classes in the list, as follows:

```
 ['amtrak_stations', 'cities', 'counties', 'new_mexico',
'railroads']
```

Any list in Python can be manipulated using the built-in Python functions and methods. Python lists are indexed starting with the number zero (0). This makes it possible to obtain specific elements in the list or to use slicing functions to create smaller lists that contain just the desired elements. You'll use indexing next.

3. **Run the following code:**

   ```
   fclist[0]
   ```

   The result is `'amtrak_stations'`.

4. **Run the following code:**

   ```
   fclist[3]
   ```

   The result is `'new_mexico'`.

5. **Run the following code:**

   ```
   fclist[-1]
   ```

   The result is `'railroads'`.

6. **Run the following code:**

   ```
   fclist[1:3]
   ```

   The result is `['cities', 'counties']`.

7. **Run the following code:**

   ```
   fclist[2:]
   ```

   The result is `['counties', 'new_mexico', 'railroads']`.

You can also create a list by typing the elements of the list, which you'll do next.

8. **Run the following code:**

```
cities = ["Alameda", "Brazos", "Chimayo", "Dulce"]
```

The number of features can be determined using the `len()` function, which you'll do next.

9. **Run the following code:**

```
len(cities)
```

The result is 4.

The `del` statement removes one or more elements from the list. Because this code does not automatically return the list, a print statement is used to view the current list.

10. **Run the following code:**

```
del cities[2]

print(cities)
```

The result is `['Alameda', 'Brazos', 'Dulce']`.

The `sort()` method can be used to sort the elements in a list, and it can also be reversed. You'll try both methods next.

11. **Run the following code:**

```
cities.sort()

print(cities)
```

The result is `['Alameda', 'Brazos', 'Dulce']`.

12. Run the following code:

```
cities.sort(reverse = True)print(cities)
```

The result is `['Dulce', 'Brazos', 'Alameda']`.

Determining list membership is accomplished using the `in` operator, which you'll use next.

13. Run the following code:

```
"Zuni" in cities
```

The result is `False`.

The `append()` method can be used to add a new element to the end of the list, and the `insert()` method makes it possible to add a new element at a given location, which you'll try next.

14. Run the following code:

```
cities.append("Zuni")

print(cities)
```

The result is `['Alameda', 'Brazos', 'Dulce', 'Zuni']`.

15. **Run the following code:**

```
cities.insert(0,"Espanola")
```

```
print(cities)
```

The result is `['Espanola', 'Alameda', 'Brazos', 'Dulce', 'Zuni']`.

Work with dictionaries

Dictionaries are like lookup tables. They consist of pairs of keys and their corresponding values. Keys are unique within a dictionary although values may not be. Keys must be of an immutable data type, such as strings, numbers, or tuples, but values can be of any type.

1. **Run the following code:**

```
    countylookup = {"Alameda": "Bernalillo County", "Brazos":
"Rio Arriba County", "Chimayo": "Santa Fe County"}
```

This dictionary consists of cities (the keys) and their corresponding counties (the values). Cities as the keys are unique, whereas the counties do not have to be.

Once created, the dictionary can be used as a lookup table, which you'll try next.

2. **Run the following code:**

```
countylookup["Brazos"]
```

The result is `'Rio Arriba County'`.

Notice that using the syntax for dictionaries is like working with elements in a list. The name of the dictionary is followed by brackets [ ], but instead of an index number inside the brackets, one of the keys is used.

The dictionary is designed to work only one way, which you'll see next. You can search a dictionary only by its keys to get the corresponding values. You cannot do the reverse and search for a value to get a key.

3. **Run the following code:**

```
countylookup["Santa Fe County"]
```

The code results in an error, as follows:

```
Traceback (most recent call last):

  File "<string>", line 1, in <module>

KeyError: 'Santa Fe County'
```

In this case, "Santa Fe County" is not one of the keys, and running the code thus returns an error.

Several additional operations can be performed on dictionaries. For example, the number of pairs can be obtained using the `len()` function, as follows.

4. **Run the following code:**

```
len(countylookup)
```

The result is 3.

The pairs in the dictionaries can also be split using the `keys()` and `values()` methods. These methods return a view object, and the `list()` function obtains a list of the element.

5. **Run the following code:**

   ```
   list(countylookup.keys())
   ```

   The result is `['Alameda', 'Brazos', 'Chimayo']`.

6. **Run the following code:**

   ```
   list(countylookup.values())
   ```

   The result is `['Bernalillo County', 'Rio Arriba County', 'Santa Fe County']`.

End of exercise 6.