Session 2:Python in Field Calculations

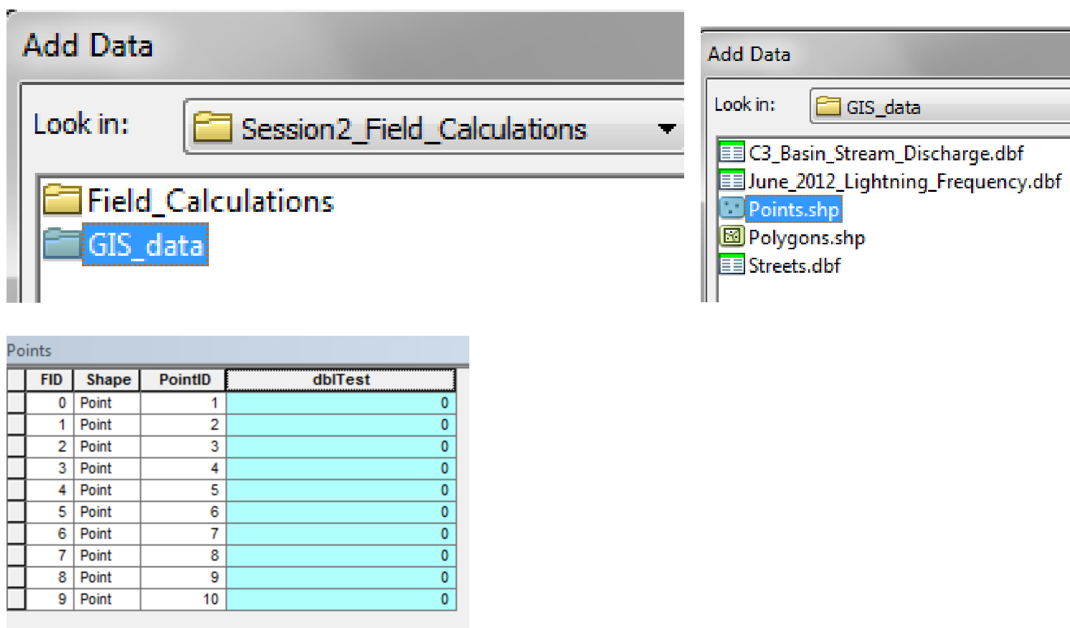
In this session we will use the Field Calculator and create some Python functions.

Python field calculation expressions can be saved to disk as a .cal file, emailed, and shared among users. We will write Python expressions to:

1) Compute new values from an input field value.
2) Compute the distance between successive points.
3) Make decisions based on input field values
4) Make decisions based on a lookup table or Python dictionary

Basic Python Field Calculations

Add the points shapefile to your ArcMap Data frame.

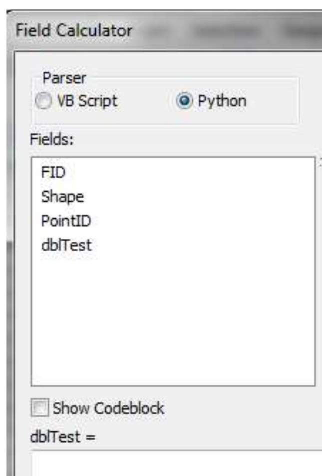


FID	Shape	PointID	dblTest
0	Point	1	0
1	Point	2	0
2	Point	3	0
3	Point	4	0
4	Point	5	0
5	Point	6	0
6	Point	7	0
7	Point	8	0
8	Point	9	0
9	Point	10	0

Our goal is to take PointID squared and put it in the dblTest field…

Shape	PointID	dblTest
Point	1	1
Point	2	4
Point	3	9
Point	4	16
Point	5	25
Point	6	36
Point	7	49
Point	8	64
Point	9	81
Point	10	100

Use your Field Calculator to start a calculation on the double precision field named dblTest. Select the Python parser…



Check on ***Show Codeblock***---this is where we would build a Python function..since we are short on time***, load a .cal file that contains the Python function…***

Building Python functions. We started with a # which is the start of a Python comment, describing you function.

Show Codeblock

Pre-Logic Script Code:

```
#function to return input value squared
```

Next, the function starts with a ***def*** keyword, followed by your function name and a colon.

Show Codeblock

Pre-Logic Script Code:

```
#function to return input value squared
def valueSquared(input) :
```

the parameter in parenthesis (input) represents the ArcGIS field value that is passed to your Python function

We passed the field value to the Python function in the main field calculator window....functionname***(!fieldname!)***

Show Codeblock

Pre-Logic Script Code:

```
#function to return input value squared
def valueSquared(input) :
```

dblTest =

```
valueSquared( !PointID!)
```

Finally ***return*** the value squared in your Python function...

Show Codeblock

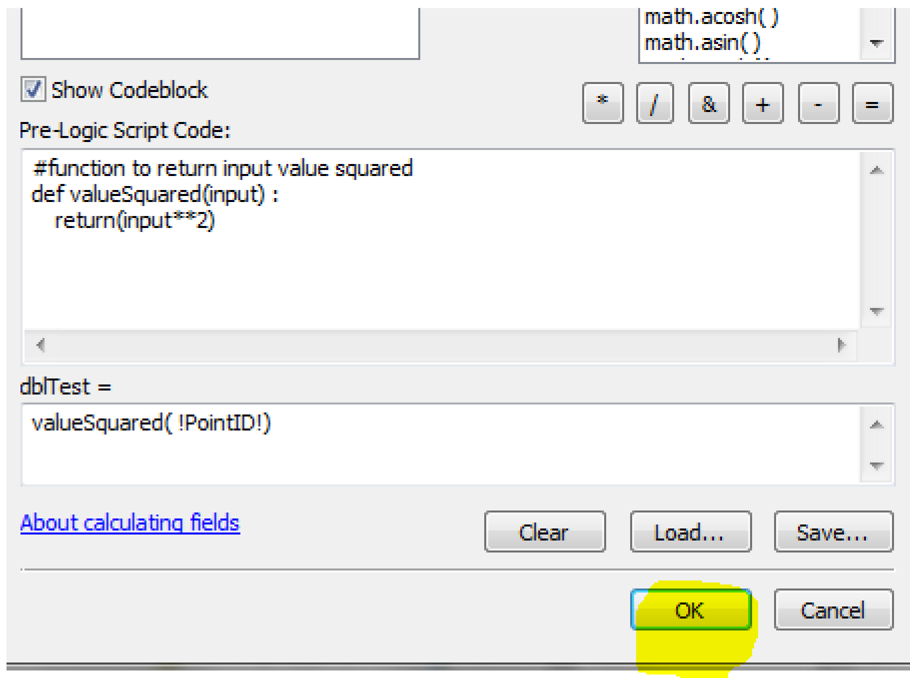
Pre-Logic Script Code:

```
#function to return input value squared
def valueSquared(input) :
   return(input**2)
```

dblTest =

```
valueSquared( !PointID!)
```

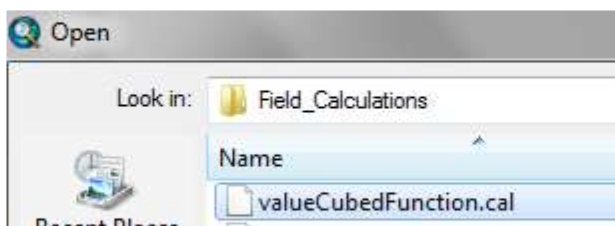
Note that all lines inside your function must be indented!



Press the OK button to execute your Python field calculation…

PointID	dblTest
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

Next, load the Python field calculation ***valueCubedFunction.cal***



There are ***four errors*** in the valueCubed field calculation…can you find all four errors?

Pre-Logic Script Code:

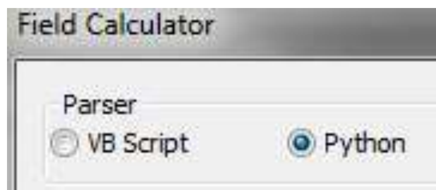
```
#function to return input value cubed
def valueCubed( input )
return( input**3  )
```

dblTest =

```
value_Cubed( !PointID!)
```

See the next page for explanation of the four errors in the above field calculation…

1) Need to specify Python parser in field calculation



2) def statement must end with a colon

```
def valueCubed( input ) :
```

3) Incorrect function name

```
dblTest =
value_Cubed( !PointID!)
```

4) Statements after function definition need to be indented

```
def valueCubed( input ) :
return( input**3 )
```

Success!:

dblTest
1
8
27
64
125
216
343
512
729
1000

Show Codeblock

Pre-Logic Script Code:

```
#function to return input value cubed
def valueCubed( input ) :
   return( input**3 )
```

```
dblTest =
valueCubed( !PointID!)
```

Computing Distance Between Points

Use the ***Add XY*** geoprocessing tool to add the X, Y coordinates from each Point shape property to your attribute table.

Points

FID	Shape	PointID	dblTest	POINT_X	POINT_Y
0	Point	1	1	395535.0005	7073266.4782
1	Point	2	4	412200.0544	7071527.5161
2	Point	3	9	423648.2219	7077179.1431
3	Point	4	16	437849.7462	7075150.3539
4	Point	5	25	434661.6489	7066745.3702
5	Point	6	36	433647.2543	7053413.327
6	Point	7	49	434516.7354	7049065.9216
7	Point	8	64	428720.1949	7058485.2999
8	Point	9	81	449877.5677	7053413.327
9	Point	10	100	453210.5785	7056746.3378

We can compute the distance between successive points as:

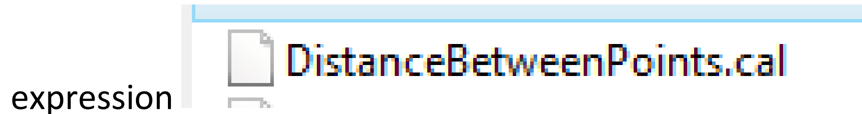
SquareRoot[$(X1 - X2)^2 + (Y1 - Y2)^2$]

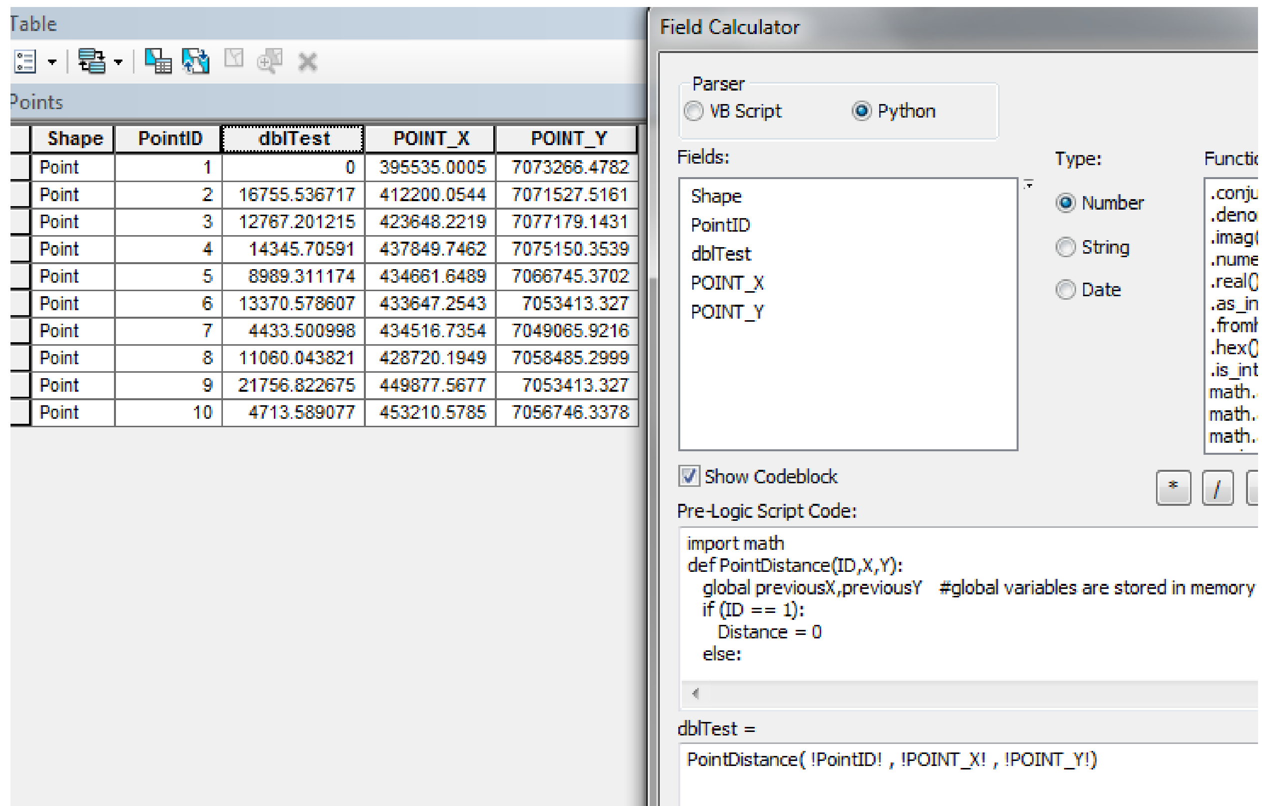
Your field calculation needs to remember the previous point X,Y values…we do this using a global statement. Do a field calculation to your dblTest field….Load field calculation from the file ***ExampleGlobal.cal***

```
def PointDistance(ID,X):
  global previousX  #global variables are stored in memory
  if( ID ==1):
    Distance = 0
  else:
    Distance = previousX
  previousX = X
  return Distance
```

Shape	PointID	dblTest	POINT_X
Point	1	0	395535.0005
Point	2	395535.00	412200.0544
Point	3	412200.05	423648.2219
Point	4	423648.22	437849.7462
Point	5	437849.74	434661.6489
Point	6	434661.64	433647.2543
Point	7	433647.25	434516.7354
Point	8	434516.73	428720.1949
Point	9	428720.19	449877.5677
Point	10	449877.56	453210.5785

To compute the distance between successive points, you need to take the square root of the difference in X and the difference in Ys. You import the math module to get access to the math.sqrt function…there are hundreds of Python modules you can import for access special functions. Use the field calculator to compute the distance between points and store the result in dblTest field….

To compute the length between each point, load and execute the field calculation expression 



Shape	PointID	dblTest	POINT_X	POINT_Y
Point	1	0	395535.0005	7073266.4782
Point	2	16755.536717	412200.0544	7071527.5161
Point	3	12767.201215	423648.2219	7077179.1431
Point	4	14345.70591	437849.7462	7075150.3539
Point	5	8989.311174	434661.6489	7066745.3702
Point	6	13370.578607	433647.2543	7053413.327
Point	7	4433.500998	434516.7354	7049065.9216
Point	8	11060.043821	428720.1949	7058485.2999
Point	9	21756.822675	449877.5677	7053413.327
Point	10	4713.589077	453210.5785	7056746.3378

```
import math
def PointDistance(ID,X,Y):
  global previousX,previousY   #global variables are stored in memory
  if (ID == 1):
    Distance = 0
  else:
```

```
PointDistance( !PointID! , !POINT_X! , !POINT_Y!)
```

Select the first 2 points and use the measure tool to check your field calculation.

Table

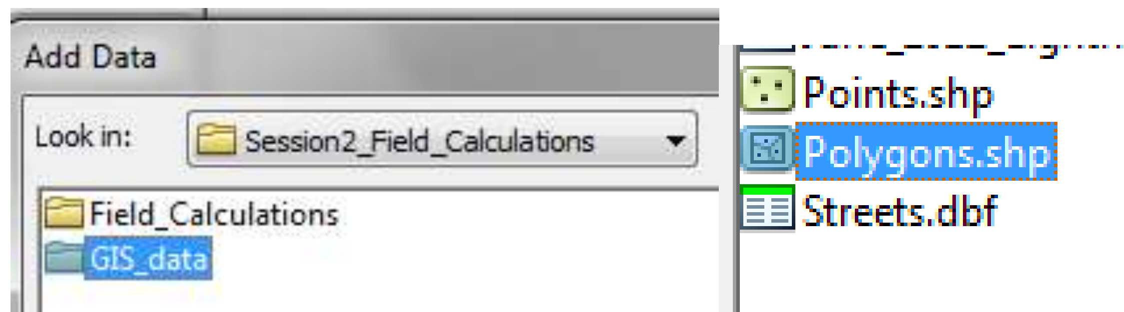
Points

FID	Shape	PointID	dblTest	POINT_X
0	Point	1	0	395535.00
1	Point	2	16755.53671	412200.05
2	Point	3	12767.20121	423648.22
3	Point	4	14345.70591	437849.74
4	Point	5	8989.311174	434661.64
5	Point	6	13370.57860	433647.25
6	Point	7	4433.500998	434516.73
7	Point	8	11060.04382	428720.19
8	Point	9	21756.82267	449877.56
9	Point	10	4713.589077	453210.57

If():Elif():Else: Decisions

In the previous field calculation, we set distance to zero if we had the first point, otherwise for each other point, we computed distance as the distance from the previous point. We did this using a Python if(): else: block.

In this example, we will compute a polygon edge index based on each polygon's area/perimeter ratio. Add the feature class ***Polygons.shp***

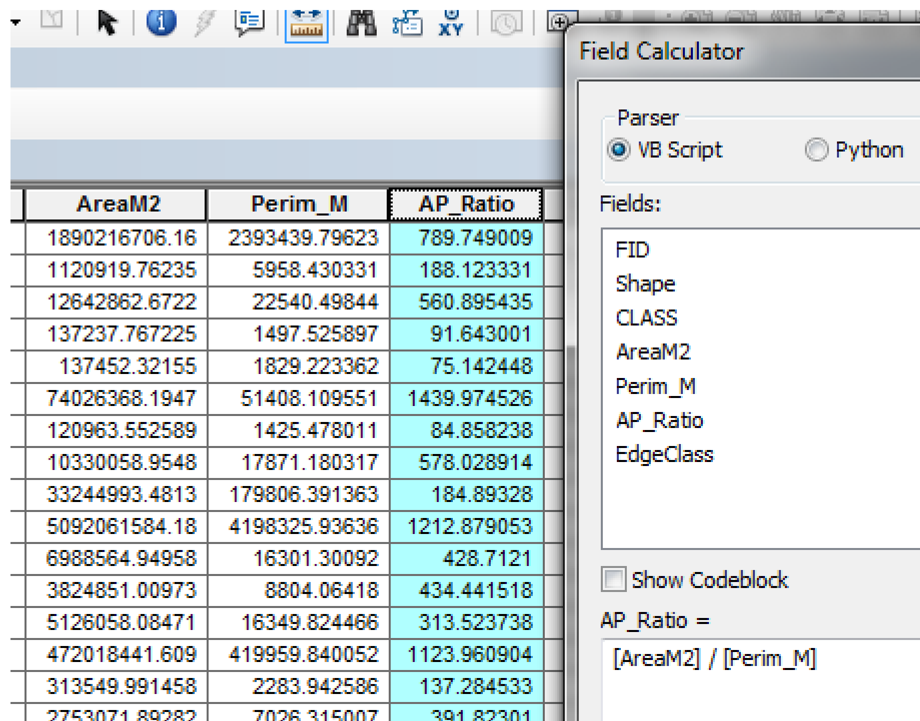


to your arcmap data frame

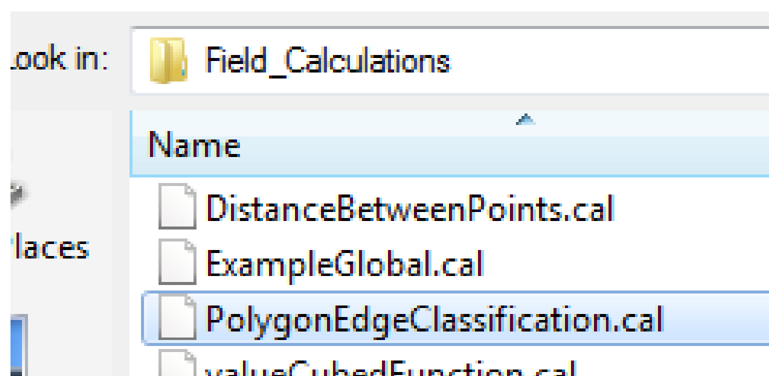
Polygons

Shape	CLASS	AreaM2	Perim_M
Polygon	Decid/Mixed/Shrub	1890216706.16	2393439.79623
Polygon	Alpine and Barren	1120919.76235	5958.430331
Polygon	Alpine and Barren	12642862.6722	22540.49844
Polygon	Decid/Mixed/Shrub	137237.767225	1497.525897
Polygon	Alpine and Barren	137452.32155	1829.223362
Polygon	Alpine and Barren	74026368.1947	51408.109551
Polygon	Alpine and Barren	120963.552589	1425.478011

Use the field calculator to compute AP_Ratio as the Area divided by the Perimeter of each polygon.



Load the Python field calculation file to compute ***EdgeClassl*** based on the Area/Perimeter ratio..



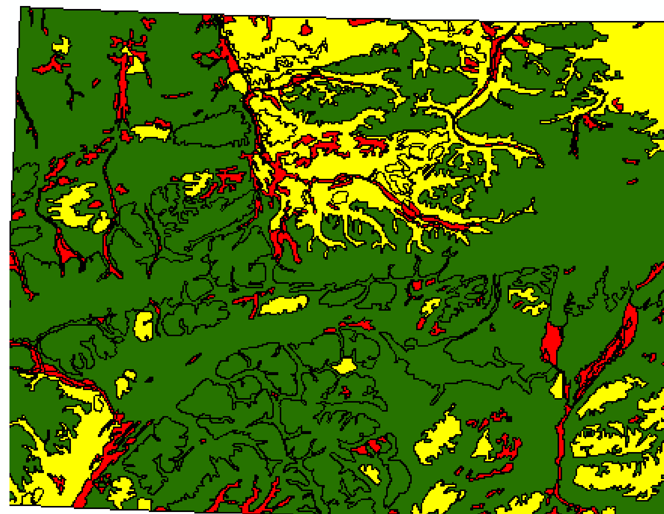
```
def myFunction(area,perimeter):
  ShapeIndex = area / perimeter
  if ( ShapeIndex) > 1000:
    txtFld = "Low"
  elif (ShapeIndex) > 500:
    txtFld = "Medium"
  else:
    txtFld = "High"
  return txtFld
```

So if the area/perimeter ratio is greater than 1000, the edge index class for the polygon will be “Low”, between 501 and 1000 will be “Medium” and less than 500 will be “High”.

AP_Ratio	EdgeClass
789.7	Medium
188.1	High
560.9	Medium
91.6	High

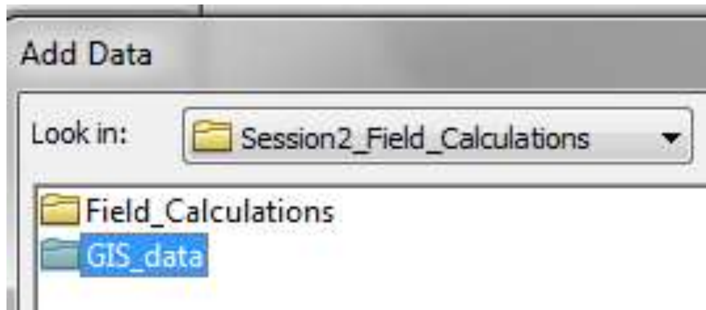
Symbolize your polygons based on their edge class values after you execute the Python function in your field calculation.

<Heading>	EdgeClass	251
Low	Low	24
Medium	Medium	48
High	High	179



Decisions Using A Python Dictionary

A Python Dictionary is a look-up table that allows you to make efficient decisions if your query involves a category. For example, add the Streets.dbf table to your arcmap data frame.



Streets

STREETNAME	MPH
REITEN RD	30
E TEMPERANCE ST	25
E MEEKER ST	25
E MEEKER ST	25
E CHERRY HILL ST	25
OLYMPIC PL	25
ALPINE WY	25

We want to create a lookup table for the following street suffix values:

RD	Road
ST	Street
PL	Place
LN	Lane
AVE	Avenue
CT	Court
WY	Way

You can do this by creating a Python dictionary where the structure is {keyvalue:newvalue,keyvalue:newvalue,keyvalue:newvalue}

First create a ***NewName*** text field of 25 characters length (since that is the length of the STREENAME field). Next, we will use the Python .title() function to convert the string from all uppercase to mixed case.

STREETNAME	MPH	NewName
REITEN RD	30	Reiten Rd
E TEMPERANCE ST	25	E Temperance St
E MEEKER ST	25	E Meeker St
E MEEKER ST	25	E Meeker St
E CHERRY HILL ST	25	E Cherry Hill St
OLYMPIC PL	25	Olympic Pl
ALPINE WY	25	Alpine Wy
E SEATTLE ST	25	E Seattle St
E SEATTLE ST	25	E Seattle St
VIEW PL	25	View Pl
E SEATTLE ST	25	E Seattle St
99 PL	25	99 Pl
S 265 PL	25	S 265 Pl
99 PL	25	99 Pl
118 PL	25	118 Pl
118 PL	30	118 Pl
SE 260 ST	25	Se 260 St

Field Calculator

Parser: VB Script / Python

Fields: OID, STREETNAME, MPH, NewName

Show Codeblock

NewName =

```
!STREETNAME!.title()
```

Next we will grab the suffix from each string and use a Python dictionary to convert the short suffix to a longer suffix. To grab the suffix, we split the string into a Python list and the grab the last element of the list.

Try the following, load the ***TextSplit.cal*** file

STREETNAME	NewName
REITEN RD	Rd
E TEMPERANCE ST	St
E MEEKER ST	St
E MEEKER ST	St
E CHERRY HILL ST	St
OLYMPIC PL	Pl
ALPINE WY	Wy
E SEATTLE ST	St
E SEATTLE ST	St
VIEW PL	Pl
E SEATTLE ST	St
99 PL	Pl
S 265 PL	Pl
99 PL	Pl

Show Codeblock

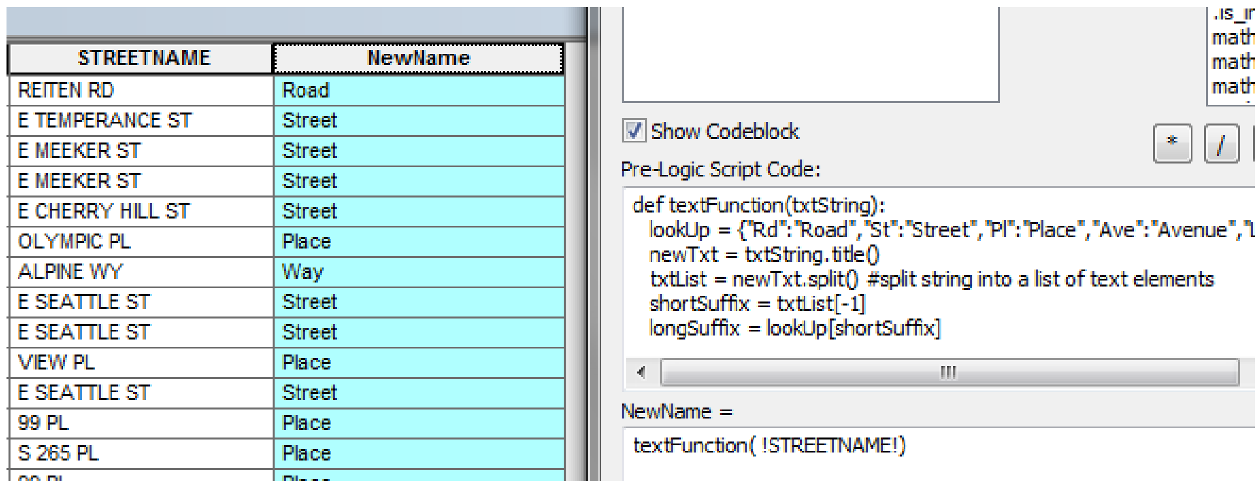
Pre-Logic Script Code:

```
#function to split string, return last item in list
def textFunction(txtString):
  newTxt = txtString.title()
  txtList = newTxt.split()
  return txtList[-1]
```

NewName =

```
textFunction( !STREETNAME! )
```

Next, build and use a Python dictionary to replace the short suffix string with a long suffix string. Load the ***lookUp.cal*** Python field calculation.



Finally, use ***lookUp2.cal*** to complete the field calculation:

For example, if the first input txtString is “REITEN RD” the function returns newString “Reiten Road”:

```
def textFunction(txtString):
  lookUp = {"Rd":"Road","St":"Street","Pl":"Place","Ave":"Avenue","Ln":"Lane","Ct":"Court","Wy":"Way"}
```

RD	Road
ST	Street
PL	Place
LN	Lane
AVE	Avenue
CT	Court
WY	Way

STREETNAME
REITEN RD

newText = txtString.title() returns variable newText with value “Reiten Rd”

txtList = newText.split() returns list txtList with values [“Reiten”, “Rd”]

shortSuffix = txtList[-1] returns variable shortSuffix with value “Rd”

longSuffix = lookUp[shortSuffix] returns variable longSuffix with value “Road”

txtList[-1] = longSuffix returns list txtList with values [“Reiten”, “Road”]

newString = " ".join(txtList) returns variable newString with value “Reiten Road”

Streets

OID	STREETNAME	NewName
0	REITEN RD	Reiten Road
1	E TEMPERANCE ST	E Temperance Street
2	E MEEKER ST	E Meeker Street

Let's take a 10-minute break…the next session will be Session 3: Geoprocessing