

The background features a topographic map with various elevation contours and labels like 'Gaspereau Trench' and '439'. This map is overlaid with a grid of semi-transparent triangles in shades of blue, purple, and green. A prominent diagonal band of larger, semi-transparent triangles in shades of yellow, green, and orange runs from the top right towards the bottom left. The text is centered in the blue-toned area.

DEVELOPER SUMMIT

March 10–13



WELCOME

Python: Developing Geoprocessing Tools

David Wynne, Jon Bodamer

Abstract

- Join us as we step through the process of creating geoprocessing tools using Python. Using script tools and Python toolboxes as examples, this workshop will highlight the important decisions in making fully functional geoprocessing tools. Distributing your toolboxes and extending geoprocessing using Python modules will also be discussed.

Pro migration

<http://esriurl.com/PythonProMigration>

- **Python code/tools you've already written ...**
 - arcpy differences
 - Python 2 to 3 differences
- **Can write Python code that will work in both**
 - Analyze Tools For Pro
 - 2to3
- **In the future...**
 - Python distributions will make 2 to 3 issues less important

Python

- Python continues to grow
 - We continue to add more functionality to arcpy
- We want you to accomplish as much as possible through Python (and arcpy)
- *Let us know what you need!*

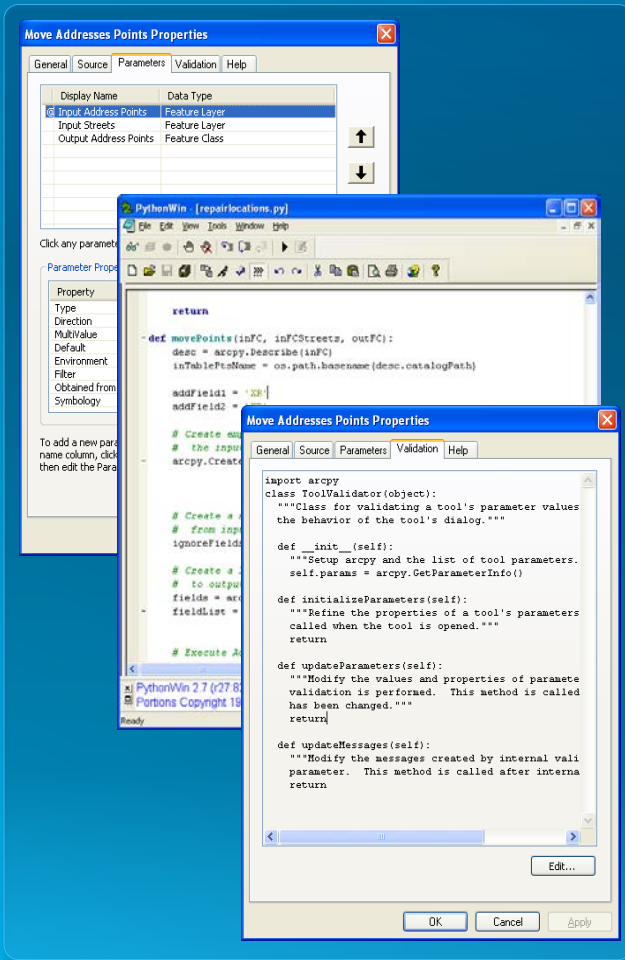
Why we build geoprocessing tools

- **Tools are a well organized functional piece that are ...**
 - **Easy to share**
 - **Generic**
 - **Become part of the geoprocessing framework (Python, dialogs, ModelBuilder, Services)**
 - **Validation minimizes coding checks**

What makes a good geoprocessing tool?

- **Essential and elemental operations on data**
 - **Simple; a reasonable number of parameters**
- **Follows geoprocessing conventions**
- **An output ('required' or 'derived')**
- **Is documented**

Deconstructing a geoprocessing tool



• A geoprocessing tool does three things

1. Defines parameters
2. Validates parameters
3. Executes source

```
import arcpy

class Toolbox(object):
    def __init__(self):
        """Define the toolbox (the name of the toolbox is the name of the
        .pyt file)."""
        self.label = "Toolbox"
        self.alias = ""

        # List of tool classes associated with this toolbox
        self.tools = [Tool]

class Tool(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Tool"
        self.description = ""
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definitions"""
        params = None
        return params

    def isLicensed(self):
        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation for each tool
        parameter. This method is called after internal validation."""
        return

    def execute(self, parameters, messages):
        """The source code of the tool."""
        return
```

Script tools vs Python toolboxes

- Using Python, we can build tools in two ways:

Script tools	Python toolboxes
Since ArcGIS 9.0	Since ArcGIS 10.1
Partially written in Python and divided (parameters / validation / source)	Written entirely in Python

- From a development perspective, Python toolboxes are easier to manage/code
- From a functionality perspective, are nearly equivalent: 'a tool is a tool'

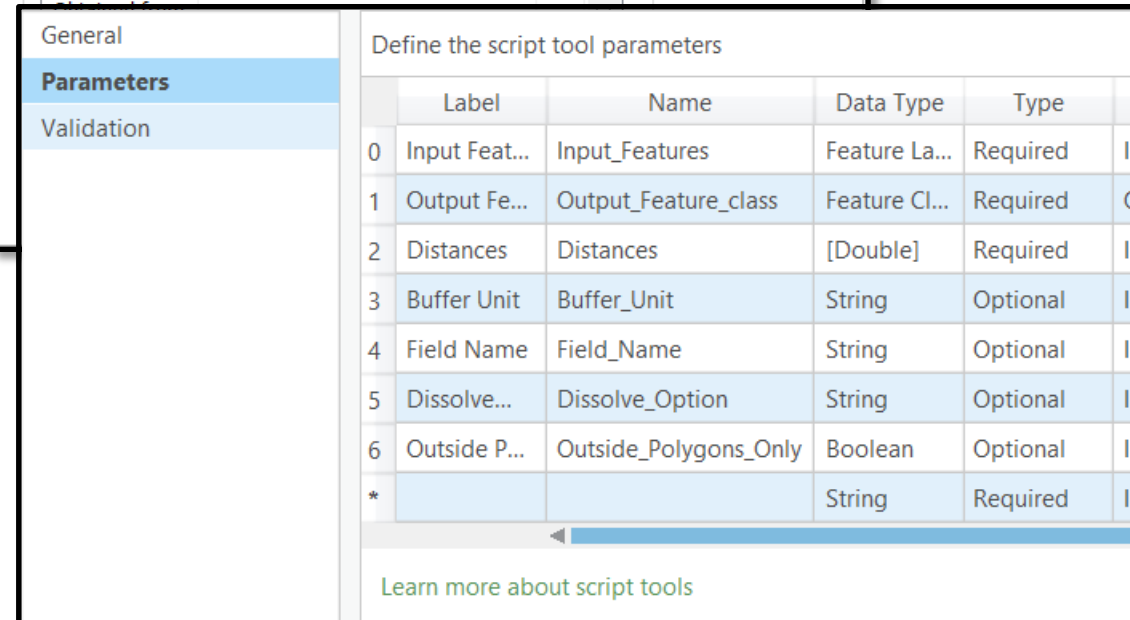
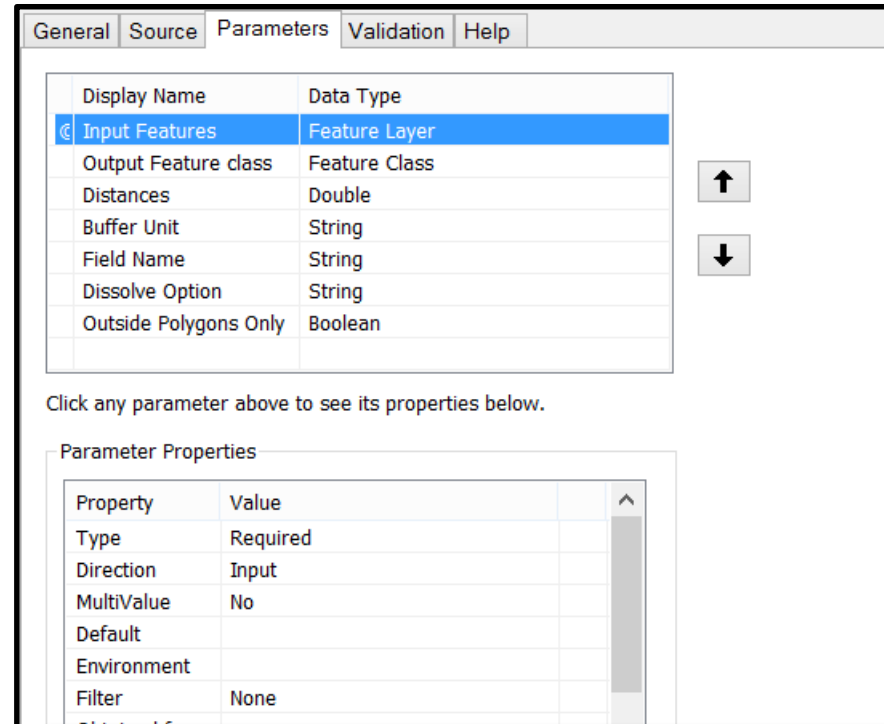
A quick note about Python toolboxes and script tools

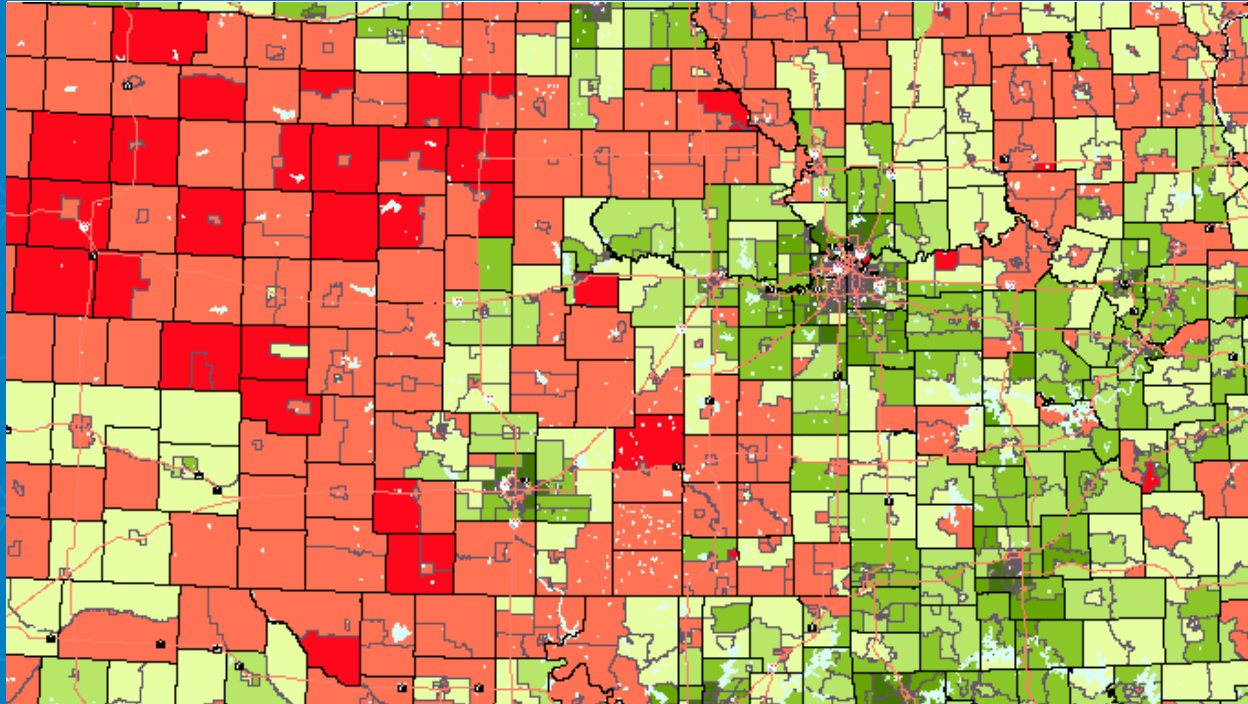
```
def updateMessages(self, parameters):  
    """Modify the messages created by internal validation for each tool  
    parameter. This method is called after internal validation."""  
  
    # Distance should never be negative  
    if parameters[2].value <= 0.0:  
        parameters[2].setErrorMessage(  
            'Distance value cannot be a negative number')  
  
    # If using percentages, distance must be less than 1.0  
    elif parameters[3].value:  
        if parameters[2].value > 1.0:  
            parameters[2].setErrorMessage(  
                'Percentages must be between 0.0 and 1.0')  
  
    return
```

```
def updateMessages(self):  
    """Modify the messages created by internal validation for each tool  
    parameter. This method is called after internal validation."""  
  
    # Distance should never be negative  
    if self.params[2].value <= 0.0:  
        self.params[2].setErrorMessage(  
            'Distance value cannot be a negative number')  
  
    # If using percentages, distance must be less than 1.0  
    elif self.params[3].value:  
        if self.params[2].value > 1.0:  
            self.params[2].setErrorMessage(  
                'Percentages must be between 0.0 and 1.0')  
  
    return
```

Parameters

- Parameters are how you interact with a tool
- Key parameter characteristics
 1. Datatype
 - Feature layers, raster layers, table views
 - String, Boolean, long, double
 2. Direction
 - Input, output
 3. Parameter type
 - Required, optional, derived





Demo: Script tools

Parameters

- As parameters are entered and modified, validation responds and interacts
- Tools validates parameters in two ways
 1. Basic (free) parameter validation, such as:
 - Does an input exist?
 - Is it the right 'type'?
 - Is this value an expected keyword?
 2. Additional rules and behavior you add

```
def getParameterInfo(self):
    """Define parameter definitions"""
    in_lines = arcpy.Parameter(
        displayName='Input Features',
        name='in_features',
        datatype='GPFeatureLayer',
        parameterType='Required',
        direction='Input')

    in_lines.filter.list = ['Polyline']

    out_points = arcpy.Parameter(
        displayName='Output Feature Class',
        name='out_features',
        datatype='DEFeatureClass',
        parameterType='Required',
        direction='Output')

    interval = arcpy.Parameter(
        displayName='Interval (units are in units of input)',
        name='interval',
        datatype='GPDouble',
        parameterType='Required',
        direction='Input')

    use_percentage = arcpy.Parameter(
        displayName='Use as percentage (or value)',
        name='use_percentage',
        datatype='GPBoolean',
        parameterType='Optional',
        direction='Input')

    # Note: 1st position is equivalent to True
    use_percentage.filter.list = ['PERCENTAGE', 'VALUE']
    use_percentage.value = 'VALUE'

    end_points = arcpy.Parameter(
        displayName='Include Start and End Points',
```

Validation

- ‘Custom’ validation
 - Python code that you write
- Provides more control
 - Parameter interaction
 - Calculate defaults
 - Enable or disable parameters
 - Messaging (to guide parameter choices and guard against invalid ones)

```
def updateParameters(self, parameters):
    """Modify the values and properties of parameters before internal
    validation is performed. This method is called whenever a parameter
    has been changed."""

    if parameters[0].value:
        if not parameters[2].altered:
            extent = arcpy.Describe(parameters[0].value).extent
            if extent.width > extent.height:
                parameters[2].value = extent.width / 100.0
            else:
                parameters[2].value = extent.height / 100.0

    return

def updateParameters(self, parameters):
    """Modify the values and properties of parameters before internal
    validation is performed. This method is called whenever a parameter
    has been changed."""

    if parameters[0].value:
        p = feedparser.parse(parameters[0].valueAsText)
        if p['bozo'] == 0: # Successful read
            entry = p.entries[0]
            f_names = entry.keys()
            f_names.remove('georss_point')
            f_names.remove('georss_elev')
            parameters[2].filter.list = f_names
```

Validation: ModelBuilder

- Describe outputs for chaining in ModelBuilder
- Data variables in ModelBuilder include descriptions of data
- By updating the schema, subsequent tools can see pending changes prior to execution



```
parameters[1].parameterDependencies = [parameters[0].name]
parameters[1].schema.clone = True
parameters[1].schema.geometryTypeRule = 'AsSpecified'
parameters[1].schema.geometryType = 'Point'
parameters[1].schema.fieldsRule = 'FirstDependencyFIDs'
```

```
id_field = arcpy.Field()
id_field.name = 'FID_1'
id_field.type = 'Integer'
```

```
parameters[1].schema.additionalFields = [id_field]
```


Validation: Messages

- Parameter messages are added in `updateMessages`

```
def updateMessages(self, parameters):  
    """Modify the messages created by internal validation for  
    parameter. This method is called after internal validation  
    has been performed.  
    """  
  
    # Distance should never be negative  
    if parameters[2].value <= 0.0:  
        parameters[2].setErrorMessage(  
            'Distance value cannot be a negative number')  
  
    # If using percentages, distance must be less than 1.0  
    elif parameters[3].value:  
        if parameters[2].value > 1.0:  
            parameters[2].setErrorMessage(  
                'Percentages must be between 0.0 and 1.0')  
  
    return
```

Validation: Licensing



- Check if a tool is licensed to execute*
- Return False, the tool cannot be executed

- * *Python toolbox only*

```
def isLicensed(self):  
    """Tool can be used if 3D Analyst is available."""  
    try:  
        if arcpy.CheckExtension("3D") != "Available":  
            raise Exception  
    except Exception:  
        return False # tool cannot be executed  
  
    return True # tool can be executed
```

Execution

1. Parameters are received by your code
2. The 'business logic' is performed
 - Communication can be added with messages and progressor
 - *Note: Error messages do not raise exceptions*
3. Derived output values (if any) are pushed back

```
def execute(self, parameters, messages):
    """The source code of the tool."""
    arcpy.AddWarning('starting...')

    in_fc = parameters[0].valueAsText
    out_fc = parameters[1].valueAsText
    interval = parameters[2].value
    is_percentage = parameters[3].value
    end_points = parameters[4].value

    # Create output feature class
    arcpy.CreateFeatureclass_management(
        os.path.dirname(out_fc),
        os.path.basename(out_fc),
        geometry_type='POINT',
        spatial_reference=arcpy.Describe(in_fc).spatialReference)

    # Add a field to transfer objectid field from input
    fid_name = 'FID_1'
    arcpy.AddField_management(out_fc, fid_name, 'LONG')

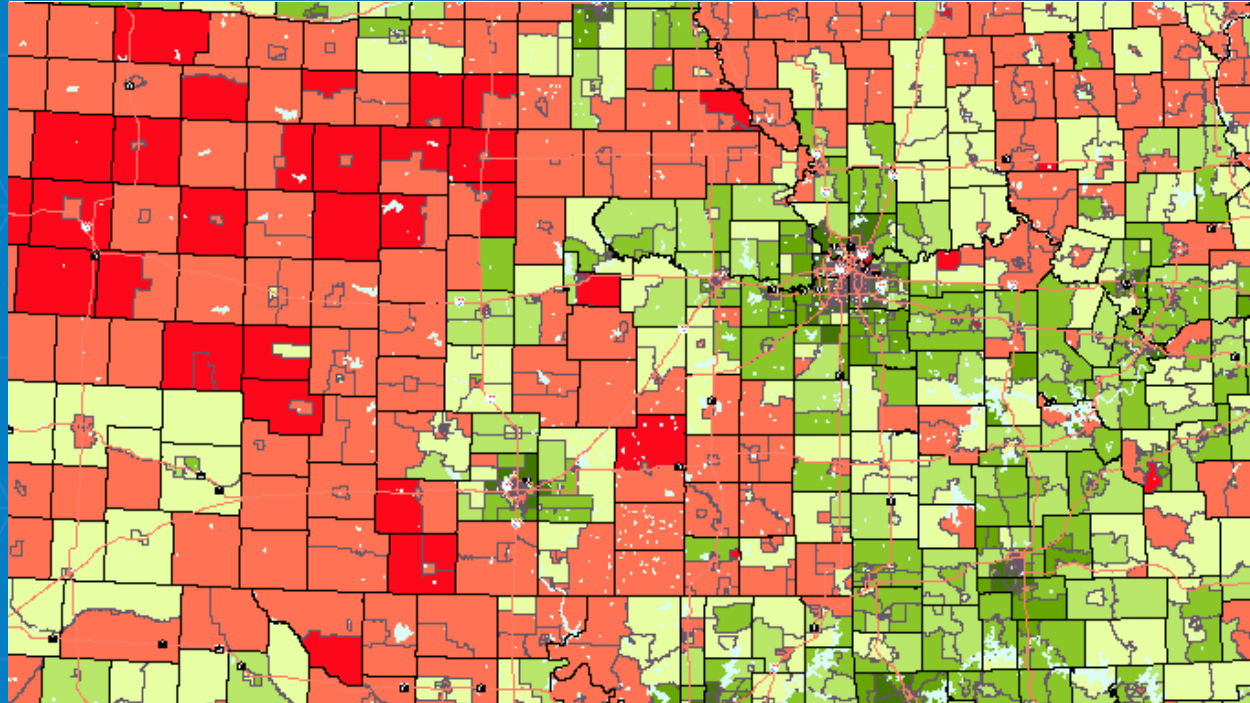
    # Create new points based on input lines
    with arcpy.da.SearchCursor(in_fc, ['SHAPE@', 'OID@']) as search_cursor:
        with arcpy.da.InsertCursor(out_fc, ['SHAPE@', fid_name]) as insert_cursor:
            for row in search_cursor:
                line = row[0]

                if line: # if null geometry--skip
                    if end_points:
                        insert_cursor.insertRow([line.firstPoint, row[1]])

                    cur_length = interval

                    max_position = 1
                    if not is_percentage:
                        max_position = line.length

                    while cur_length < max_position:
                        insert_cursor.insertRow(
```



Demo: Python toolboxes

Validation – Feature Layers

- The type of a layer parameter changes depending if the input is a layer or feature class

```
def updateParameters(self, parameters):  
    """Modify the values and properties of parameters before internal  
    validation is performed. This method is called whenever a parameter  
    has been changed."""  
  
    if parameters[0].altered:  
        if hasattr(parameters[0].value, 'value'): # Feature class  
            data = parameters[0].value.value  
  
        elif hasattr(parameters[0].value, 'dataSource'): # Layer  
            data = parameters[0].value.dataSource
```

Using parameter objects

- Improvement for 10.3.1 / Pro 1.1
- Describe accepts parameter objects directly
 - For layers, using parameter.value is expensive
 - Using parameter directly is much faster

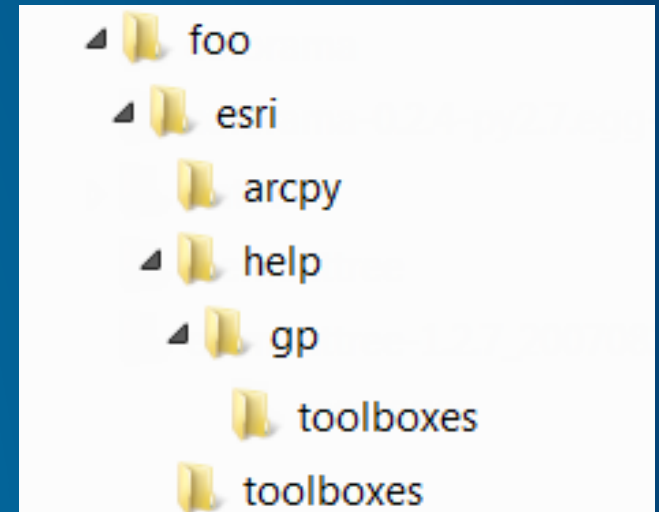
```
def updateParameters(self, parameters):  
    """Modify the values and properties of parameters before internal  
    validation is performed. This method is called whenever a parameter  
    has been changed."""  
  
    if parameters[0].altered and parameters[0].value:  
        shp_type = arcpy.Describe(parameters[0].value).shapeType
```

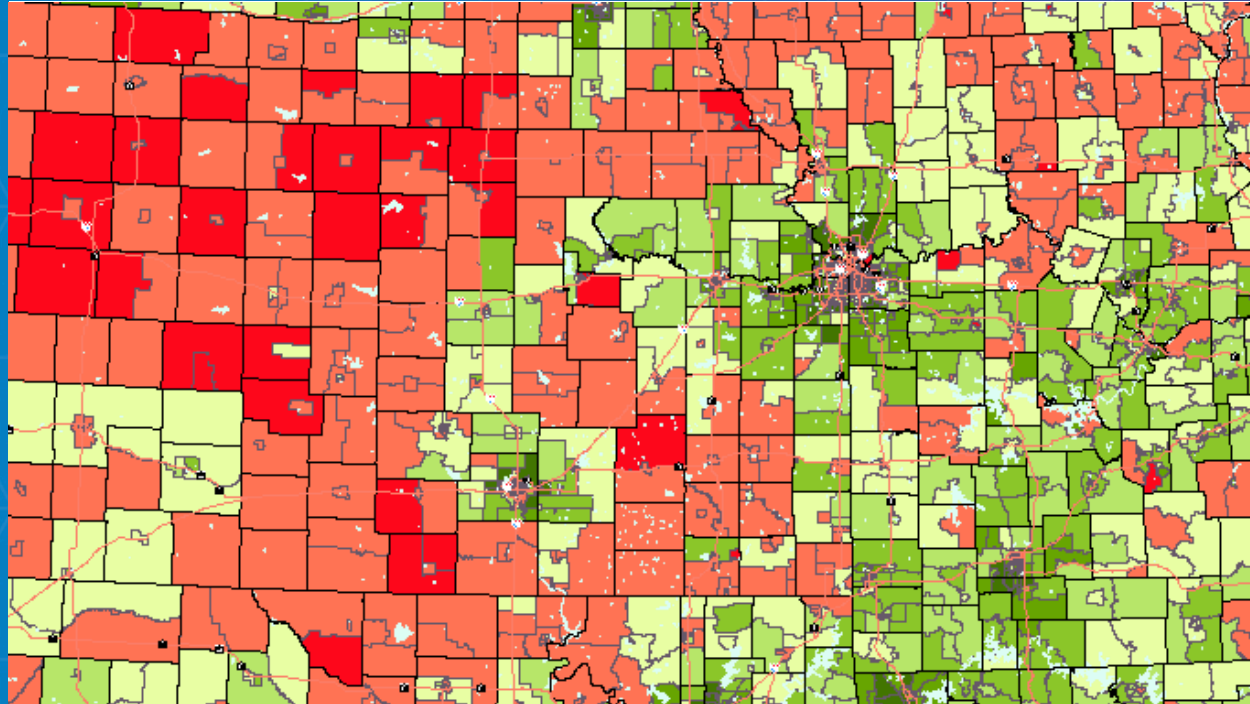
For 10.3.1 / Pro 1.1

```
def updateParameters(self, parameters):  
    """Modify the values and properties of parameters before internal  
    validation is performed. This method is called whenever a parameter  
    has been changed."""  
    param0 = parameters[0]  
    layer = param0.valueAsText  
    if not param0.altered and layer:  
        shp_type = arcpy.Describe(param0).shapeType
```

Distributing your tools in a Python module

- Create Python modules that play nice in ArcGIS
- Are easily distributable
- Toolboxes appear as System Toolboxes
- Toolboxes are incorporated into arcpy
 - Supports dialog side-panel help and localized messages
- In addition to site-package part, needs specific structure





Demo: Distributing tools

Rate This Session

www.esri.com/RateMyDevSummitSession