

Optimizing script efficiency

1

This video will discuss some tips for increasing the speed and efficiency of a script. We'll compare the speed of different scripting approaches using the Spatial Join exercise as our example.

Script efficiency

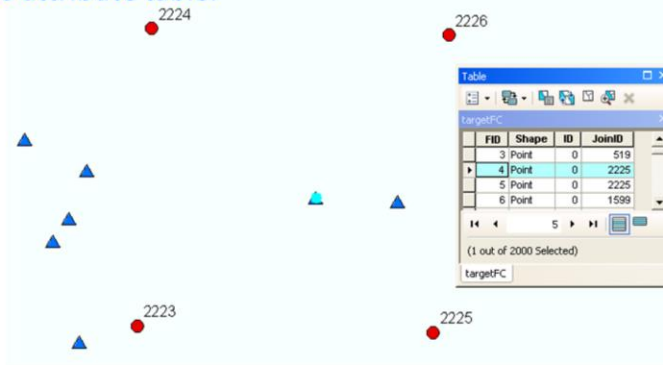
- More efficient scripts save time and computing resources.
- There are many approaches to accomplish a task – some are much faster than others.
- The `time.clock` tool is very helpful for testing the speed of different approaches.

2

Making a script as efficient as possible can be important when working with large datasets or processes that require large numbers of calculations. There are usually many different ways to accomplish a task in a script but it isn't always obvious beforehand which approach will be fastest. The `time.clock` function can be very helpful in identifying the methods and approaches that will improve script speed.

Spatial join exercise -

- Develop a script that spatially joins point features from two different datasets
 - For each point in dataset 1, get the FID value that corresponds to the nearest point in dataset 2 and write to attribute table.

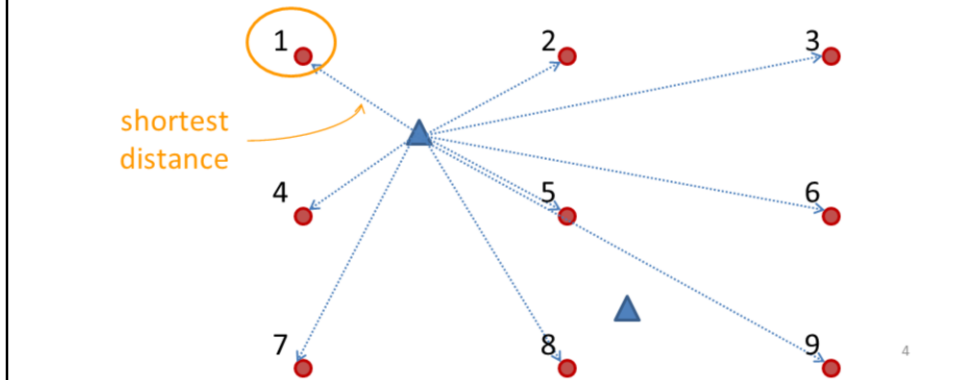


3

We will use the Spatial Join exercise to explore some ways to improve the efficiency of a script. Recall that the objective of this exercise was to spatially join the points features from two different datasets. For each target feature in dataset1, the FID of the nearest join feature in dataset 2 needed to be recorded in the attribute table of dataset 1. The exercise did not allow the use of ArcGIS's Spatial Join tool.

Basic strategy

- For each feature in targetFC, calculate the distance to every feature in joinFC...
- identify the minimum distance and the FID of the joinFC feature associated with that distance.



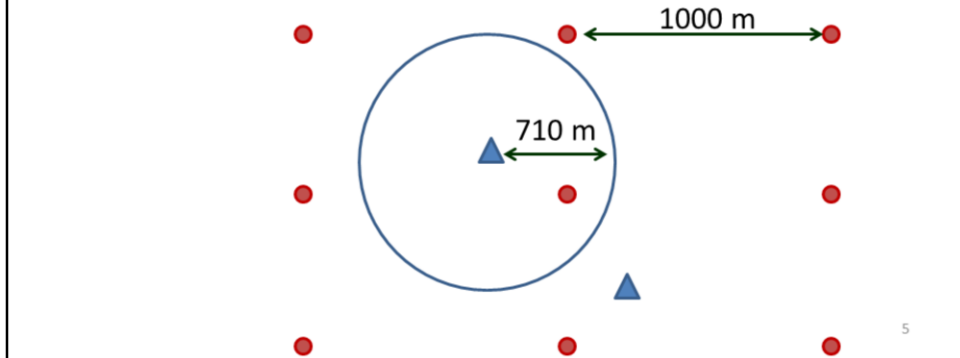
The basic approach for accomplishing the objective in the Spatial Join exercise is to calculate the distances from each **target feature** to every other **join feature**.

The join feature with the minimum distance to a given target feature is identified and matched to the target.

This animation illustrates the process for a single target point.

Step 1 – remove irrelevant points from joinFC

- If points in joinFC are uniformly spaced (i.e. in a grid), we may be able to eliminate some points prior to analysis...
 - determine distance, D, between joinFC points
 - use SelectLayerByLocation to select joinFC points within a distance larger than $\frac{1}{2} (D * \sqrt{2})$ of the targetFC points (use larger distance if joinFC point spacing is not perfectly uniform)



The first step to improving the efficiency of the script in this exercise is to eliminate any unnecessary points from the joined feature class. This will reduce the number of points that will need to be evaluated for each target feature.

In the exercise, the join features are regularly spaced in a grid. We can determine the distance between the points in ArcMap.

The exercise instructions allow us to use the SelectLayerByLocation tool so we can use it to select join feature points that are within specified distance of the target features. In this case, the spacing between the join feature points is 1000 m. We can calculate the minimum radius around a target feature within which we should be guaranteed to find a join feature. If the join features were not uniformly spaced, then we would have to use a larger search radius to be sure that a join feature will be found.

Spatial join exercise – reducing the dataset

```
arcpy.MakeFeatureLayer_management(joinFC, "joinFC_lyr")  
arcpy.SelectLayerByLocation_management("joinFC_lyr",  
    "INTERSECT", targetFC, "710 METERS")
```

Total number of
items in joinFC

3509

Number of selected
items in joinFC

1116

- Number of calculations reduced by 2/3

6

In the script, we can use the **SelectLayerByLocation** tool to select the join features that are within 710 m of the target features.

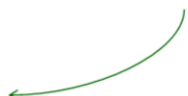
By restricting the analysis to the selected join features, we reduce the number of join features that must be considered from 3509 to 1116. This reduces the number of the total number of calculations that need to be performed by 2/3.

Pairing the features – nested cursors

- Use cursors to read data from each dataset.
- Need to pair each targetFC point with every joinFC point with a nested for loop.
- How should the cursors be applied? Nest the joinFC cursor?

```
rows_target = arcpy.UpdateCursor(targetFC)
for row_target in rows_target:
    # get X and Y from row_target
    rows_join = arcpy.SearchCursor(joinFC)
    for row_join in rows_join:
        # find minimum distance...
```

recreate or reset cursor:
rows_join.reset()
to reset



The script needs to pair each target feature with each join feature.

This requires a 2-level for loop in which the 1st level iterates through the target features and the 2nd level iterates through the join features.

There are a few different ways that we can use the cursors in this case.

The first approach is to put the join feature class cursor within the nested loop. This will require recreating or reset the cursor for each iteration of the target feature loop.

Pairing the features – using a dictionary

- Nesting cursors requires the joinFC cursor to be created or reset 2000 times (once for each targetFC feature).
- Recreating cursors can be eliminated by putting relevant joinFC info into a dictionary. The dictionary can then be used in place of cursor.

```
joinFC_dct = {}  
rows_join = arcpy.SearchCursor(joinFC)  
for row_join in rows_join:  
    # add XY data to dictionary  
rows_target = arcpy.UpdateCursor(targetFC)  
for row_target in rows_target:  
    # get X and Y from row_target  
    for ID in joinFC_dct:  
        # find minimum distance...
```

put joinFC data into dictionary

retrieve data from dictionary for each target feature

If we nested the joinFC search cursor within targetFC loop, then the cursor would need to be iterated 2000 times.

In this next approach, we'll avoid re-iterating the joinFC cursor by storing the relevant data from the joinFC in a dictionary.

The first step uses the joinFC cursor is used to add the coordinate data to the dictionary. The keys for the dictionary correspond to the joinFC FID values.

The second step in the script iterates the target feature cursor. This time we'll iterate through the joinFC dictionary rather than the joinFC cursor.

Pairing the features – using a list

- Another option for avoiding nested cursors...
 - use a list to store joinFC data instead of a dictionary

```
joinFC_lst = []  
rows_join = arcpy.SearchCursor(joinFC)  
for row_join in rows_join:  
    # append XY data to list  
rows_target = arcpy.UpdateCursor(targetFC)  
for row_target in rows_target:  
    # get X and Y from row_target  
    for ID, x2, y2 in joinFC_lst:  
        # find minimum distance...
```

put joinFC data into list

retrieve data from list for each target feature

9

Another approach to avoiding the nested joinFC cursor is to use a list to store the joinFC data instead of a dictionary.

In this case, the joinFC data are stored in a 2-level list with each sub-list containing the FID and the x and y coordinates.

Within the target cursor loop, the joinFC list is iterated in place of the dictionary or the cursor.

Pairing the features – speed test

- So which option for pairing the features is fastest?

nested cursors

≈1600
seconds



dictionary option

2.9
seconds



list option

2.5
seconds



- All things being equal, lists are fastest followed closely by dictionaries. Always avoid nesting cursors.

10

It may not be obvious beforehand which of the 3 methods we just discussed will be fastest.

For this slide, I've tested the run time for each of the methods using the **time.clock** tool...

The nested cursor approach is by far the slowest having taken more than 1000 times longer than the other two approaches. Lists are slightly faster to iterate than dictionaries. For each iteration of the loop, the cursor took only a fraction of a second longer than the other methods. However, this small difference was greatly magnified by the loop. Loops are the most important part of a script to focus on when trying to improve script speed.

Calculating distance to all pairs of features

- To identify the point nearest to a targetFC point, we need to calculate distances to all joinFC points...

```
rows_target = arcpy.UpdateCursor(targetFC)
```

```
for row_target in rows_target:
```

```
    # get x1 and y1 from row_target centroid
```

```
    dMin = 99999999
```

```
    join_ID = -1
```

} set null values for minimum distance (dMin) and join_ID

```
    for ID, x2, y2 in joinFC_lst:
```

```
        d = ((x1-x2)**2 + (y1-y2)**2)**0.5
```

```
        if d < dMin:
```

```
            dMin = d
```

```
            join_ID = ID
```

} keep minimum distance and associated ID for joinFC feature

11

Within the targetFC loop, we need to identify the join feature that is closest to each target feature.

The approach taken in this example calculates the distance to all join features for each target feature. Assuming there are 2000 target features and 1000 join features, then this requires distance to be calculated 2 million times.

Comparisons instead of calculations

- Do we really need to calculate distances for all pairs?
- We could just calculate distances for **joinFC** points that are close to targetFC point...

```
# get x1 and x2.
```

```
# set null values for dMin and join_ID
```

```
minX, maxX = x1-1100, x1+1100
```

```
minY, maxY = y1-1100, y1+1200
```

```
for ID, x2, y2 in joinFC_lst:
```

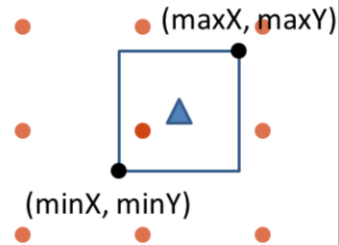
```
    if minX < x2 < maxX:
```

```
        if minY < y2 < maxY:
```

```
            # calculate distance and keep dMin and
```

```
            # associated join_ID
```

only calculate distance if x2 is
in range and if y2 is in range



12

Each distance calculation requires several math operations – each of which takes time.

This slide will present an alternate approach in which distances are calculated only for join features that are close to a given target feature.

We can define a square buffer around each target feature that is large enough to guarantee that it contains at least one join feature but small enough that it can contain at most 4 join features.

In the targetFC loop, we can define the coordinates of the lower left and upper right corners of the buffer for a given target feature.

In the joinFC loop, we can use conditional statements to test if the join feature falls within the buffer - we'll only calculate distances to these join features.

Comparisons vs. calculations – speed test

- Which can you do faster?

True or false?
 $634,218.1 < 694,357.7$

Or...

$$\begin{array}{r} 694,357.7 \\ - 634,218.1 \\ \hline 60,139.6 \end{array}$$

calculate distance
for all pairs

9.8
seconds



calculate distance
for few pairs

2.4
seconds



- All things being equal, it is faster to make comparisons (e.g. greater than, less than) instead of calculations. 13

What can you do faster- compare two numbers or subtract them?

We're definitely faster at comparing numbers but how about the computer?

It turns out that computers are faster at comparisons too. In this example, the reducing the number of distance calculations improved the script speed by 400%.

Module functions

- Programming languages have to be translated into a form the computer understands (i.e. machine code)
- Translation occurs on-the-fly unless the script is compiled
- Modules are compiled (i.e. translated) before running.
- Can we speed things up by having the script call a custom module?

```
rows = arcpy.UpdateCursor(outputFC)
for row in rows:

    centroid = row.getValue("Shape").centroid
    x1, y1 = centroid.X, centroid.Y

    s = time.clock()
    join_ID = spatial_join_helper.getClosestFC(x1, y1, joinLst)
    e = time.clock()
    elapsedT += e-s

    row.JoinID = join_ID

rows.updateRow(row)
```



```
01110011 01100101 01110010 01
01100101 01110010 00100000 01
01101000 01100001 01110100 00
01100100 01101001 01110011 01
01110010 01101001 01100010 01
01110100 01100101 01110011 00
01100001 01101110 01111001 00
01101001 01101110 01100011 01
01101101 01101001 01101110 01
00100000 01101101 01100101 01
01110011 01100001 01100111 01
01110011 00100000 01110100 01
```

14

The purpose of a scripting language is to translate our commands into instructions that the computer can understand.

Translation of a script into machine code occurs on-the-fly unless the script is first compiled.

Scripts are not typically compiled unless they are module scripts.

We'll test how much of a difference that using compiled module scripts can make for our Spatial Join exercise.

Modularizing the script

- Turn section that identifies ID of nearest joinFC into a function and put in a separate script file.

```
def getClosestFeature (x1, y1, joinLst):
```

```
    minX, maxX = x1-1100, x1+1100
```

```
    minY, maxY = y1-1100, y1+1100
```

```
    minD = 99999
```

```
    join_ID = -1
```

```
    for ID, x2, y2 in joinLst:
```

```
        if minX < x2 < maxX:
```

```
            if minY < y2 < maxY:
```

```
                d = ((x2-x1)**2 + (y2-y1)**2)**0.5
```

```
                if minD > d:
```

```
                    minD = d
```

```
                    join_ID = ID
```

```
    return join_ID
```

Note: there are no arcpy
functions included

15

In this slide, I've taken the contents of the targetFC loop and put them into a function in a module script. Recall that a module script is contained in a separate file from the script that calls the function.

In the script on this slide, note that there are no arcpy statements within the function.

Modularizing the script

- Calling the module function from the main script...

```
rows = arcpy.UpdateCursor(outputFC)
```


```
for row in rows:
```

```
    centroid = row.getValue("Shape").centroid
```

```
    x1, y1 = centroid.X, centroid.Y
```

```
    join_ID = spatial_join_module.getClosestFC (x1, y1, joinLst)
```

name of
module script



name of module
function



16

In the main script, I've added a statement in the targetFC loop that calls the function from the module.

Module vs. no module – speed test

- Do functions in modules run faster?

no modules

2.4
seconds



with module

1.2
seconds



-
- Module scripts are compiled so functions in modules run up to 2 times faster.

17

Do modules really provide a speed advantage?

Let's put it to the test and compare the script that uses the module function with the one that doesn't...

The script with the module function is twice as fast, in this case.

Arcpy vs. python functions in a module – speed test

- What if we also included the arcpy statements in the module?

module without
arcpy statements

4.5
seconds



module with arcpy
statements

4.5
seconds



-
- arcpy functions do not run any faster when in a module.

18

Let's see if there is any advantage to putting arcpy statements in module functions.

Here, we'll compare a script in which the arcpy statements placed in module functions to a script that calls the arcpy statements directly.

It turns out that there is no speed advantage for putting arcpy statements in modules. Arcpy functions are already compiled by their native language, and so they cannot be further compiled in python.

Conclusions

- Avoid creating a cursor more than once for a dataset
 - i.e. always avoid creating cursors in a loop
- Lists are slightly faster than dictionaries – but time savings could be significant for large datasets.
- Reduce the number of datapoints that need to be processed, if possible.
- Use comparisons to minimize the number of calculations required.
- Convert python code to functions in module scripts
 - no advantage for putting arcpy code in modules

19

Some take-home messages from this video are to

1. Avoid creating a cursor more than one time for a given dataset and never create a cursor within a loop. Cursors are complicated functions with a lot of capabilities – they will always be slower than basic data collections such as lists or dictionaries.
2. Lists are slightly faster than dictionaries – all things being equal. The time savings for lists will be more substantial when working with loops with large numbers of iterations. There are, of course, occasions when dictionaries are more efficient or convenient than lists. Recall from the previous video that lists are much faster than arrays – it is more efficient to convert the array to a list before iterating.
3. Reduce the number of data points that need to be processed whenever possible. The fewer the data points, the fewer the calculations that will be needed.
4. Comparisons are always faster than calculations
5. Converting python statements to module functions can improve the speed by 200%. There is no benefit, in terms of speed, for putting arcpy statements in a module function.

Extra credit challenge

- Record-time for Spatial Join script from past students...

with scipy and
"shapefile" module

0.5
seconds!!



instructor's
script

12
seconds



Shapefile module available at: <https://code.google.com/archive/p/pyshp/> ²⁰

To conclude this video, I'll show the timed results for the two students who hold the record for the fastest Spatial Join script from this course in the past.

The scripts from both students had the same processing time. We'll compare the processing times of their scripts and the instructor's script which set the standard.

The student's script was nearly 25 times faster than the instructor's script. The instructor's script was optimized by using `arcpy` – iterating through the two cursors accounted for 90% of the processing time. The student's scripts used the `shapefile` module to avoid using cursors altogether.

The `shapefile` module can be downloaded online at the link provided. It has the same basic capabilities of `arcpy`'s cursor tools but is somewhat less user-friendly.