13

Python and ArcGIS Enterprise

Enterprise GIS represents a progression of GIS from a support role to a decision-making role. By centralizing and organizing geospatial data, and integrating all connected attribute data between all parts of an organization, GIS becomes a central component of organizations, and requires accurate, live, and up-to-date location-based information. Field teams have a live location interface to record work operations. Planners and civil engineers can use map tools to perform land queries and generate mailing lists, using the same accurate address dataset as 911 dispatchers. Finance departments can trust that mail won't be returned because of inaccurate and outdated addresses. Insufficient interoperability between goals and data is the problem, and enterprise GIS is the solution.

As a GIS professional framework, enterprise GIS demands accuracy, precision, and attention to detail. It also requires thoughtful planning and lots of discussion with clients to ensure that the centralized GIS database will serve different needs. Data schemas are devised to minimize management and maximize utility. Data updates are scheduled to maintain the uptime of crucial Web Map services. Data collection by field teams is organized and centralized. Each of these requirements can be supported by the Python modules introduced in this book.

In this chapter, we will cover the following topics:

- Enterprise GIS
- ArcGIS enterprise
- The role of Python
- Publishing service connection and definition files
- Publishing services from MXDs
- A sample ArcPy and ArcREST enterprise GIS workflow

ArcGIS enterprise

Starting with ArcGIS 10.5, ArcGIS Server is now called ArcGIS Enterprise. ArcGIS Enterprise allows for organizational enterprise GIS-- on -site, cloud hosted, or a mix of both. It includes ArcGIS Server, portal for ArcGIS, the ArcGIS Web Adaptor, and ArcGIS DataStore bundled together. Each will be available with ArcPy scripting capabilities.

ArcGIS Enterprise's ArcGIS Server is used to publish map service APIs on the web or on an internal network for use by location-aware software. It allows for a large number of users to access a centralized and versioned geodatabase through Web Maps.

Portal for ArcGIS is an organizational ArcGIS Online-like environment, which is hosted and managed on-site by the organization. It allows for easy Web Map production to allow departments to access dynamic custom Web Maps, which rely on services published by ArcGIS Enterprise.

ArcGIS Web Adaptor is used to protect data source URLs for map services by assigning them a proxy URL address. It also manages large amounts of requests by allowing one address to be shared between multiple servers.

ArcGIS DataStore simplifies data management between multiple on-site and cloud databases and data sources. This includes the data for ArcGIS Online, Portal for ArcGIS, and on-site databases.

All of these components work together in an enterprise GIS environment to make up-to-date location data available to users. Because of the complexity of managing an enterprise GIS environment, Python should be used to automate the ArcGIS Enterprise components. Python scripts that maintain data on a schedule can be converted into operating system scheduled tasks once they are tested and free of bugs. Services can be published and republished automatically. Remote data sources can be queried, processed, and analyzed, or added to the enterprise database.

The role of Python

All of the Python modules introduced in this book are powerful on their own, but the correct combination of each will enable even small GIS shops to improve data management and sharing to an enterprise level. Data analysis, management, and visualization can be successfully automated using a combination of ArcPy, ArcREST, and the ArcGIS API for Python.

ArcPy can be used with ArcGIS Enterprise in many different ways. ArcPy has enterprise database management tools that are used to create geodatabases, feature datasets, and feature classes, and then generate, update, and publish data. ArcPy also has access to ArcToolbox tools, and can be used to create new script tools that chain together both data cursors and ArcToolbox tools.

Data management and data publishing are two major uses of ArcPy in enterprise GIS. For data management, creating, updating, and deleting data within an enterprise database is a daily process. Once the data has been updated, the publishing process starts.

For now, ArcGIS Enterprise uses Python 2.7, though it may be ported to Python 3 in the future.

ArcREST is used to manage data services for ArcGIS Server, ArcGIS Online, and Portal for ArcGIS. Queries and updates to the ArcGIS REST API services are handled by a module specifically built for the task. Data is loaded to, and queried from, the cloud as layers.

ArcGIS API for Python is used to manage the ArcGIS Online layers and ArcGIS Enterprise services. The integration with the powerful Jupyter Notebooks allows for unique analysis capabilities. It will continue to grow in prominence as Esri expands ArcGIS Enterprise, ArcGIS Pro, and ArcGIS Online.

Data management

Enterprise GIS data management using ArcPy uses the cursors and geometry objects described in `Chapter 4`, *ArcPy Geometry Objects and Cursors*. We will use it to create, update, and delete data with scripts that rely on the cursors. For routine data maintenance, scripts are useful as scheduled tasks that can pull data from one source and add or update the data in another database.

The Editor class

The `Editor` class is a special ArcPy data access class, which governs edits for enterprise geodatabases. With start, stop, commit, and rollback functionalities, the `Editor` class is used constantly within enterprise workflows.

The `Editor` class is instantiated (or called) by passing a workspace path to a geodatabase. In this, a geodatabase registered with `Database Connections` in ArcCatalog is passed to the `Editor` class as follows:

```
import arcpy
workspace = r'Database Connections\SanFrancisco.gdb'
editor = arcpy.da.Editor(workspace)
```

The instantiated class can be used for database transactions, with commits and rollbacks available. The actual inserts and updates are handled with the insert cursor and update cursor respectively.

Inserts

Inserting new rows of data into an enterprise database feature class is more complicated than a shapefile or file geodatabase. Instead, the `Editor` class is used to start edit sessions and operations before the insert cursor can be used.

When starting an operation using the `startEditing` method, there is a difference in how the `Editor` class treats versioned versus non-versioned datasets. The two parameters, such as `with_undo` and `multiuser_mode`, need to be set up correctly for the feature class and workspace that is being edited. For versioned data, `with_undo` will allow for versioned data to roll back the database commits. For the `with_undo` parameter, non-versioned data does not support undo or redo.

If the `SanFrancisco` geodatabase was upgraded to an enterprise geodatabase, and used a longitude and latitude CRS, adding a bus stop might look like this:

```
import arcpy
workspace = r'Database Connections\SanFrancisco.gdb'
feature_class = r'Database Connections\SanFrancisco.gdb\
  SanFrancisco\Bus_Stops'
editor = arcpy.da.Editor(workspace)
editor.startEditing(False, True) # parameters: with_undo,
multiuser_mode
editor.startOperation()
with arcpy.da.InsertCursor(feature_class, ["STOPID","SHAPE@XY"]) as
incursor:
    row = [1234321, (-122.123,37.97)]
    incursor.insertRow(row)
    editor.stopOperation()
    editor.stopEditing(True)
```

Only one editing session can be open at one time within a script. While the edit is saved in memory each time an edit operation is stopped using `stopOperation`, the edits are not committed until `stopEditing` is called. You can perform multiple edits within a session, and may consider a start and stop operation with each edit. This can slow down the insert or update process.

Updates and field calculations

A similar operation is used to update data within an enterprise geodatabase. The `Editor` class is again used to start the edit session, and then an `UpdateCursor` method is used to select a row or rows and update values like this:

```
import arcpy
workspace = r'Database Connections\SanFrancisco.gdb'
feature_class = r'Database
Connections\SanFrancisco.gdb\SanFrancisco\Bus_Stops'
editor = arcpy.da.Editor(workspace)
editor.startEditing(False, True)
editor.startOperation()
with arcpy.da.UpdateCursor(feature_class, ["SHAPE@XY"],
  "STOPID=1234321",) as upcursor:
     for row in upcursor:
        row[0] =  (-122.221,37.45)
        upcursor.updateRow(row)
editor.stopOperation()
editor.stopEditing(True)
```

Another useful update feature is field calculation, which can be done using the `Editor` class and the `CalculateField` tool. It does not require a session to start and stop:

```
import arcpy
workspace = r'Database Connections\SanFrancisco.gdb'
feature_class = r'Database Connections\SanFrancisco.gdb
  \SanFrancisco\Census2010'
editor = arcpy.da.Editor(workspace)
arcpy.CalculateField_management(feature_class, 'FieldName',
  "'NewValue'", "PYTHON")
```

For more information on the `Editor` class and its parameters, check out the API reference at `http://desktop.arcgis.com/en/arcmap/latest/analyze/arcpy-data-access/editor.htm`.

Deleting data

Deleting rows is used to update data within an enterprise geodatabase, but it can be permanent, so be careful. The `Editor` class starts the edit session, and then `UpdateCursor` is used to select a row or rows, and delete it. **Be very careful with this, as it can delete data irrevocably**:

```
import arcpy
workspace = r'Database Connections\SanFrancisco.gdb'
feature_class = r'Database Connections\SanFrancisco.gdb
  \SanFrancisco\Bus_Stops'
editor = arcpy.da.Editor(workspace)
editor.startEditing(False, True)
editor.startOperation()
with arcpy.da.UpdateCursor(feature_class, ["SHAPE@XY"],
"STOPID=1234321",) as upcursor:
for row in upcursor:
  upcursor.deleteRow()
editor.stopOperation()
editor.stopEditing(True)
```

As soon as the edits are saved on non-versioned data, the rows cannot be retrieved. On versioned data, it is also important to be sure to check if the deleted rows are retrievable, and if the edits are on the correct version.

Publishing map services

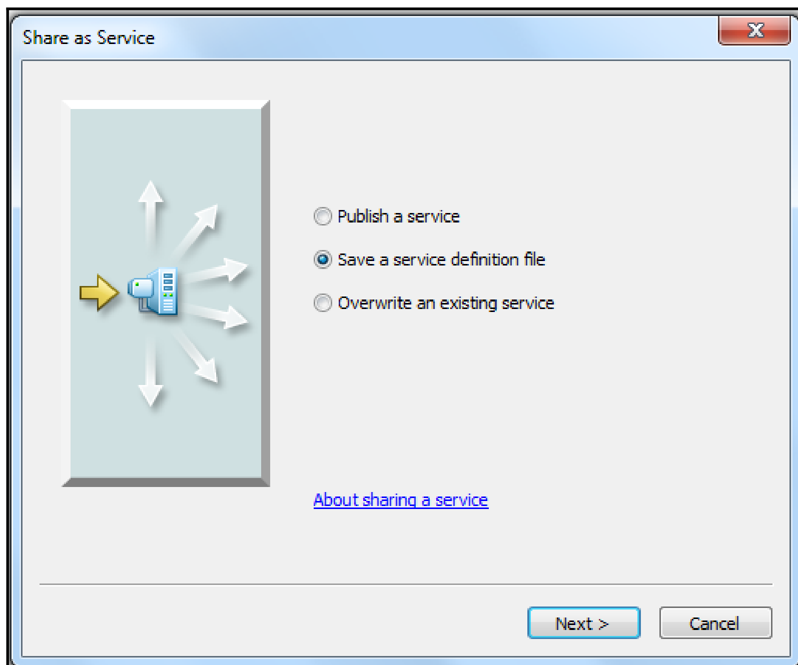
We will generate service definition files in ArcGIS Desktop, and use ArcPy to schedule the publishing of services. This can be used to ensure that data updates take place when most convenient for all users and administrators.

As covered in `Chapter 8`, *Introduction to ArcGIS Online*, data services can be published from MXDs as web services using ArcGIS Online. Similarly, data services can be published using ArcGIS Enterprise, making the data available as a map service to all applications within an enterprise organization. Publishing service definition files and saving them to a folder allows them to be available to ArcPy.

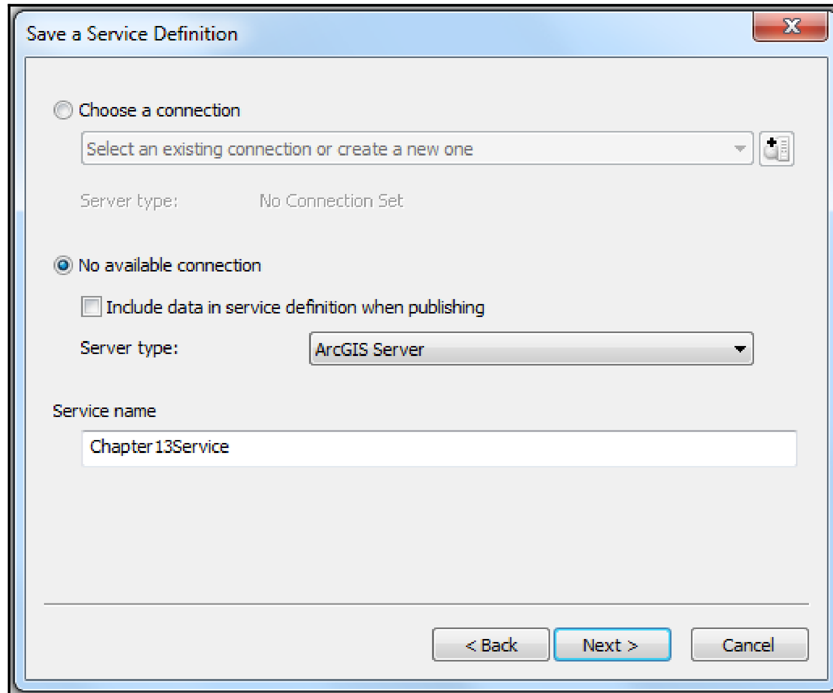
Service definition files

Service definition files are used to schedule the future publishing of a map service. As seen in `Chapter 8`, *Introduction to ArcGIS Online*, there are a number of steps involved with publishing a service from an MXD. Service definition files are used when there is a need to define a service, but publish it later. Let's generate a service definition file from an MXD to use with a script by performing the following few steps:

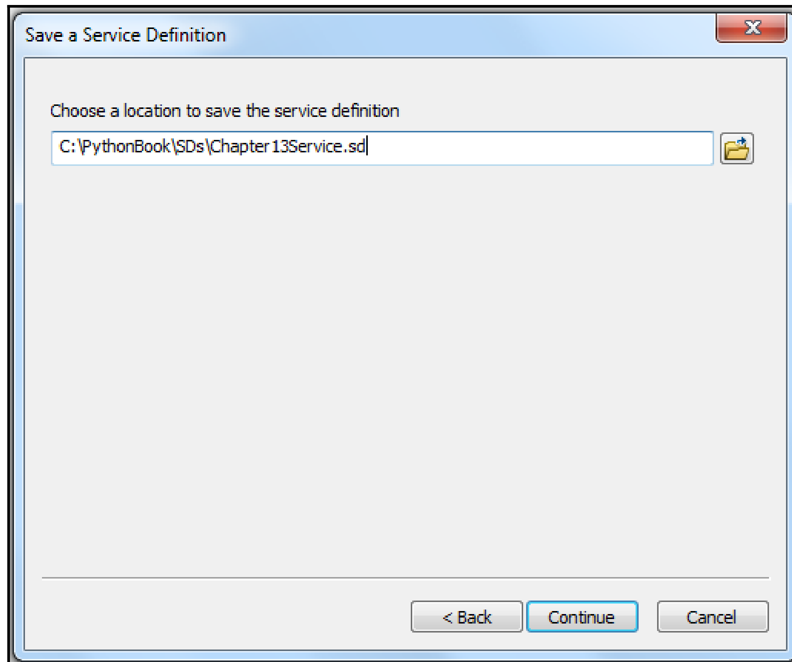
1. Open an MXD with data to be published as a service, and select **Service from the File | Share As menu**. Select the **Save a service definition file** option as shown in the following screenshot:



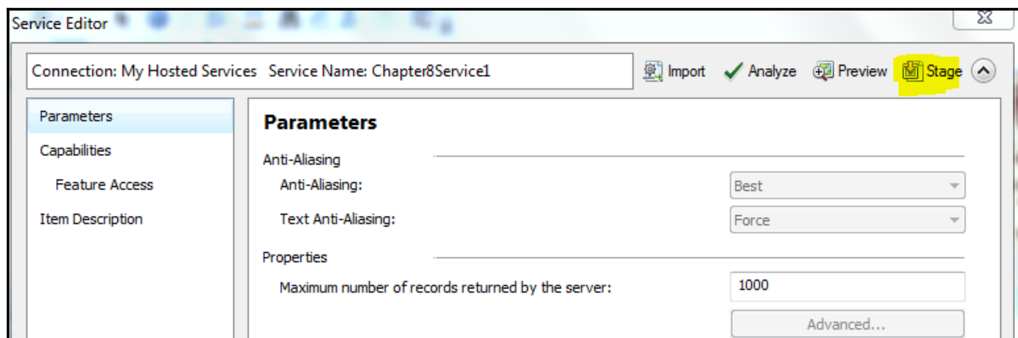
2. Push the **Next** button. If you are creating a service definition with an established server connection, select the server from the list of server connections. If you are creating it with no server connection, select the second radio button and decide if the data will be packaged together with the service definition and the type of server that will publish the service later:



3. Decide on where the service definition file will be published using the folder button. This folder will need to be accessible to the scripts that will publish the services from the service definitions, and to ArcGIS enterprise administrators:



4. After establishing the location of the output service definition file, refer to the steps in `Chapter 8`, *Introduction to ArcGIS Online*, to see how to pick service specifics and manipulate the configuration until satisfied with the error report.
5. Save the service definition file when you are satisfied with the configuration using the **Stage** button in the upper-right corner.



Create the server connection file

ArcPy allows you to create service connection files for servers. These connection files allow the automated publishing of service definition files. The arcpy.CreateGISServerConnectionFile method requires a number of parameters for configuration, and produces a file. These parameters include the level of access to the server and the output folder. For this example, the level of access is administrator (only publisher and administrator accounts can publish services):

```
access = "ADMINISTER_GIS_SERVICES"
folderpath = r"C:\PythonBook\Services"
file_name = "service.ags"
server = "http://{servername}:6080/arcgis/admin"
AGIS_folder = False
staging_folder = folderpath
user = "{username}"
pass = "{password}"
options = [access, folderpath, file_name, server,"ARCGIS_SERVER",
  AGIS_folder, staging_folder, user, pass]
arcpy.mapping.CreateGISServerConnectionFile(*options)
```

Here, {username} and {password} are templates for you to insert your own credentials. For more information on file configuration, check out http://desktop.arcgis.com/en/arcmap/latest/analyze/arcpy-mapping/creategisserverconnectionfile.htm.

Take a look at the use of the asterisk (*) on the options list (with all of the parameters). This shortcut allows the list options to be expanded when passed into a function, passing all of the parameters to waiting variables.

Publishing service definitions

ArcPy can be used to publish a service from the new service definition file using the new service connection. The arcpy.UploadServiceDefinition_server tool is designed to work with published service definition files. As parameters, the tool requires the file path of the service definition file and the server connection to the publishing server:

```
>>> server_conn = r"C:\PythonBook\Services\service.ags"
>>> service_file = r"C:\PythonBook\SDs\Chapter13Service.sd"
>>> arcpy.UploadServiceDefinition_server(service_file, server_conn)
```

Multiple service definition files can be uploaded when they are in a folder. Using the `glob` module, we can get all of the files as follows:

```
>>> server_conn = r"C:\PythonBook\Services\service.ags"
>>> service_folder = r"C:\PythonBook\SDs\*.sd"
>>> import glob
>>> service_files = glob.glob(service_folder)
>>> for service_file in service_files:
  arcpy.UploadServiceDefinition_server(service_file, server_conn)
```

For all of the optional parameters, which allow for configuration overrides and more, check out `http://desktop.arcgis.com/en/arcmap/latest/tools/server-toolbox/upload-service-definition.htm`.

The `glob` module makes it easy to get the full file path of any type of file within a folder. Pass the folder and the asterisk wildcard with the file extension pattern to `glob.glob` to get a list of file paths that match the extension.

Publishing a service from an MXD

Using the `arcpy.mapping` module and ArcToolbox server tools, it is possible to publish services from an MXD. As long as the MXD and its layers are configured correctly, a service can be created and published using ArcPy:

```
import arcpy
map_document = r"C:\PythonBook\MXDs\Service.mxd"
service_draft = r"C:\PythonBook\SDs\service_draft.sddraft"
service = r"C:\PythonBook\SDs\service.sd"
connection = r"C:\PythonBook\Services\service.ags"
description = 'Data service published by user'
tag = 'data'
arcpy.mapping.CreateMapSDDraft(map_document, service_draft,
  service, 'ARCGIS_SERVER', connection, True, None, description,
  tag)
results = arcpy.mapping.AnalyzeForSD(service_draft)
if len(results['errors'].keys())==0:
     arcpy.StageService_server(service_draft, service)
     arcpy.UploadServiceDefinition_server(service, connection)
```

As long as there are no errors returned by the analysis (see the note that follows), the MXD is published to the server specified by the connection variable. The service will contain the data sources and cartography contained within the MXD.

The results returned by the analysis are in a dictionary, which contains subdictionaries. The errors are in a subdictionary, so we can use the dictionary `keys` method to call the error keys list, and use the `len` function to get the length of the list (that is, the number of errors). If the length is zero, then no errors were returned, and the service can be uploaded.

Publishing a map to portal for ArcGIS

Portal for ArcGIS, now included with ArcGIS enterprise, allows for an on-site Web Map "portal", which is very similar to ArcGIS Online. All data is hosted on - site instead of in the cloud, but it also allows for the easy generation of Web Maps for data visualization and collection.

The ArcToolbox server tools contain tools for working with portal for ArcGIS. One very useful tool, the MXD to Web Map tool, allows you to publish to a registered portal. If we were to publish a Web Map of the data from the bus stop analysis in a map document, we could publish it to a portal as follows:

```
import arcpy
map_document = r"C:\Projects\MXDs\Bus_Stops.mxd"
map_title = "BusStops"
user = "{username}"
password = "{password}"
tag = "bus stops"
description = "New map of the Bus Stops for Portal."
arcpy.MXDToWebMap_server(map_document, map_title, description, tag,
  Username=user, Password=password)
```

Read the MXD to Web Map tool help documentation in ArcGIS Desktop or ArcGIS Online to explore the other optional parameters of the tool. The username, password, and active portal information must be set in the ArcGIS administrator, and the user must have the correct permissions for publishing.

ArcREST

ArcREST is written in Python, and uses built-in Python libraries, which can perform web queries, to interact with ArcGIS REST API services. In fact, we can perform an ArcGIS REST API service URL query and process it using only the standard Python library, as shown in the following code snippet:

```
import urllib, urllib2, json
url =   "https://sampleserver6.arcgisonline.com/arcgis/rest/
  services/SF311/FeatureServer/0/query?"
data_parameters = { 'where' : 'district IS NOT NULL', 'f':'json'}
```

```
encode_params = urllib.urlencode(data_parameters)
response = urllib2.urlopen(url, encode_params)
json_data = json.loads(response.read())
```

This preceding code block shows how to pass a URL with parameters to an API that returns JSON data. The built-in library, `urllib`, contains the `urlencode` method to make the parameters URL friendly. The `urllib2` module uses the `urlopen` method to add the encode parameter data to the URL. If the URL and parameters are valid, it passes a response. The response data is sent to the `json.loads` method of the `json` library, where it is converted into a JSON data type, and assigned to the variable `json_data`.

ArcREST FeatureService objects

The ArcREST module is built to interact with the ArcGIS REST API services as well as ArcGIS Online services. The advantage of using a package like ArcREST when dealing with service URLs is that it is written specifically for ArcGIS REST API services. All of the requirements are built into the package, and are available to be called when required. It has classes and class functions for feature services, map services, and layers. The class parameters include optional geometry filters and conditionals, as well as security parameters.

When passed a valid service URL, the `FeatureService` class of the `arcrest.ags.featureservice` module processes the returned JSON data response into a `featureservice` object:

```
import arcrest
url = "https://sampleserver6.arcgisonline.com/arcgis/rest/
  services/SF311/FeatureServer"
fs_object = arcrest.ags.featureservice.FeatureService(url)
```

A `mapservice` object can be created by using the `arcrest.ags.mapservice.MapService` class as follows:

```
import arcrest
url = "https://sampleserver6.arcgisonline.com/arcgis/rest/
  services/USA/MapServer"
ms_object = arcrest.ags.mapservice.MapService(url)
```

Both feature service objects and map service objects have properties and methods. Using these properties and methods, we can access information about the services and its layers. The layers are available as layer objects with their own properties and methods. They are accessed with the `layers` method of the service objects:

```
import arcrest
url = "https://sampleserver6.arcgisonline.com/arcgis/rest/services/
  SF311/FeatureServer"
fs_object = arcrest.ags.featureservice.FeatureService(url)
layers = fs_object.layers
for layer in layers:
    print layer.name, layer.id, layer.url
```

The preceding code snippet would generate the following output:

```
Incidents 0
https://sampleserver6.arcgisonline.com/arcgis/rest/services/SF311/
  FeatureServer/0
```

Conditional queries and geometry filters

ArcREST service objects and layer objects have a `query` method that allows for both conditionals and geometry filters to be applied when querying layer data. This allows us to set data parameters for each data search. The result is JSON data, which must be parsed.

In the script `ArcRESTquery.py`, the `where` keyword is used to set the conditional, and the `geometryFilter` keyword accepts a polygon geometry object. The `returnCountOnly` keyword, when used, calculates the total number of features that meet the query criteria:

```
import arcrest, arcpy
arcpy.env.overwriteOutput = True
url = "https://sampleserver6.arcgisonline.com/arcgis/rest/services/
  SF311/FeatureServer"
fs_object = arcrest.ags.featureservice.FeatureService(url)
layers = fs_object.layers
layer = layers[0]
array = arcpy.Array()
array.extend([arcpy.Point(-122.55,37.70),
arcpy.Point(-122.55,37.81),
arcpy.Point(-122.35,37.81),arcpy.Point(-122.35,37.70)])
sf_extent = arcpy.Polygon(array)
arcpy.CopyFeatures_management(sf_extent,r'C:\Projects\extent.shp')
extent_filter = arcrest.filters.GeometryFilter(sf_extent)
json_data = layer.query(where='district IS NOT NULL',
  geometryFilter=extent_filter, returnCountOnly=True)
print json_data
```

This is the JSON output (the count may change over time). The script also produces an extent shapefile:

```
{u'count': 9535}
```

As query results have a standard limit of 1,000 features, knowing the total number of query results is required when retrieving data from service layers.

Explore the ArcREST module documentation at `http://esri.github.io/ArcREST/` for more information on the available properties and methods of service and layer objects.

Creating a Python module

An important step towards creating reusable code is to package its component functions into a module that can be called from the Python path by any script. To start, we need to create a folder in the `site-packages` folder, where most Python modules are placed when downloaded and extracted using the Python module process or using `pip`. Open up the `site-packages` folder in **Windows Explorer** by navigating to the `C:\Python27\ArcGIS10.5\Lib\site-packages` (or `C:\Python27\Lib\site-packages` if you're using a default Python 2.7 installation) folder.

If you create your own functions to reuse those functions that you find useful, or to share with other users, package the functions together as a module. Modules package together functions in one or more scripts into a folder, which can be shared with others (though they often depend on other modules to run). We have used some of the built-in modules, such as the CSV module, and third-party modules, such as ArcPy. Let's explore their construction to get a feel of how a module is packaged for use and sharing. Open the `Chapter13` scripts folder, and look in the `functions` folder. There is a special file inside the folder, which allows the code inside the folder to be imported as a module, just like CSV or any other Python module.

The __init__.py file

Within the module folder (in this case, the folder `functions`), a special file has been added to let Python recognize the folder as a module. This file, called `__init__.py`, takes advantage of the special property of Python called "magic" objects or attributes, which are built into Python. These "magic" objects use the leading and trailing **double underscores** to avoid any confusion with custom functions.

These are double underscores. The single underscores are usually used for so-called private functions within custom Python classes.

The `__init__.py` file is used to indicate that the folder is a module, which makes it importable using the `import` keyword. The file can hold Python commands such as import statements that initiate the module by calling any modules that it may rely on. However, there is no requirement to add import commands to the `__init__.py` file; it can be an empty file, and will still perform the module recognition functionality that we require as follows:

1. Create a folder called `common` in the `Lib\site-packages` folder in your Python27 installation (for example, `C:\Python27\ArcGIS10.5\Lib\site-packages`).
2. Open up IDLE or your favorite IDE, and in the folder called `common`, add a new Python file and call it `__init__.py`. This file will remain empty for now.
3. Now that we have initiated the module, we need to create a script that will hold our common functions. Let's call it `useful.py`, because these functions will be the most useful for this analysis and others. Save it into the `common` folder.
4. The next step is to transfer functions that we created in earlier chapters. These valuable functions are locked into those scripts, so by adding them to `useful.py`, we will make them available to all other scripts that we craft. Add the `createCSV` and `createXLS` functions from `Chapter 2`, *Creating the First Python Script*, `useful.py`, and any other functions that you have written and rely on.
5. Now that the simple module is organized, the functions can be imported into a script as follows:

```
from common.useful import createCSV, createXLS
```

Some modules, such as `arcpy`, are not placed within the `Lib\site-packages` folder. They require the Python path to be modified to make them "importable". Placing modules within the site-packages folder eliminates this requirement.

A sample workflow

Imagining a sample workflow that relates to all readers is a hard thing to do. However, the tools that will work for most organizations solve similar problems, such as getting data, cleaning and analyzing data, and publishing data as either a printable map, a Web Map, or both. In this example, we will use the web to get data, process it, analyze it, add it to a dataset, and then publish it as a map service.

To understand complaints within a specific distance of a bus stop, we will create a workflow that accesses San Francisco 311 complaint data reported by users to an online data source. We will then use ArcREST to download the data, and process it into a JSON format. We will analyze the data location, locate complaints within a distance of each bus stop, and build a rudimentary complaint score statistic. We will then update the value of a field called `complaintScore` within each row in a bus stop feature class located in an enterprise geodatabase version of `SanFrancisco.gdb`.

Functions in the workflow

To best organize this workflow, we'll use a set of functions designed to collect, process, and update data. These functions are designed to show you how to divide up each step of a workflow into its component logic. They are saved in the folder called `Chapter13`. They are organized in a script called `dataquery.py`, which is inside the `functions` folder along with an `__init__.py` file. The `functions` folder is now a module that is imported to the `sampleworkflow.py` script.

The first thing required is a data source. With access to an ArcGIS REST API web service, we can query the data source for the data required. We'll use the Esri `SF311` layer in this example. It is a feature service, so it will require the use of the ArcREST feature service class:

```
import arcpy, arcrest
url="https://sampleserver6.arcgisonline.com/arcgis/rest/services/
SF311/FeatureServer"
```

This workflow requires a function similar to that of the geometry of the bus stops (a `PointGeometry` object). The default fields returned by this function are the shape field and the object ID, which require a path to a feature class or shapefile. However, the function defaults allow for any data to be retrieved from the feature class. A dictionary is generated using a dictionary comprehension (an advanced Python method that iterates through a list to create a dictionary). The keys are the field names, and the data values are lists generated by the `list()` function:

```
def getData(featurepath,fields=['SHAPE@', 'OID@'], sql='',srid=None):
      'retrieve specific fields and return a dictionary of their
        values'
      data_dic = {field:list() for field in fields}
      #dictionary comprehension
      with arcpy.da.SearchCursor(featurepath,fields,sql,srid) as
        cursor:
        for row in cursor:
            for counter,field in enumerate(fields):
            data_dic[field].append(row[counter])
      return data_dic
```

The next thing required is a function that accepts the geometry of a bus stop, and buffers it. Each geometry is buffered using the radius provided by the `distance` variable. A list of buffer polygons is returned:

```
def bufferGeometry(geometries,distance=400):
    'create a buffer around a geometry at a radius of {distance}'
    buffer_geometries = []
    for geometry in geometries:
        buffer_geom = geometry.buffer(distance)
        buffer_geometries.append(buffer_geom)
    return buffer_geometries
```

This following function will take a list of geometries and transform them into another coordinate system. This is useful when generating geometry filters for queries. It uses the `projectAs` method of the geometry objects:

```
def transformGeometry(geometries,srid=None):
    'project geometries into different spatial reference systems'
    transform_geometries = []
    if srid != None:
        srid = arcpy.SpatialReference(srid)
    else:
        srid = arcpy.SpatialReference(4326)
    for geometry in geometries:
        projected = geometry.projectAs(srid)
        transform_geometries.append(projected)
```

```
        return  transform_geometries
```

The next thing required is a function that creates a geometry filter from an extent using string manipulation. As the geometry filter requires a `geometry` object, such as a polygon, this function creates one using the extent of the buffer:

```
def extentGeometry(geometries):
    'create extent geometry filters'
    extent_geometries = []
    for geometry in geometries:
        extent = geometry.extent
        array = arcpy.Array()
        extent_vertices = [arcpy.Point(extent.XMin, extent.YMin),
          arcpy.Point(extent.XMin, extent.YMax),
          arcpy.Point(extent.XMax, extent.YMax),
          arcpy.Point(extent.XMax, extent.YMin)]
        array.extend(extent_vertices)
    extent_geometry = arcpy.Polygon(array,geometry.spatialReference)
    extent_filter = arcrest.filters.GeometryFilter(extent_geometry)
    extent_geometries.append(extent_filter)
    return extent_geometries
```

We need a function to communicate with the service and query the layer by extent and/or by the conditional. This function performs the query and returns the JSON response data as a Python dictionary:

```
def queryLayer(url, layerid, conditional="1=1",geom=None):
  'query a layer using a feature service URL'
    fs_object = arcrest.ags.featureservice.FeatureService(url)
    layers = fs_object.layers layer = layers[layerid]
    json_data = layer.query(where=conditional, geometryFilter=geom)
    return json_data
```

With data sources and filters set, the next thing required is a function to create a complaint score. There are many ways to calculate such a score, but this will just be a simple count of complaints in the area. Using the `len` function, we can get a count of the number of rows returned. The rows are organized using the key `features` within the JSON response data:

```
def complaintScore(featureset):
  'create a tally based complaint score'
  complaint_score = len(featureset.features)
  return complaint_score
```

Finally, updating a table in the geodatabase with the processed complaint score data will require a function. This will accept a processed data dictionary that contains three values for each key, such as `sql`, where the conditional limits the updates to one specific row, a list of `fields` with the columns that will be updated in the row, and a list of `data` items to update for those fields. The `Editor` class is used to start, stop, and record the enterprise database transaction:

```
def updateEnterpriseFeature(data_dict, workspace, feature_class,
    srid=None,with_undo=False,multi=True):
    'update specific rows and fields in an enterprise feature
    class'
    editor = arcpy.da.Editor(workspace)
    editor.startEditing(with_undo, multi)
    editor.startOperation()
    for datakey in data_dict.keys():
        sql, data, fields = data_dict[datakey]
        with arcpy.da.UpdateCursor(feature_class, fields,sql,srid) as
          upcursor:
            for row in upcursor:
                row= data
                upcursor.updateRow(row)
    editor.stopOperation()
    editor.stopEditing(True)
```

All of these functions are saved in the script called `dataquery.py` inside the folder called `functions`. Because there is an `__init__.py` file in the folder as well, and the script `sampleworkflow.py` is next to the `functions` folder, the functions in `dataquery.py` can be imported into a script to be used to request, process, analyze, and update data. This workflow or a version of it could be used as a scheduled task, which updates a dataset with new values hourly, daily, or weekly.

By calling the `dataquery.py` functions in a script, we can run a complete workflow. An example of this is the `sampleworkflow.py` script. When importing modules, the script will look locally (that is, in the same folder) first and then to the PYTHONPATH to find the `functions.dataquery` module:

```
import arcpy, arcrest
from functions.dataquery import
getData,bufferGeometry,transformGeometry
from functions.dataquery import
extentGeometry,queryLayer,complaintScore,updateEnterpriseFeature
url = "https://sampleserver6.arcgisonline.com/arcgis/rest/services
  /SF311/FeatureServer/"
layerid = 0
workspace = r"Database Connections\SanFranciscoDB"
```

```
feature_class = r"Database Connections\SanFranciscoDB\
  SanFrancisco\Bus_Stops"
bus_stop_fields = ["SHAPE@", "OID@", "STOPID"]
sql = "NAME = '71 IB' AND BUS_SIGNAG = 'Ferry Plaza'"
bus_data_dic = getData(feature_class,bus_stop_fields, sql)
bus_data_geometries = bus_data_dic['SHAPE@']
bus_data_buffers = bufferGeometry(bus_data_geometries)
bus_data_transforms = transformGeometry(bus_data_buffers)
bus_data_extents = extentGeometry(bus_data_transforms)
datascores = []
for extent in bus_data_extents:
    query_data = queryLayer(url,layerid,geom=extent)
    score = complaintScore(query_data)
    datascores.append(score)
    data_dic = {}
for counter,stop in enumerate(bus_data_dic['STOPID']):
    sql = "STOPID = {0}".format(stop)
    fields = ["complaintScore"]
    data = [datascores[counter]]
    data_dic[stop] = sql, data, fields
updateEnterpriseFeature(data_dic, workspace,feature_class)
print "update process complete"
```

This sample workflow has been recorded in the `Chapter13` script folder as `sampleworkflow.py`. In the script, the required libraries (including our custom module called `functions`) are imported, and then the variables are defined. This includes the URL of the service we will query (the `SF311` data service) along with the workspace and file path to an enterprise GIS database version of the `SanFrancisco.gdb` file and the bus stops feature class. It also includes the fields to be passed to the `getData` function.

A second version of the script has been saved as `sampleworkflow2.py`. This script allows you to test out the tools without needing an enterprise geodatabase. It imports an update function that does not require an `Editor` class. If you use it, you can add a new field to the bus stops feature class to add the calculated data, or to overwrite data in an existing field.

Variables are passed to the `getData` function, where the data returned is organized into a dictionary with field name keys and ordered lists of data as values. The geometries list is retrieved using the `SHAPE@` key, and is passed to `bufferGeometry` to get buffer polygons using the function's default 400-foot distance. The polygons are then transformed from projected coordinates to geographic (latitude/longitude) coordinates using the function `transformGeometry`. These transformed buffer polygons are passed to the `extentGeometry` function, which requests the extent property of each geometry. The returned `Extent` object is passed to a string template to get an ArcREST layer query geometry filter in the correct format.

These geometry filter strings are then iterated, with each string passed to the `queryLayer` function. The returned JSON data is passed to the `complaintScore` function. This function counts the number of complaints that are located within the extent area around the buffer geometry of the bus stop. This score is added to the list called `datascores`.

Iterating through the `STOPID` list in `bus_stops_dic` allows us to create the formatted dictionary for the `updateFeatureClass` function. The stop IDs are passed to an SQL conditional statement template. The fields to be updated for that stop ID are assigned. The data (in this case, the new complaint score) is extracted from the `datascores` list using indexing. The stop ID is then used as the key in the dictionary called `data_dic`.

Once the dictionary is formatted, it is passed along with the `workspace` and `feature_class` variables to the `updateFeatureClass` function. This will add the new complaint score value to an existing field called `complaintScore`, which must exist in the feature class data schema.

Summary

Enterprise GIS is becoming a standard for smart organizations and cities. GIS will continue to climb in importance as it achieves greater integration within organizations. This will increase the reliance on automation within GIS departments, and enterprise GIS professionals will need to embrace programming to achieve organizational goals for data quality.

This book is an introduction to using Python to automate GIS analysis, data management, data processing, and data publishing. As GIS grows and changes, it will require greater knowledge of Python automation to handle the big datasets (most of which involve location!) produced in the modern world. ArcGIS enterprise will power most enterprise GIS structures in the world, and Esri has embraced the use of Python for automating all parts of its GIS software ecosystem.

We hope you have enjoyed our review of the useful Python libraries created by Esri to power their ArcGIS software. If you continue to practice these lessons and apply them to your work daily, you will become comfortable using these libraries.

Thank you for reading our book!