Calculating Intersections for Large Datasets

# **Introduction**

## Our Goal

Our project involves calculating the Annual Average Daily Traffic on roadways in Greenville. We have a set of 1.9 million traffic routes, and a set of points that represent existing traffic data collection sites through Greenville.



As the next step of our project, we want to calculate which data collection sites lie on each route, so that we know how to use their data. ArcGIS provides an Intersect tool that calculates the intersection between polylines (like our traffic routes) and points (like our traffic data collection sites) – perfect for our needs.

## The Challenge

This is a simple task for small datasets. However, like many GIS computing jobs, the time and computing power it takes to find intersection points scales exponentially with the size of your dataset – and this dataset contains almost two million polylines. This could take days to process on a single machine, and that’s a long time to wait for your results, especially when you have to leave a university computer logged in for all that time. As a matter of fact, in many cases, a single computer will actually run out of memory and crash when trying to process an entire dataset by itself. Therefore, we need to use a type of divide-and-conquer system.

## Parallelizing Your Work

So how do we go about this? Well, it’s actually not that hard! Instead of running a single process to find the intersection of every route with every point, we can split the routes into hundreds of small subsets, each containing a handful of routes, and then calculate the intersections with the points representing the traffic data collection sites. There are relatively few points (only 258), so we don’t need to split this feature class – we can compare each subset of routes with the entire set of points.

Let’s break this process down into three tasks:

1. **Splitting Your Datasets**

Basically, you want to use a Python script to split your routes dataset into a large number of subsets. We’ll call this script split.py. For many types of GIS jobs (such as Intersect), you will split some or all of your routes dataset into blocks, based on their object ID or their position in a grid. ArcGIS, through its Python interface ArcPy, provides this functionality. In the next section, we’ll talk about what exactly Python is and how to use it to accomplish this task.

1. **Running Your Intersect Jobs**

After your dataset is broken up into pieces, each piece can be treated as a separate job for ArcGIS. We could write a script to process each piece sequentially on one computer, but we have a useful trick up our sleeve: HTCondor, or Condor for short. Condor is a scheduler that distributes computing tasks across a network of workstations. Since we have dozens of workstations at our disposal, we will use Condor to send each machine a piece of our project and run an ArcGIS script on each piece.

The basics of the distribution process: You will use a Condor submit script (a .sub file) that sends each piece to a separate machine for processing, along with another Python script, intersect.py, that tells the target machine what to do with the piece it receives. This Python script is the one that will actually call the ArcGIS function (Intersect\_analysis, in this case), again using ArcPy. A function is a piece of code that runs a procedure or routine that either returns a value or preforms a task.

The benefit of using Condor is that instead of using your one computer to process all that data, each computer in the network can process a small part and then send it back to you. All of these computers can process data at same time, or in *parallel*. All your individual jobs together are called a *cluster*. So the benefit is that Condor allows you to parallelize your work using cluster computing.

1. **Combining Your Results**

When a remote computer finishes with its individual job, it will send its results back to your PC. When all of the jobs in your cluster have finished, it is time to merge the results back together into one final dataset. Each individual piece has a result feature class (in this case, the result feature class contains points of intersection for that set of routes) that can be combined into one feature class. We can accomplish this task using one more Python scripts, which we’ll call merge.py.

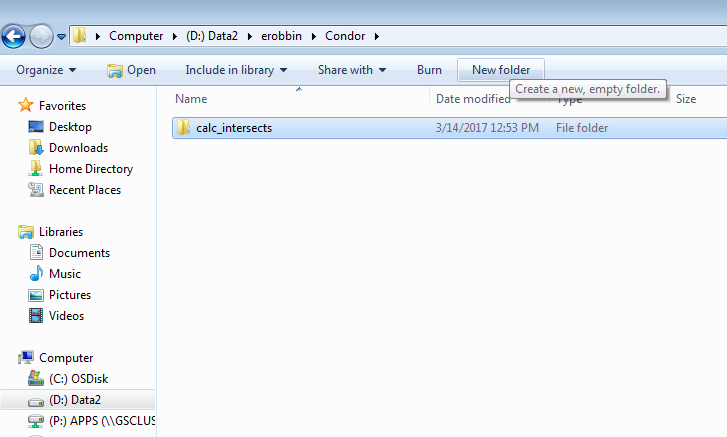
# **Getting scripts from Python in ArcGIS**

You know how to perform a lot of tasks inside ArcMap, using a GUI (graphical user interface). This is all the point-and-click tools you see. However, you may not know that everything you do in ArcMap can be performed using a Python script, either inside or outside of the ArcGIS software.

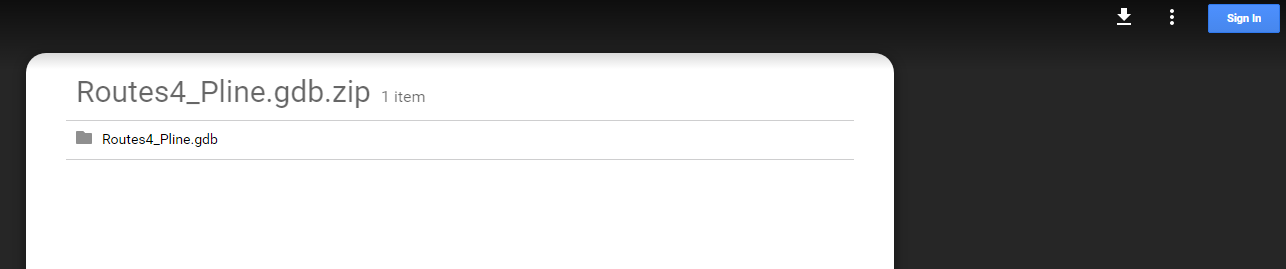
ArcGIS provides a Python interface to its functions called ArcPy, which allows us to run GIS functions by typing lines of code. For example, let’s see how to do this Intersect task both using the GUI and using Python. Follow along, but do not actually run the intersect at the end. The data set is too large and must be parallelized using Condor in order to run and not take forever.

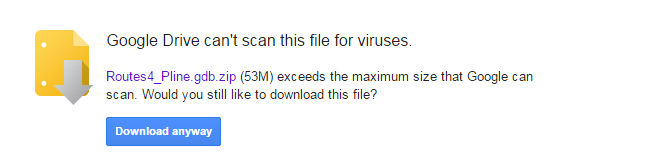
### **Getting the data you need**

* 1. Create a working folder somewhere on the D: drive. You may call this folder whatever you like. I chose to call it calc\_intersects.

****

* 1. Next, you need to get the dataset you want to calculate intersections for. This geodatabase is hosted on Google Drive. Open a web browser and navigate to the following link: <https://drive.google.com/file/d/0B68smydYt7UwUXUtdFlBMGhfTTQ>
  2. You will see the download screen.Click the downward arrow in the top-right of the screen to download the geodatabase, Routes4\_Pline.gdb. If you get a message informing you that the file is too big to scan for viruses, click “Download anyway.”

****

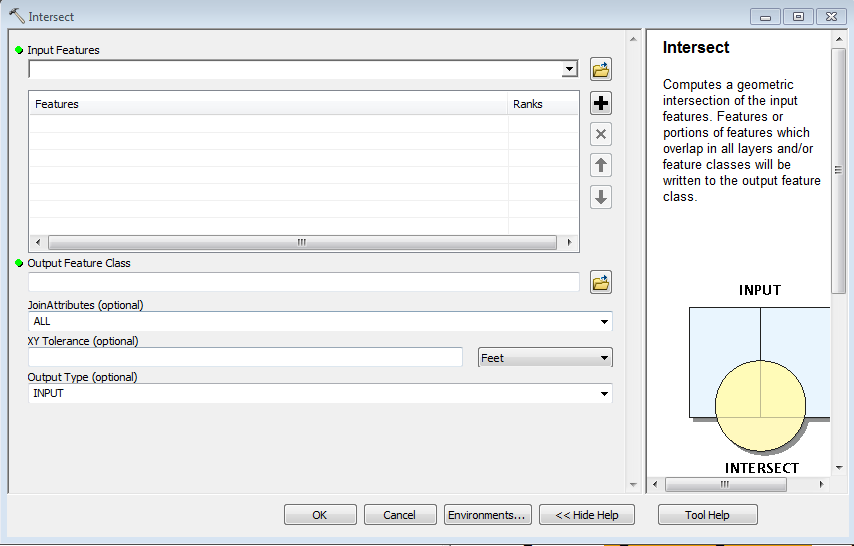
****

* 1. Navigate to your Downloads folder and unzip the downloaded geodatabase by right-clicking the zip file, selecting Extract all in the menu, and pressing Enter. In the window that pops up, copy the Routes4\_Pline.gdb directory to your working folder.
  2. Open ArcMap and create a Folder Connection to your working folder.

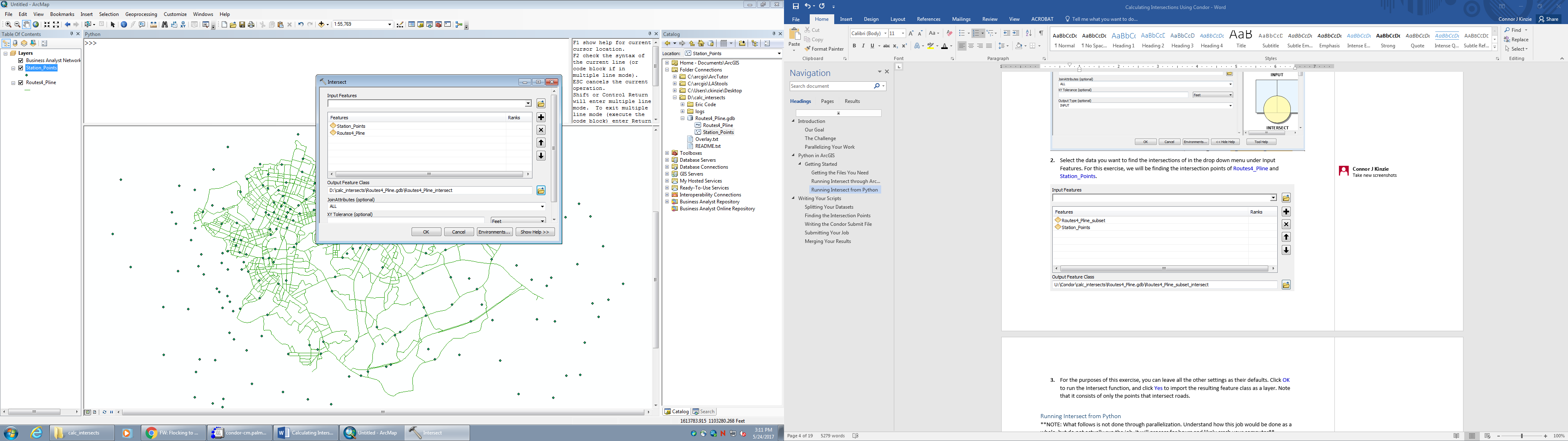
### **Running Intersect through ArcMap**

**\*\*NOTE: What follows is not done through parallelization. Understand how this job would be done as a whole, but do not actually run the job. It will process for hours and likely crash your computer\*\***

1. Select Intersect from the Geoprocessing menu at the top of the screen. This opens the Intersect tool window.

****

1. Select the data you want to find the intersections of in the drop down menu under Input Features. For this exercise, we will be finding the intersection points of Routes4\_Pline and Station\_Points.



1. For the purposes of this exercise, you can leave all the other settings as their defaults. You would click OK to run the Intersect function, and then click Yes to import the resulting feature class as a layer. Note that it consists of only the points that intersect roads.

### **Running Intersect from Python**

**\*\*NOTE: What follows is not done through parallelization. Understand how this job would be done as a whole, but do not actually run the job. It will process for hours and likely crash your computer\*\***

The exact same task can be accomplished almost as simply using Python, once you have a little expertise. In fact, we can run Python directly inside ArcMap for convenience.

1. Open the Python window by clicking on the Python icon on the Standard toolbar. Drag the window that appears to the blue arrow at the bottom of the screen to dock the Python scripting shell.

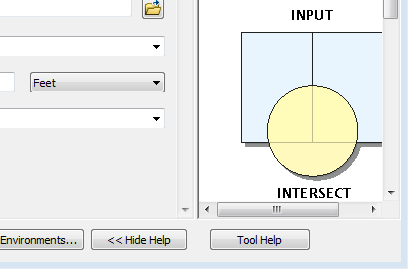
****

1. The one step we need to add is to tell the Python interpreter which geodatabase we are currently working in. To do this, we will need the full path of the Routes4\_Pline.gdb that we plan to use. Right-click Routes4\_Pline.gdb and select Properties at the end of the menu. The path in the Name box should already be highlighted, so right click it and select Copy from the menu.
2. Now we’re ready to write our first line of Python code. We want to set the workspace variable of the environment parameter of ArcPy. Click on the Python shell, then type:

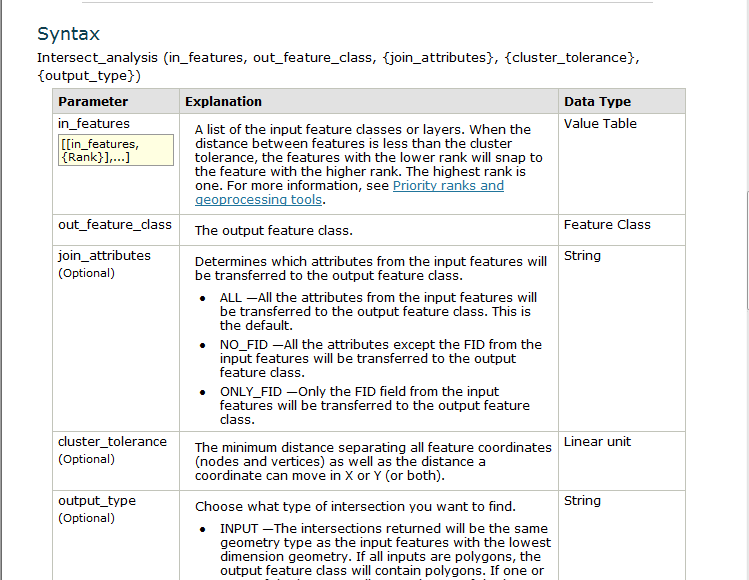
**arcpy.env.workspace = “D:\calc\_intersects\Routes4\_Pline.gdb”**

Instead of typing the full path of the geodatabase, you can right-click and Paste the path you copied in the last step. However, you must encase it within quotation marks, just like in the screenshot.

1. Now ArcPy knows which geodatabase to work with. It’s time to run the intersect function itself, but how do we know what syntax to use? Fortunately, ArcGIS includes detailed reference materials easily accessible through ArcMap. Open the Intersect tool from the Geoprocessing menu, just as you did in the last section. In the bottom right, click the button labeled Tool Help. A help window will appear.

****

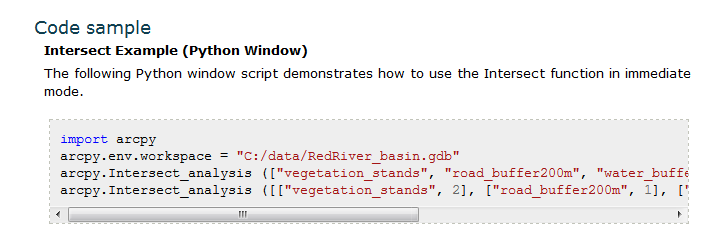
1. This screen contains a detailed explanation of the Intersect tool’s parameters and behavior. It also includes the Python function used to call this tool through ArcPy. Scroll down a few lines to the Syntax section.

****

1. In Python, a function is called using the following form:

***function\_name (argument\_1, argument\_2, …)***

Immediately below the section header Syntax, the Python function for the Intersect tool is listed, followed by a table explaining the different parameters (or arguments) that can be passed to the function. Note also that the third through fifth arguments are enclosed in curly braces – this means they are optional. You don’t have to pass them to the function if you don’t want to change from the default value. Scroll down a little further to the Code sample section for an example of the Python code in action.



1. Here’s a sample of the Python code in action. Notice that they call the Intersect\_analysis function twice with different options, but we will focus on the first call.

**arcpy.Intersect\_analysis([“vegetation\_stands”, “road\_buffer200m”,**

**“water\_buffer100”], “mysites”, “ALL”, “”, “”)**

Let’s break this down. The first argument is a Python list containing the feature classes we want to find the intersection points for. The basic syntax behind a Python list is

*[list\_item\_1, list\_item\_2, …]*

Therefore, this function finds the intersection points of *vegetation\_stands, road\_buffer200m,* and *water\_buffer100.*

The second argument, *mysites*, is the name of the output feature class containing the intersection points. This feature class will be automatically created – it should not exist before calling the function.

The third, fourth and fifth arguments, “ALL”, “”, and “”, relate to the *join\_attributes*, *cluster\_tolerance*, and *output\_type* parameters. As we saw in the table above under Syntax, these are optional and the author did not necessarily need to include them. “ALL” is already the default for the join\_attributes parameter, and the empty quotation marks for the fourth and fifth arguments are interpreted as the defaults as well.

1. Now that we’ve seen an example, we need to adapt this to our task. First, let’s plan out what our arguments will be.

For the first argument, *in\_features*, we pass a list containing our input feature classes.

**[“Routes4\_Pline\_subset”, “Station\_Points”]**

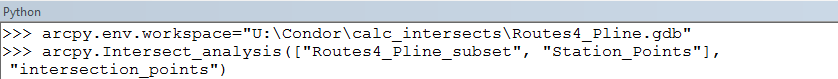
For the second argument, *out\_feature\_class*, we pass a string containing the desired name of our output feature class containing the intersection points. A string is just a piece of text that is defined using quotation marks. This can be anything you want – I’m going to call it intersection\_points.

The defaults are fine for the optional third through fifth arguments, so we’ll leave those out completely.

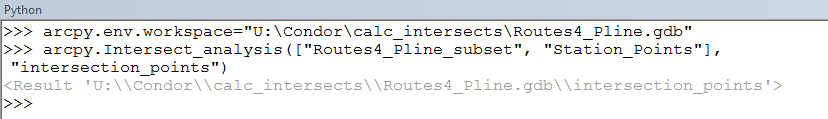
1. We’re ready to put it all together. Click on the Python shell and type the full command:

**arcpy.Intersect\_analysis([“Routes4\_Pline\_subset”, “Station\_Points”], “intersection\_points”)**

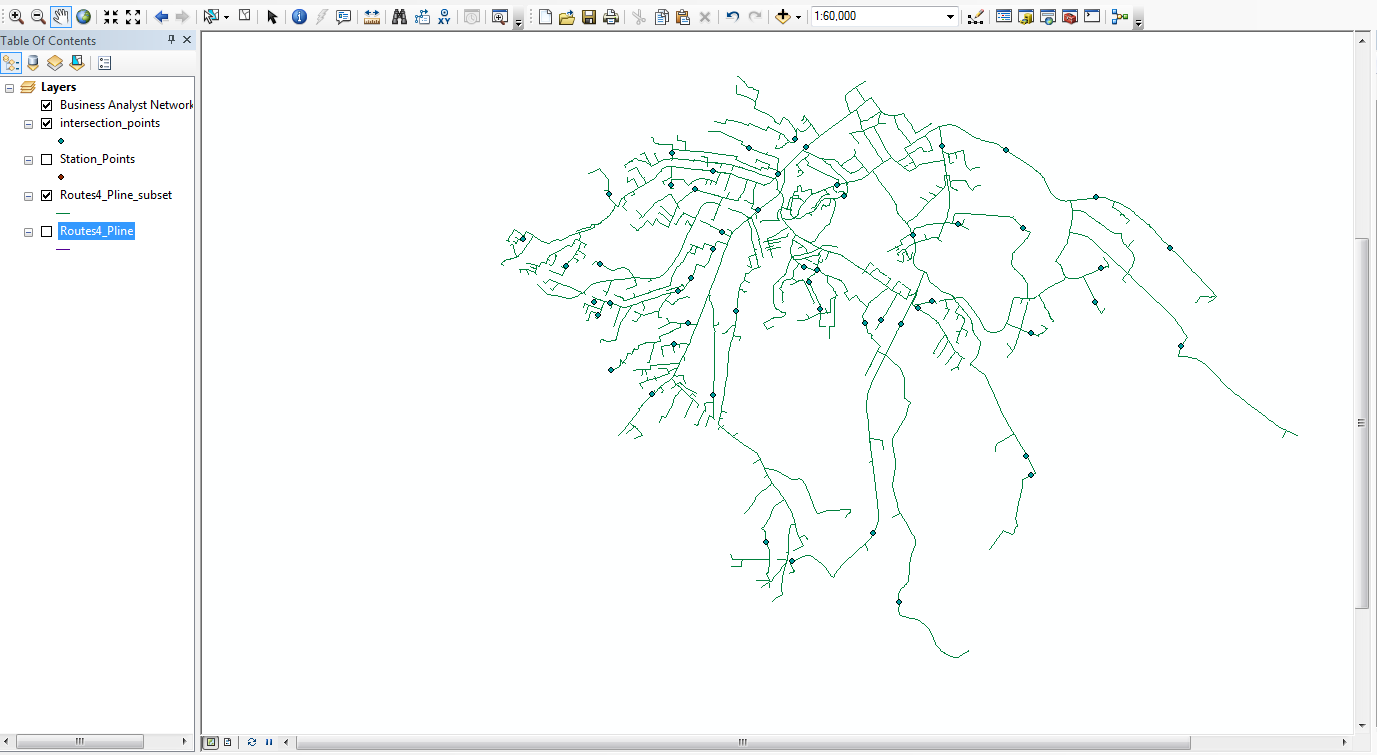
Note that splitting a command over multiple lines is ok in this case – Python considers any code within parentheses to be part of a single line.



If you Enter, the Intersect tool will run.



1. You’re done! If run, a resulting class, intersection\_points, should automatically appear as a feature layer. Note that the two sets of intersection points you would get from using the GUI Intersect tool and from using Python would be exactly the same, because underneath the hood, ArcGIS processed your data in the same way.

****

# **Writing Your Own Scripts**

## For Splitting Your Datasets

For small tasks, it is easy to use the terminal to run all of your commands, but for other tasks it’s better to create files to store your scripts. For this example, we have a large dataset so we want to split the feature class into manageable subsets and do computations through condor. Using this method will drastically cut down on the processing time, so you may follow along on your computer. We will start by making a script that we will call split.py.

To do this, click on the windows button and search for IDLE (python GUI). This will open the IDLE terminal that we will use to run our scrips, but first let’s save it in a new file. Click the file button and then open a new file. Make sure to save this file in your working directory (D:\calc\_intersects) as split.py.

For this task, we will need some more complicated Python tools. Python divides much of its functionality into different *modules* that you can import as needed. Let’s import all the modules we will need for this script:

**import os**

**import shutil**

**import arcpy**

A brief description of each module:

os and shutil contain functions for working with file systems (copying files, changing directories, etc).

arcpy contains all the tools from ArcGIS that we will use in our script.

Let’s start by defining our workspace.

**arcpy.env.workspace = "Routes4\_Pline.gdb"**

We’ll do the same for the feature classes we plan to use.

**routesFC = “Routes4\_Pline”**

**pointsFC = “Station\_Points”**

The next step is to count the number of polyline features we have in the routes feature class. We will need this in order to split the class into chunks. We can do this using a Python function called GetCount\_management. The GetCount\_management function will pull the number of features from the Attribute Table of the Routes4\_Pline feature class. So using this function, we can define a variable called numObservations to hold that number.

**numObservations = int(arcpy.GetCount\_management**

**(polylineFC).getOutput(0))**

Next, let’s tell the computer how big to make each subset of the routes. 14,000 sounds right, since we have roughly 280,000 routes to divide, and 280000/14000 will give us about 20 subsets.

**subsetEntries = 14000**

We can use this number to calculate the number of new feature classes we will be creating, which will be the total number of features divided by the number of features desired in each subset.

**numNewFeatureClasses = (numObservations // subsetEntries) + 1**

The // means we are dividing, but discarding the remainder because we don’t want to end up with a fractional number. For example, if we had 500 features and wanted to make subsets of 200 features each, we need two feature classes of 200, plus a third to hold the remainder of 100 (that’s what the + 1 at the end is for).

Excellent! Now we need to loop through the feature class, breaking off chunks of size subsetEntries and exporting them to a new subset in a new geodatabase. Here’s how that is set up:

**for i in range(1, (numNewFeatureClasses + 1)):**

This is what is called a for loop. The code inside the loop will run once for each number *i* in the range from 1 to the number of feature classes we are creating. We need to add one to the top end, because Python is inclusive at the bottom of a range and exclusive at the top end. That is to say, if you were to write for i in range(a, b), it would start at *i = a*, but it would end at *i = b – 1*.

The size of the loop is defined in python using indentation. Everything in the same loop needs to be indented the same amount. The easiest way to keep up with indentation is through tabs.

Now, let’s specify what each loop iteration will do. We should specify a name for the new feature class we’ll make, as well as the new geodatabase we will save it in. No need to get fancy – we’ll just take the name of the original feature class and append the index number to it, so we know which subset it is. Then, we’ll name the geodatabase after it.

**subsetFC = routesFC + '\_' + str(i)**

**outGeodatabase = subsetFC + '.gdb'**

As you see, strings can be concatenated with the + operator. *i* represents the current index number – this variable will increment by one with every iteration of the loop.

Since we’ll split based on the object ID, for each iteration, we need to figure out which object ID will be our starting point and which will be the ending point.

Let’s say we want subsets of 200 features each. We can make a little table to help figure out how to determine the start and endpoints:

|  |  |  |
| --- | --- | --- |
| **ITERATION (*i)*** | **START ID** | **END ID** |
| 1 | 1 | 200 |
| 2 | 201 | 400 |
| 3 | 401 | 600 |

Notice that the start ID is always one greater than (i – 1) times the subset size, and the end ID is always that number plus the subset size, minus one. We can turn this into code:

**startNdx = ((i – 1) \* subsetEntries) + 1**

**endNdx = startNdx + subsetEntries - 1**

To use this information to select the subset of features, we need to phrase it in the form of a query that ArcGIS understands. ArcGIS uses this query expression to select features based on a list of attributes. This statement looks like:

WHERE OBJECTID\_1 >= *startNdx* AND OBJECTID\_1 <= *endNdx*

Note that the object ID is named OBJECTID\_1 in the attribute table for this feature class.

We need to construct this as a string to pass to the arcpy Select\_management function. This function assumes the “WHERE” part, so we don’t need to include that. Our code therefore looks like:

**whereClause = '"OBJECTID\_1" >= ' + str(startNdx) + ' AND "OBJECTID\_1" <= ' + str(endNdx)**

That’s all we need to select the subsets. The Arcpy function for Select is:

arcpy.Select\_analysis(*input\_feature\_class, output\_feature\_class, whereClause*).

(Remember, you can always search for the right function to use, using the Search window in ArcMap.)

Our input feature class will be the class we’re selecting from: routesFC.

Our output feature class will be the new class we want to export the subset to: subsetFC.

Our whereClause is the query statement we just wrote: whereClause.

Given those arguments, the code looks like:

**arcpy.Select\_analysis(routesFC, subsetFC, whereClause)**

This will create a subset from the input feature class and save it to a new feature class. Now, we just need to package that subset feature class to be sent off for processing. Every feature class must be packed in its own geodatabase, so let’s make a geodatabase specifically for this subset:

**arcpy.CreateFileGDB\_management(“.”, outGeodatabase)**

This creates a new geodatabase in the current folder named after the subset feature class (remember, outGeodatabase is a variable that holds the string *subsetFC* + *.gdb)*.

Next, we need to export the subset feature class to the new geodatabase. Because each computer in the Condor network needs to find the intersection between the subset it receives and the entire set of points, we also need to send the Points feature class along with the subset. Therefore, we will export the feature classes in the form of a list to the geodatabase:

**arcpy.FeatureClassToGeodatabase\_conversion([subsetFC, pointsFC], outGeodatabase)**

Lists in python are defined using square brackets [ ] with your list items separated by commas.

Now that we’ve exported the subset feature class its own geodatabase, there’s no need to keep it around in the original geodatabase. Delete it using

**arcpy.Delete\_management(subsetFC)**

One last step within the loop. We should compress the new geodatabase in a ZIP file, since we’ll be sending it across a network, and compressing it to a smaller file will make things go a lot faster. To make a new ZIP file, we’re going to use the make\_archive command from the shutil module in Python. The syntax goes like this:

shutil.make\_archive(*file\_name, format*, *root\_directory, base\_directory)*

* *file\_name* is what we want the ZIP file to be named, minus the .zip extension. It’s fine, even preferable, if we keep the same name as the geodatabase it contains, so let’s just use outGeodatabase.
* *format* is the compression format we want to use. In this case, that’s ‘zip’.
* *root\_directory* will be the folder that contains the files/folders we want to zip. That’s the current directory, so we will use “.”
* base\_directory is the directory that will be the first level of the zip file. Since a geodatabase is just a directory, and that’s the base prefix of all the actual files, we will again use outGeodatabase.

Put it all together, and we get:

**shutil.make\_archive(outGeodatabase, ‘zip’, “.”, outGeodatabase)**

You’re done with the loop. All we have left is to clean up all the geodatabases we made. They’re all zipped up, so we should delete the originals. To do this, we can use the function rmtree in the shutil module to delete everything from disk.

**shutil.rmtree(outGeodatabase)**

That’s it for the splitting script! To run the script, click on the run menu and click “Run Module”.

1. import os, shutil
2. import arcpy
3. arcpy.env.workspace = "Routes4\_Pline.gdb"
4. routesFC = "Routes4\_Pline"
5. pointsFC = "Station\_Points"
6. numObservations = int(arcpy.GetCount\_management(routesFC).getOutput(0))
7. subsetEntries = 14000
8. numNewFeatureClasses = (numObservations // subsetEntries) + 1
9. for i in range(1, (numNewFeatureClasses + 1)):
10. subsetFC = routesFC + '\_' + str(i)
11. outGeodatabase = subsetFC + '.gdb'
12. startNdx = ((i - 1) \* subsetEntries) + 1
13. endNdx = startNdx + subsetEntries – 1
14. whereClause = '"OBJECTID\_1" >= ' + str(startNdx) + ' AND " OBJECTID\_1" <= ' + str(endNdx)
15. arcpy.Select\_analysis(routesFC, subsetFC, whereClause)
16. arcpy.CreateFileGDB\_management(".", outGeodatabase)
17. arcpy.FeatureClassToGeodatabase\_conversion([subsetFC, pointsFC], outGeodatabase)
18. arcpy.Delete\_management(subsetFC)
19. shutil.make\_archive(outGeodatabase, 'zip', ".", outGeodatabase)
20. shutil.rmtree(outGeodatabase)

## For Submitting your Job in Condor

Now that the data is all split up, we need to submit it to Condor for intersection. There are two files we need to submit to Condor: a submit file and a python script.

To tell the Condor scheduler how to distribute your jobs, you need to create a Condor submit file. Most of the time, this will only involve taking a template and making slight modifications for your project, which is good because submit files can be a bit cryptic. This file is not written in Python, but in a custom syntax.

Download the submit folder from this link:

<https://drive.google.com/file/d/0B5fzAJKv9ly7aTVCVDNhU00teVU>

Then, copy the template intersect\_stations.sub from your downloads folder to your working folder.

You can use the submit file to pass strings to your Python script as arguments. The Intersect script is going to want to know what the geodatabase and feature classes are called, so let’s create variables at the top of the submit file (if they don’t already exist) by adding these lines at the top, right underneath the comments.

**geodatabasePrefix = Routes4\_Pline**

**pointsFC = Station\_Points**

Next, we need to tell Condor what the intersect script will be called. I am going to call it intersect.py, but you can name your script differently.

**scriptName = intersect.py**

Only a couple more things to change. Scroll down to the “executable” section and make sure that the version of ArcGIS listed matched the version you are currently using.

**executable = C:\Python27\ArcGIS10.5\python.exe**

Scroll down to “arguments.” This will be the actual command that the destination computer runs. We will have three arguments: the name of the Python script itself, the routes feature class name, and the points feature class name. Since these are already stored in variables, we can use these variables with the following syntax: $(*variable\_name)*

**arguments = $(scriptName) $(routesFC) $(pointsFC)**

The last change is to tell Condor what data files to transfer. For us, that’s the zipped geodatabases. These will be named using the form Routes4\_Pline\_1.gdb.zip, Routes4\_Pline\_2.gdb.zip, etc. Since the Routes4\_Pline part is already saved in the variable geodatabasePrefix, we can use this syntax to tell it to grab anything starting with that:

**queue filename matching files $(geodatabasePrefix)\*.gdb.zip**

And the submit file is complete.

## For Running the Intersection on your data

Next, we need to create the script that actually performs the intersection. First, you’ll need to open a new IDLE document and save a blank file in your working directory called intersect.py.

Again, we need to import some modules:

**import os, sys, arcpy, shutil**

**from zipfile import ZipFile**

You should recognize os, shutil, and arcpy from the split script we just wrote. We’re going to import sys to work with some system functions (like command-line arguments) and zipfile to easily work with ZIP files,

First, we need the names of the routes and points feature classes. Fortunately, we passed the routesFC argument and the pointsFC as command line arguments. We can access command-line arguments by using the syntax sys.argv[*index]*, where *index* equals the position of the argument in the list of all command-line arguments. For example, a program that accepts three arguments would have sys.argv[1], sys.argv[2], and sys.argv[3]. The routes and point feature classes are in command-line arguments one and two, so we should save those to convenient variables

**routesFC = sys.argv[1]**

**stationsFC = sys.argv[2]**

Next, we need to know the name of the geodatabase we’re working with. We know the geodatabase was named <*subsetFC>*.gdb where subsetFC is our routesFC:

**workingGDB = routesFC + “.gdb”**

Remember, the geodatabase is zipped up, so let’s extract it. To find its name, we’ll use glob to get a list of every zip file in the folder, then use the ZipFile(*file\_to\_extract)*.extractall() from the zipfile module function to extract its contents.

**ZipFile(workingGDB + “.zip” ).extractall()**

We need to make sure to define our workspace so condor knows where to save the new intersected results.

**arcpy.env.workspace = workingGDB**

We should create a name for our resulting feature class. After the intersection points are found, they will be saved to this class. A convenient choice would be the same thing as the routes feature class name, with an intersect\_ prefix to identify it as the intersection points output:

**intersectFC = “intersect\_” + routesFC**

That’s all we need to run the Intersect function. The syntax for this is the same as when we ran it inside ArcMap, so it should be familiar to you:

arcpy.Intersect\_analysis(*[feature\_classes\_to\_intersect*], *output\_feature\_class*)

In our case, we will type:

**arcpy.Intersect\_analysis([routesFC, stationsFC], intersectFC)**

And the actual Intersect is done. Zip up the geodatabase the exact same way that you did in the last script:

**shutil.make\_archive(workingGDB, ‘zip’, “.”, workingGDB)**

…and clean up the old geodatabase:

**shutil.rmtree(workingGDB)**

1. import os, sys, shutil
2. from zipfile import ZipFile
3. import arcpy
4. routesFC = sys.argv[1]
5. stationsFC = sys.argv[2]
6. intersectFC = "intersect\_" + routesFC
7. workingGDB = routesFC + ".gdb"
8. ZipFile(workingGDB + ".zip").extractall()
9. arcpy.env.workspace = workingGDB
10. arcpy.Intersect\_analysis([routesFC, stationsFC], intersectFC)
11. shutil.make\_archive(workingGDB, 'zip', ".", workingGDB)
12. shutil.rmtree(workingGDB)

You’re all done with the Intersect script!

# **Submitting your Job to the pool**

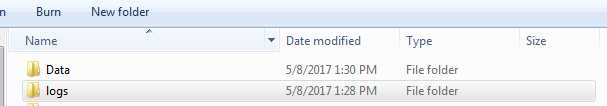
To submit a job, make sure your intersect.py, intersect\_stations.sub, and zipped up data subsets are all in the same folder. It is okay if other files, like your other Python scripts, are in the folder.

Time to send our files to Condor. Hold shift and right click the white space inside your folder. In the menu that appears, select “Open command window here”.

First, you’ll need to add your credentials to the Condor database by typing “condor\_store\_cred add”. When prompted, enter your username and password.

****

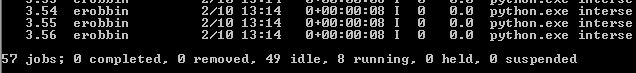
Next, create an empty folder named logs in your working folder. Condor will use this folder for its log files.



Finally, run your Condor job by typing “condor\_submit intersect\_stations.sub”. Your job cluster will be submitted to Condor, using the instructions in the submit file.

****

You may check the status of your job at any time by typing “condor\_q” into the command line. You’ll know the job is done when this list is empty and you see “0 jobs”.

****

When it’s done, all of the resulting zipped geodatabases will be in your working folder.

# **Merging your results**

The final step is to merge all our subsets back together, so we can get a complete set of the intersection points for the whole dataset.

Open up one more IDLE file and name it merge.py, then import the following modules.

**import os, sys, shutil, glob**

**from zipfile import ZipFile**

**import arcpy**

First things first, let’s also define routesFC:

**routesFC = “Routes4\_Pline”**

We’re about to create a loop to go through the intersection point feature classes in each of the geodatabases and merge them all together. Before we do that, we should tell Python what we expect their names will be, and create an empty list to add them to (which will come in handy when it’s time to merge and clean up):

**intersectFC = “intersect\_” + routesFC**

**intersectFCList = []**

Let’s also save the name of the main geodatabase, because we’ll be switching between geodatabases inside the loop:

**mainGeodatabase = “Routes4\_Pline.gdb”**

Next, we need to count and unzip all those geodatabases that came back from the intersect. We know they all end in “.gdb.zip”, so we can use the glob function from the glob module to get everything in the folder that matches that pattern:

**geoDBList = glob.glob(“\*.gdb.zip”)**

This returns a list of everything in the folder matching that pattern. To get the count of items in a list, simply use the length function, len():

**numGeoDBs = len(geoDBList)**

Now, time to loop. We will have geodatabases with index numbers 1 to N, where N equals the number of geodatabases we’re looping through. Therefore, our for loop declaration looks like this:

**for i in range(1, numGeoDBs + 1):**

Remember that the top of a range is exclusive, meaning if you want to include numGeoDBs, you need to set your range to one above it.

Inside the loop, we first need to set the names we’re using. Remember, when we split the routes feature class, we appended the loop iteration (index) number to the subset feature classes and geodatabases by its iteration number when we first created these geodatabases. That number stuck with it throughout the whole process. Therefore, let’s make sure we account for that:

**inGeodatabase = routesFC + “\_” + str(i) + “.gdb”**

**currIntersectFC = intersectFC + “\_” + str(i)**

And remember, we haven’t unzipped anything yet. Let’s do that

**inGeodatabaseZipped = inGeodatabase + “.zip”**

**ZipFile(inGeodatabaseZipped).extractall()**

We want to enter the freshly unzipped geodatabase and grab its intersection points feature class. First thing to do is set the active ArcPy workspace to it:

**arcpy.env.workspace = inGeodatabase**

Now we can export the subset’s intersection points (currIntersectFC) to the main geodatabase. We have done this before, so the syntax shouldn’t be too tough:

**arcpy.FeatureClassToGeodatabase\_conversion([currIntersectFC], mainGeodatabase)**

Note that the first argument always takes a list, even if it’s just a list of one item.

Don’t forget to add the intersection points feature class to the list we made before the loop (intersectFCList):

**intersectFCList.append(currIntersectFC)**

We no longer need the subset geodatabase or the zipped files, so we should delete them:

**shutil.rmtree(inGeodatabase)**

**os.remove(inGeodatabase + ".zip")**

And that’s it for the loop body. Make sure to reset the indention to what you had before you started the loop.

Reset the workspace to the main geodatabase (after all, it’s the only one left!):

**arcpy.env.workspace = mainGeodatabase**

Now we’re ready to merge the subset intersection points. The syntax for that command is:

arcpy.Merge\_management(*list\_of\_feature\_classes\_to\_merge, output\_feature\_class)*

Remember, we saved all of the subset intersection point feature classes in the list called intersectFCList. And you can call your output feature class anything you’d like – I’ll stick with appending \_merged to intersect\_Routes4\_Pline. Therefore, the full command is:

**arcpy.Merge\_management(intersectFCList, intersectFC + “\_merged”)**

One step left in the whole project – deleting all the individual subset intersection points you copied to the main geodatabase. We have a list of their names, so it’s easy to loop through and delete each one.

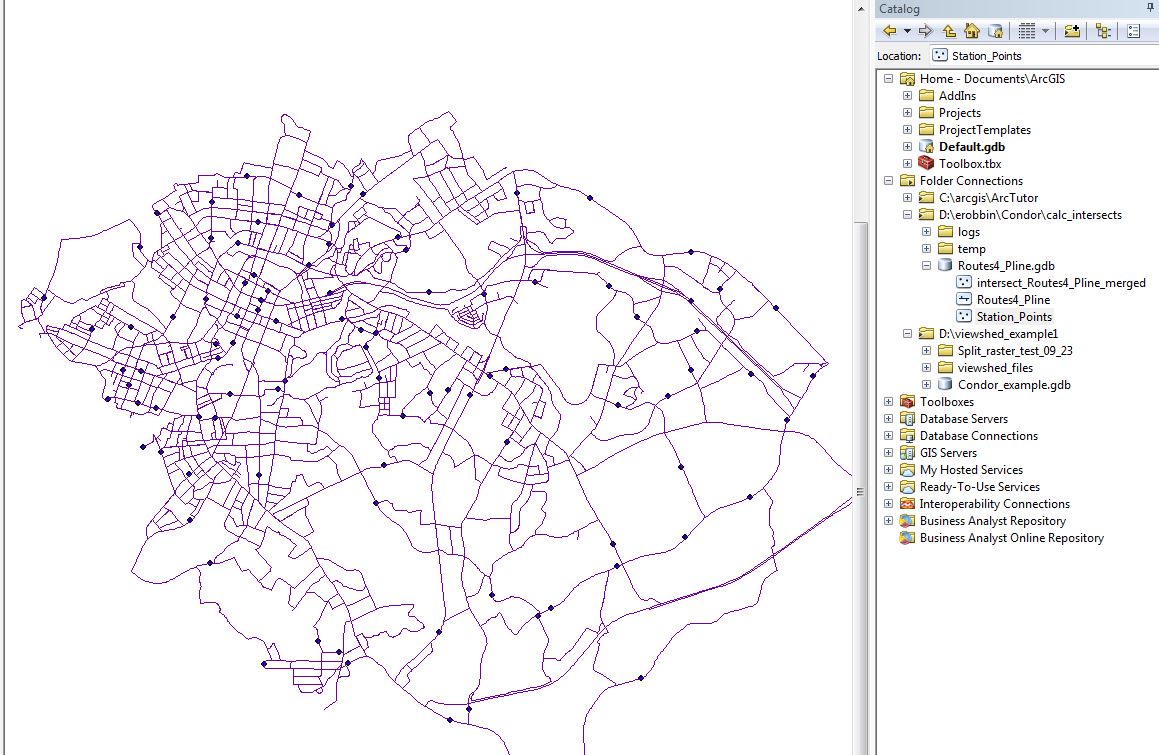
**for listToDel in intersectFCList:**

**arcpy.Delete\_management(listToDel)**

Run your script to obtain your final merged feature class.

1. import sys, os, shutil, glob
2. from zipfile import ZipFile
3. import arcpy
4. routesFC = "Routes4\_Pline"
5. intersectFC = "intersect\_" + routesFC
6. intersectFCList = []
7. mainGeodatabase = "Routes4\_Pline.gdb"
9. geoDBList = glob.glob("\*.gdb.zip")
10. numGeoDBs = len(geoDBList)
12. for i in range(1, numGeoDBs + 1):
13. inGeodatabase = routesFC + "\_" + str(i) + ".gdb"
14. currIntersectFC = intersectFC + "\_" + str(i)
15. inGeodatabaseZipped = inGeodatabase + ".zip"
16. ZipFile(inGeodatabaseZipped).extractall()
17. arcpy.env.workspace = inGeodatabase
18. arcpy.FeatureClassToGeodatabase\_conversion([currIntersectFC], mainGeodatabase)
19. intersectFCList.append(currIntersectFC)
20. shutil.rmtree(inGeodatabase)
21. os.remove(inGeodatabase + ".zip")
22. arcpy.env.workspace = mainGeodatabase
23. arcpy.Merge\_management(intersectFCList, intersectFC + "\_merged")
24. for listToDel in intersectFCList:
25. arcpy.Delete\_management(listToDel)

After that’s done, you’re done too! Congratulations. You should have a new feature class in your original geodatabase called intersect\_Routes4\_Pline\_merged. Check it out in ArcMap!

****