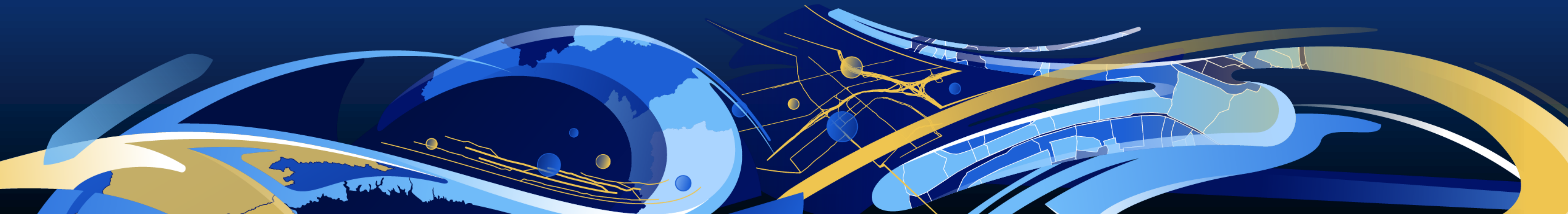
ArcPy: Building Geoprocessing Tools

Dave Wynne, Hannes Ziegler



ArcPy: Building Geoprocessing Tools

Dave Wynne,

- Documentation Lead

- Product Engineer

Hannes Ziegler,

- Product Engineer

Workshop description

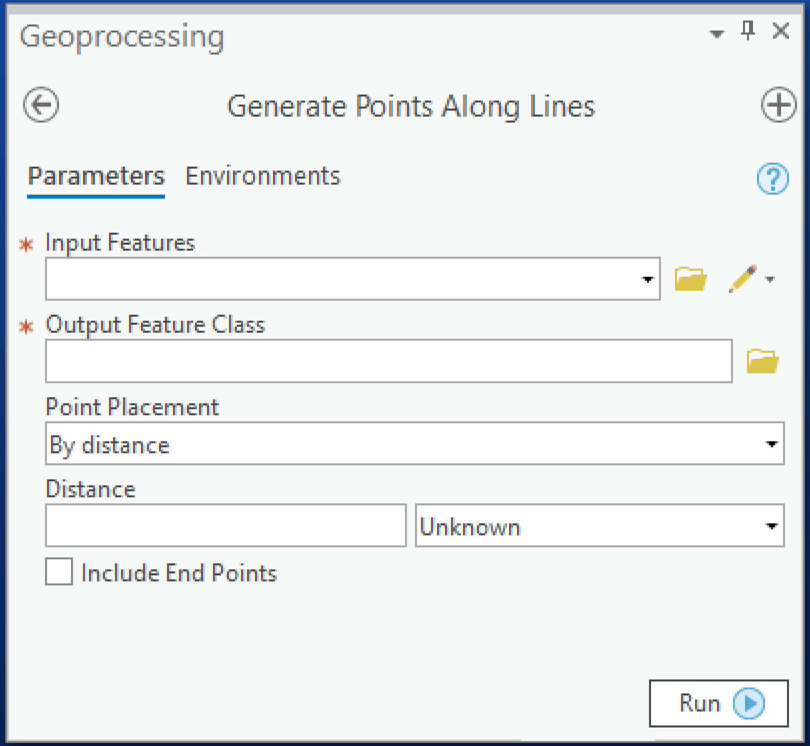
Being able to build a geoprocessing tool from Python is a fundamental building block for adding your own custom functionality into ArcGIS. Join us as we step through the process of taking your Python code and turning it into fully functional geoprocessing tools. Both script tools and Python toolboxes will be explored.

https://esriurl.com/uc20buildtools

- ☐ Tool basics
- ☐ Tool mechanics
- ☐ Design
- ☐ Script tools
- ☐ Python toolboxes
- ☐ Parameters
- ☐ Validation

Why we build geoprocessing tools

- Easy to access and run from within ArcGIS
- Familiar look and feel
- Use your tool in multiple ways
  - Python, ModelBuilder, a geoprocessing service
- Supported in multiple products



* Input Features

* Output Feature Class

Point Placement: By distance

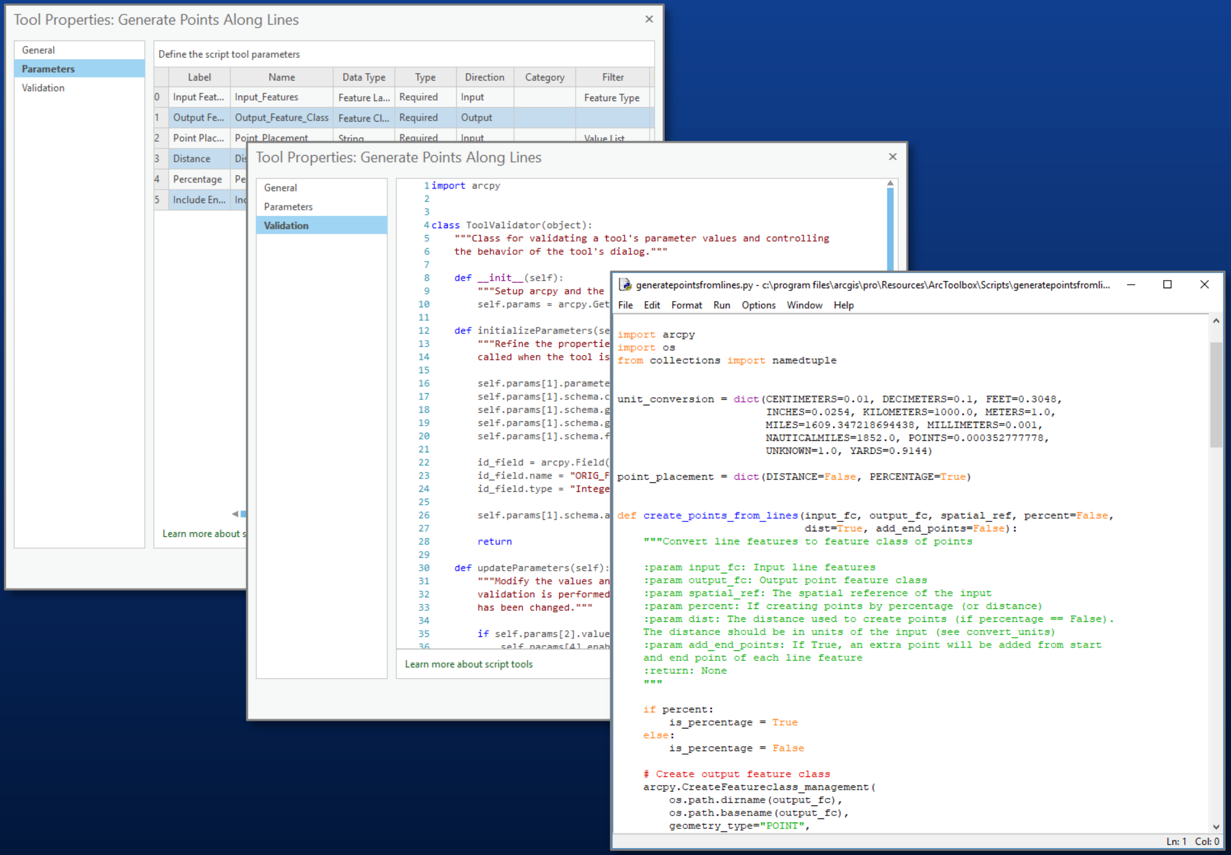
Distance: Unknown

☐ Include End Points

Tool structure

- A geoprocessing tool is made from 3 primary parts

1. Parameters
2. Validation
3. Execution code



Tool Properties: Generate Points Along Lines

Define the script tool parameters

	Label	Name	Data Type	Type	Direction	Category	Filter
0	Input Feat...	Input_Features	Feature La...	Required	Input		Feature Type
1	Output Fe...	Output_Feature_Class	Feature Cl...	Required	Output		
2	Point Plac...	Point_Placement	String	Required	Input		Value List
3	Distance	Di					
4	Percentage	Pe					
5	Include En...	Inc					

Learn more about s

Tool Properties: Generate Points Along Lines

```
import arcpy


class ToolValidator(object):
    """Class for validating a tool's parameter values and controlling
    the behavior of the tool's dialog."""

    def __init__(self):
        """Setup arcpy and the
        self.params = arcpy.Get

    def initializeParameters(se
        """Refine the propertie
        called when the tool is

        self.params[1].paramete
        self.params[1].schema.c
        self.params[1].schema.g
        self.params[1].schema.g
        self.params[1].schema.f

        id_field = arcpy.Field(
        id_field.name = "ORIG_F
        id_field.type = "Intege

        self.params[1].schema.a

        return

    def updateParameters(self):
        """Modify the values an
        validation is performed
        has been changed."""

        if self.params[2].value
```

Learn more about script tools

generatepointsfromlines.py - c:\program files\arcgis\pro\Resources\ArcToolbox\Scripts\generatepointsfromli...

```
import arcpy
import os
from collections import namedtuple


unit_conversion = dict(CENTIMETERS=0.01, DECIMETERS=0.1, FEET=0.3048,
                       INCHES=0.0254, KILOMETERS=1000.0, METERS=1.0,
                       MILES=1609.347218694438, MILLIMETERS=0.001,
                       NAUTICALMILES=1852.0, POINTS=0.000352777778,
                       UNKNOWN=1.0, YARDS=0.9144)

point_placement = dict(DISTANCE=False, PERCENTAGE=True)


def create_points_from_lines(input_fc, output_fc, spatial_ref, percent=False,
                             dist=True, add_end_points=False):
    """Convert line features to feature class of points

    :param input_fc: Input line features
    :param output_fc: Output point feature class
    :param spatial_ref: The spatial reference of the input
    :param percent: If creating points by percentage (or distance)
    :param dist: The distance used to create points (if percentage == False).
    The distance should be in units of the input (see convert_units)
    :param add_end_points: If True, an extra point will be added from start
    and end point of each line feature
    :return: None
    """

    if percent:
        is_percentage = True
    else:
        is_percentage = False

    # Create output feature class
    arcpy.CreateFeatureclass_management(
        os.path.dirname(output_fc),
        os.path.basename(output_fc),
        geometry_type="POINT",
```

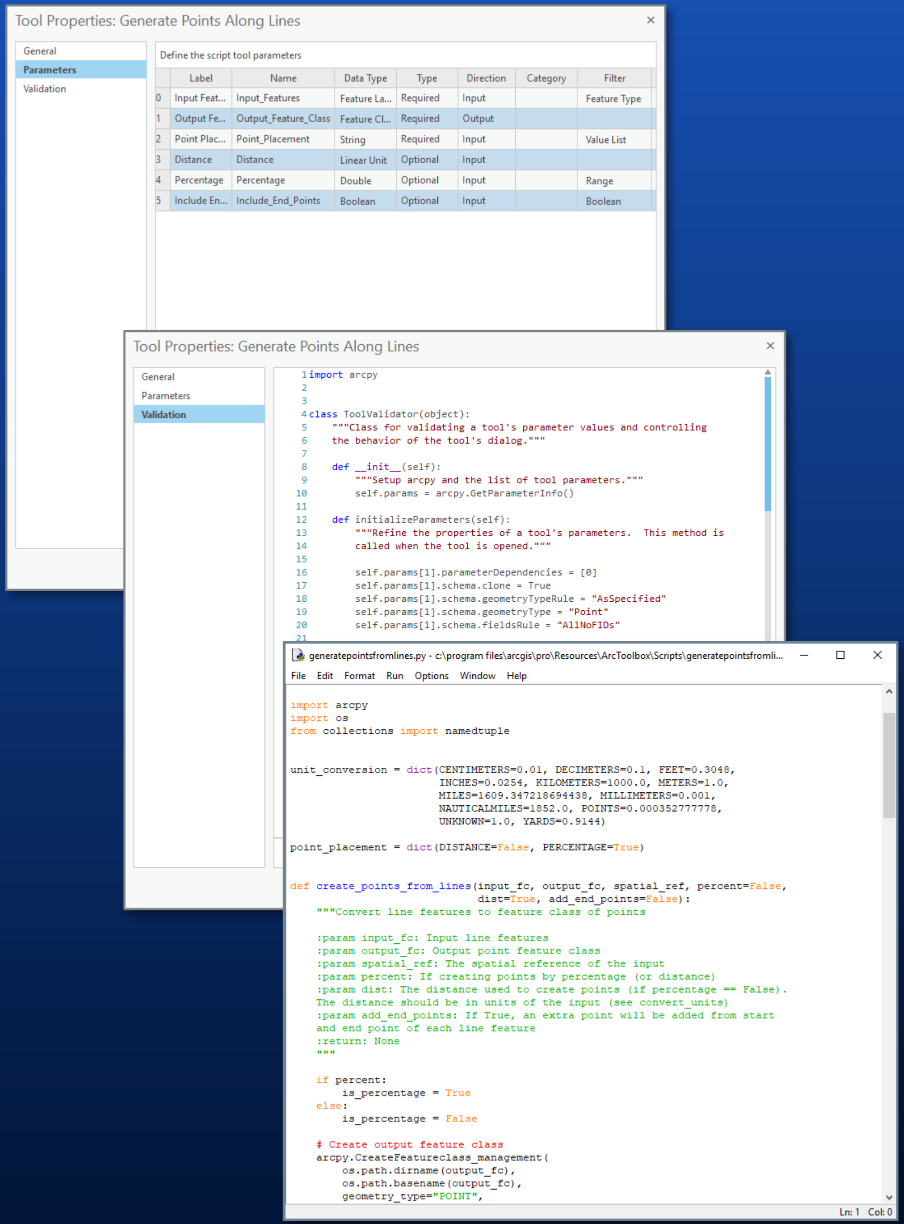
Ln: 1 Col: 0

Script tools vs Python toolboxes

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

1. Script tools

- Parameters defined through wizard
- Validation is Python
- Execution is Python



Tool Properties: Generate Points Along Lines

Define the script tool parameters

	Label	Name	Data Type	Type	Direction	Category	Filter
0	Input Feat...	Input_Features	Feature La...	Required	Input		Feature Type
1	Output Fe...	Output_Feature_Class	Feature Cl...	Required	Output		
2	Point Plac...	Point_Placement	String	Required	Input		Value List
3	Distance	Distance	Linear Unit	Optional	Input		
4	Percentage	Percentage	Double	Optional	Input		Range
5	Include En...	Include_End_Points	Boolean	Optional	Input		Boolean

Tool Properties: Generate Points Along Lines

```
import arcpy


class ToolValidator(object):
    """Class for validating a tool's parameter values and controlling
    the behavior of the tool's dialog."""

    def __init__(self):
        """Setup arcpy and the list of tool parameters."""
        self.params = arcpy.GetParameterInfo()

    def initializeParameters(self):
        """Refine the properties of a tool's parameters.  This method is
        called when the tool is opened."""

        self.params[1].parameterDependencies = [0]
        self.params[1].schema.clone = True
        self.params[1].schema.geometryTypeRule = "AsSpecified"
        self.params[1].schema.geometryType = "Point"
        self.params[1].schema.fieldsRule = "AllNoFIDs"
```

generatepointsfromlines.py - c:\program files\arcgis\pro\Resources\ArcToolbox\Scripts\generatepointsfromli...

```
import arcpy
import os
from collections import namedtuple


unit_conversion = dict(CENTIMETERS=0.01, DECIMETERS=0.1, FEET=0.3048,
                       INCHES=0.0254, KILOMETERS=1000.0, METERS=1.0,
                       MILES=1609.347218694438, MILLIMETERS=0.001,
                       NAUTICALMILES=1852.0, POINTS=0.000352777778,
                       UNKNOWN=1.0, YARDS=0.9144)

point_placement = dict(DISTANCE=False, PERCENTAGE=True)


def create_points_from_lines(input_fc, output_fc, spatial_ref, percent=False,
                             dist=True, add_end_points=False):
    """Convert line features to feature class of points

    :param input_fc: Input line features
    :param output_fc: Output point feature class
    :param spatial_ref: The spatial reference of the input
    :param percent: If creating points by percentage (or distance)
    :param dist: The distance used to create points (if percentage == False).
    The distance should be in units of the input (see convert_units)
    :param add_end_points: If True, an extra point will be added from start
    and end point of each line feature
    :return: None
    """

    if percent:
        is_percentage = True
    else:
        is_percentage = False

    # Create output feature class
    arcpy.CreateFeatureclass_management(
        os.path.dirname(output_fc),
        os.path.basename(output_fc),
        geometry_type="POINT",
```

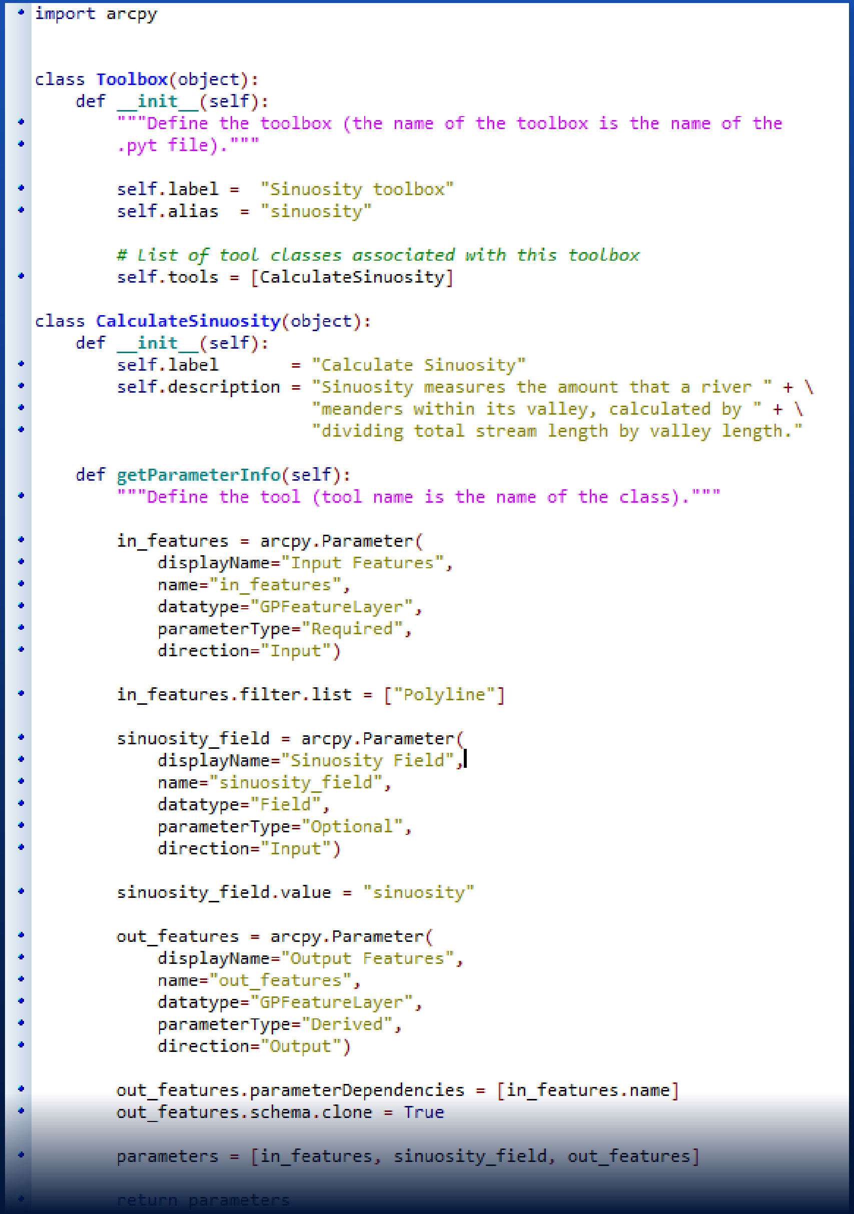
Script tools vs Python toolboxes

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

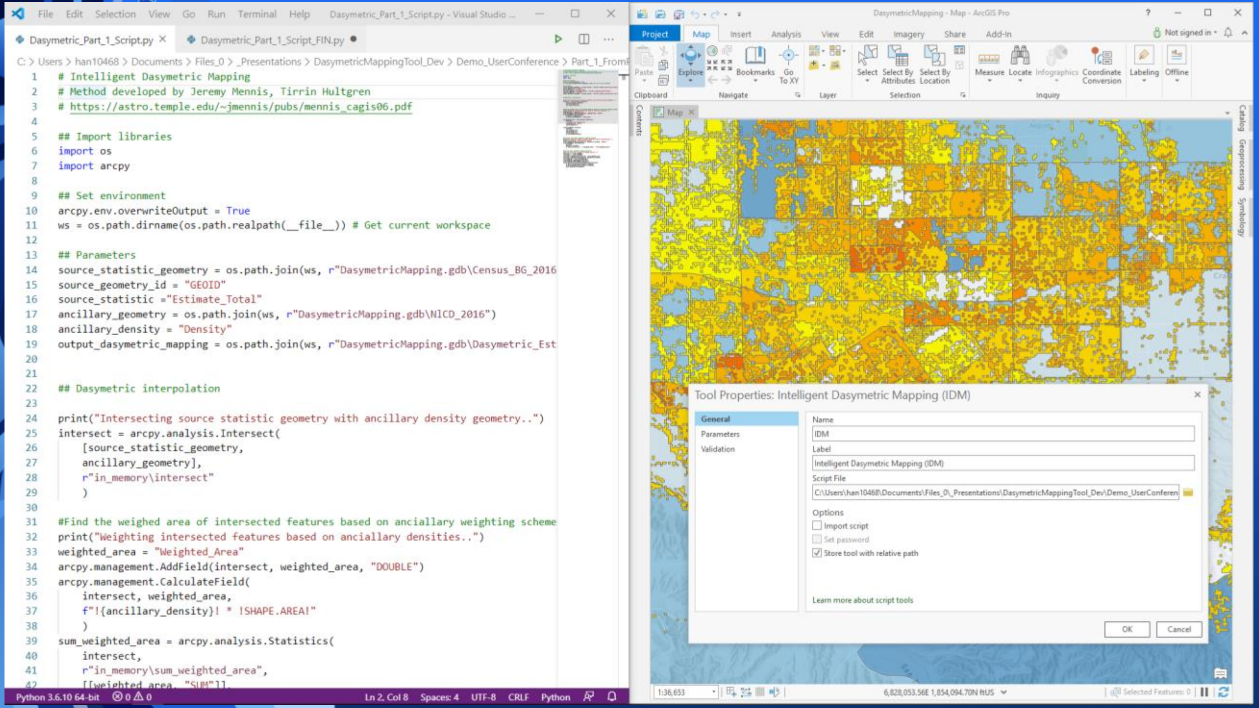
2. Python toolboxes

- Parameters are Python
- Validation is Python
- Execution is Python

- **Which do I use?**
  - **“A tool is a tool”**



```
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 =  "Sinuosity toolbox"
        self.alias  = "sinuosity"

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

class CalculateSinuosity(object):
    def __init__(self):
        self.label       = "Calculate Sinuosity"
        self.description = "Sinuosity measures the amount that a river " + \
                           "meanders within its valley, calculated by " + \
                           "dividing total stream length by valley length."

    def getParameterInfo(self):
        """Define the tool (tool name is the name of the class)."""

        in_features = arcpy.Parameter(
            displayName="Input Features",
            name="in_features",
            datatype="GPFeatureLayer",
            parameterType="Required",
            direction="Input")

        in_features.filter.list = ["Polyline"]

        sinuosity_field = arcpy.Parameter(
            displayName="Sinuosity Field",
            name="sinuosity_field",
            datatype="Field",
            parameterType="Optional",
            direction="Input")

        sinuosity_field.value = "sinuosity"

        out_features = arcpy.Parameter(
            displayName="Output Features",
            name="out_features",
            datatype="GPFeatureLayer",
            parameterType="Derived",
            direction="Output")

        out_features.parameterDependencies = [in_features.name]
        out_features.schema.clone = True

        parameters = [in_features, sinuosity_field, out_features]

        return parameters
```


File Edit Selection View Go Run Terminal Help Dasymetric_Part_1_Script.py - Visual Studio ...
Dasymetric_Part_1_Script.py
Dasymetric_Part_1_Script_FIN.py
C: > Users > han10468 > Documents > Files_0 > _Presentations > DasymetricMappingTool_Dev > Demo_UserConference > Part_1_FromP
Python 3.6.10 64-bit ⊗ 0 ⚠ 0 Ln 2, Col 8 Spaces: 4 UTF-8 CRLF Python
DasymetricMapping - Map - ArcGIS Pro
Project Map Insert Analysis View Edit Imagery Share Add-In
Not signed in
Paste Explore Bookmarks Go To XY Select Select By Attributes Select By Location Measure Locate Infographics Coordinate Conversion Labeling Offline
Clipboard Navigate Layer Selection Inquiry
Contents
Map
Catalog Geoprocessing Symbology
1:36,653
6,828,053.56E 1,854,094.70N ftUS
Selected Features: 0

```python
# Intelligent Dasymetric Mapping
# Method developed by Jeremy Mennis, Tirrin Hultgren
# https://astro.temple.edu/~jmennis/pubs/mennis_cagis06.pdf

## Import libraries
import os
import arcpy

## Set environment
arcpy.env.overwriteOutput = True
ws = os.path.dirname(os.path.realpath(__file__)) # Get current workspace

## Parameters
source_statistic_geometry = os.path.join(ws, r"DasymetricMapping.gdb\Census_BG_2016
source_geometry_id = "GEOID"
source_statistic ="Estimate_Total"
ancillary_geometry = os.path.join(ws, r"DasymetricMapping.gdb\NlCD_2016")
ancillary_density = "Density"
output_dasymetric_mapping = os.path.join(ws, r"DasymetricMapping.gdb\Dasymetric_Est


## Dasymetric interpolation

print("Intersecting source statistic geometry with ancillary density geometry..")
intersect = arcpy.analysis.Intersect(
    [source_statistic_geometry,
    ancillary_geometry],
    r"in_memory\intersect"
    )

#Find the weighed area of intersected features based on anciallary weighting scheme
print("Weighting intersected features based on anciallary densities..")
weighted_area = "Weighted_Area"
arcpy.management.AddField(intersect, weighted_area, "DOUBLE")
arcpy.management.CalculateField(
    intersect, weighted_area,
    f"!{ancillary_density}! * !SHAPE.AREA!"
    )
sum_weighted_area = arcpy.analysis.Statistics(
    intersect,
    r"in_memory\sum_weighted_area",
    [[weighted_area, "SUM"]],
```

Tool Properties: Intelligent Dasymetric Mapping (IDM)

General
Parameters
Validation

Name: IDM

Label: Intelligent Dasymetric Mapping (IDM)

Script File: C:\Users\han10468\Documents\Files_0_Presentations\DasymetricMappingTool_Dev\Demo_UserConferen

Options
- [ ] Import script
- [ ] Set password
- [✓] Store tool with relative path

Learn more about script tools

OK Cancel

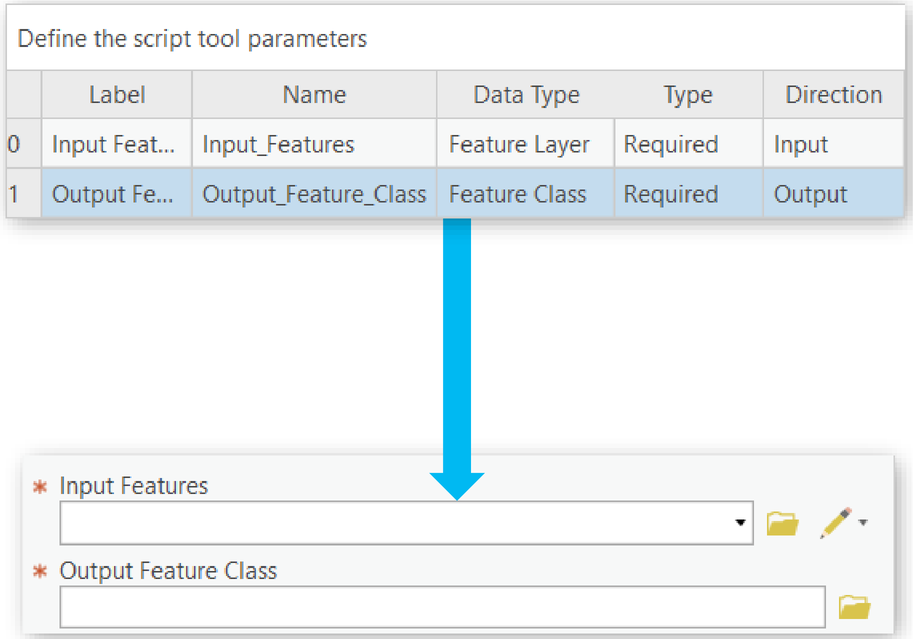
Demo: From Python to geoprocessing tool

Hannes Ziegler

How tools work

Dave Wynne

Tool parameters are initialized based on their definitions



Define the script tool parameters

	Label	Name	Data Type	Type	Direction
0	Input Feat...	Input_Features	Feature Layer	Required	Input
1	Output Fe...	Output_Feature_Class	Feature Class	Required	Output

Interaction

Every time a parameter value is modified, 3 methods are called:

1. **updateParameters** (method you write)
2. **Internal validation**
3. **updateMessages** (method you write)

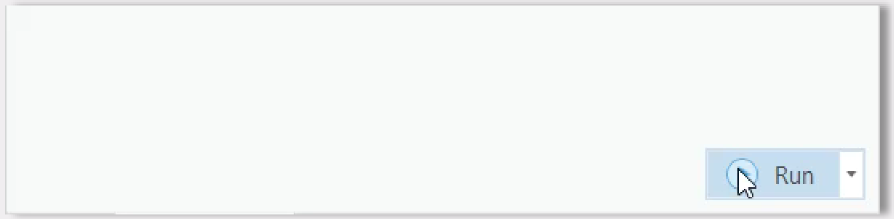
- **System checks, such as:**
  - **Have all the required parameters been supplied?**
  - **Are the values of the appropriate data types?**
  - **Does the input or output exist?**
  - **Do values match their filter?**

Run the tool

- Parameter values are sent
- Your execution code is called
- Script receives arguments

While the execute code is running, your code can communicate via:

- Messages
- A progressor



Outputs

- Every tool should have an output
- Set derived output values, if any
  - Use SetParameter/SetParameterAsText methods

finish

Wrap up

- Outputs are added to the map
- Symbology is applied, if any
- Result is added to the history

outputs

communicate

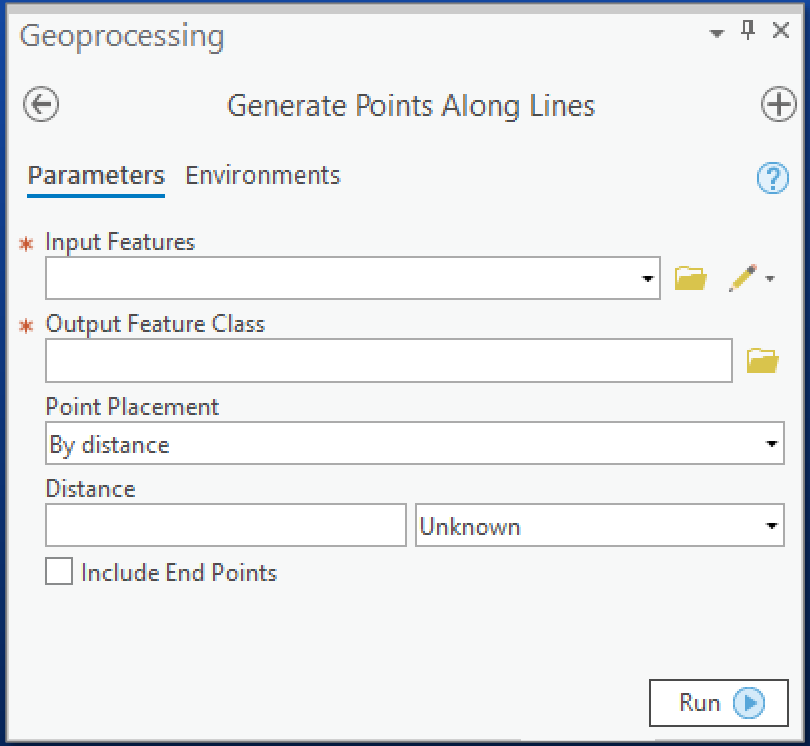
execute

validation

start

Parameters

- Parameters are how you interact with a tool
- Simple rules to guide behaviors
  - Does an input exist?
  - Is the input the right type?
  - What are valid fields for this data?
  - Is this value an expected keyword?



Input Features

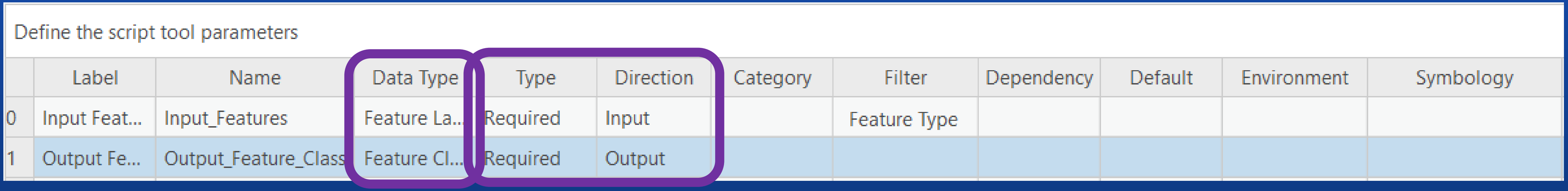
Output Feature Class

Point Placement: By distance

Distance: Unknown

☐ Include End Points

Parameter properties

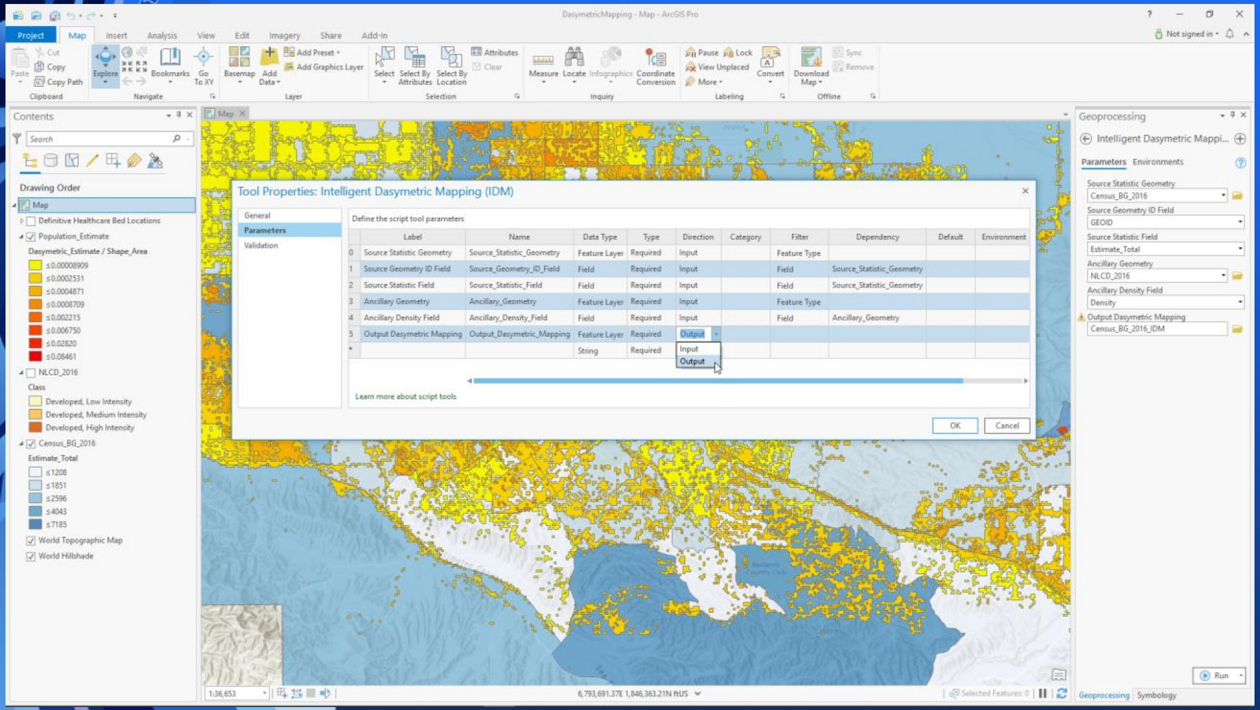

Define the script tool parameters

	Label	Name	Data Type	Type	Direction	Category	Filter	Dependency	Default	Environment	Symbology
0	Input Feat...	Input_Features	Feature La...	Required	Input		Feature Type				
1	Output Fe...	Output_Feature_Class	Feature Cl...	Required	Output						

- Data type
  - Feature Layer, Raster Layer, Table View, …
  - String, Boolean, Long, Float, …
- Parameter type
  - Required, Optional, Derived
- Direction
  - Input, Output

Demo: Working with tool parameters

Hannes Ziegler



Tool Properties: Intelligent Dasymetric Mapping (IDM)

General
Parameters
Validation

Define the script tool parameters

	Label	Name	Data Type	Type	Direction	Category	Filter	Dependency	Default	Environment
0	Source Statistic Geometry	Source_Statistic_Geometry	Feature Layer	Required	Input		Feature Type			
1	Source Geometry ID Field	Source_Geometry_ID_Field	Field	Required	Input		Field	Source_Statistic_Geometry		
2	Source Statistic Field	Source_Statistic_Field	Field	Required	Input		Field	Source_Statistic_Geometry		
3	Ancillary Geometry	Ancillary_Geometry	Feature Layer	Required	Input		Feature Type			
4	Ancillary Density Field	Ancillary_Density_Field	Field	Required	Input		Field	Ancillary_Geometry		
5	Output Dasymetric Mapping	Output_Dasymetric_Mapping	Feature Layer	Required	Output					
*			String	Required						

Learn more about script tools

OK Cancel

Intelligent Dasymetric Mappi...

Parameters Environments

Source Statistic Geometry: Census_BG_2016
Source Geometry ID Field: GEOID
Source Statistic Field: Estimate_Total
Ancillary Geometry: NLCD_2016
Ancillary Density Field: Density
Output Dasymetric Mapping: Census_BG_2016_IDM

Tool validation

Dave Wynne

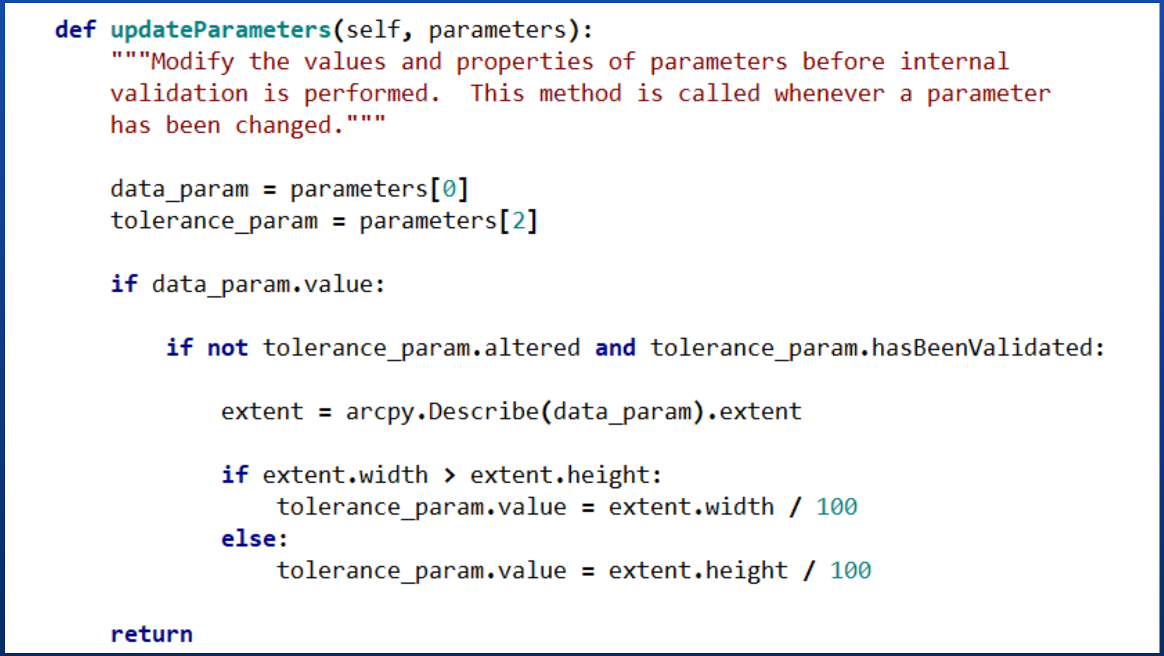
Validation

- Provides more control
  - Parameter interaction
  - Calculate defaults
  - Enable or disable parameters
- Setting parameter errors and messages
- Defining output characteristics
  - *Chain tools in ModelBuilder*

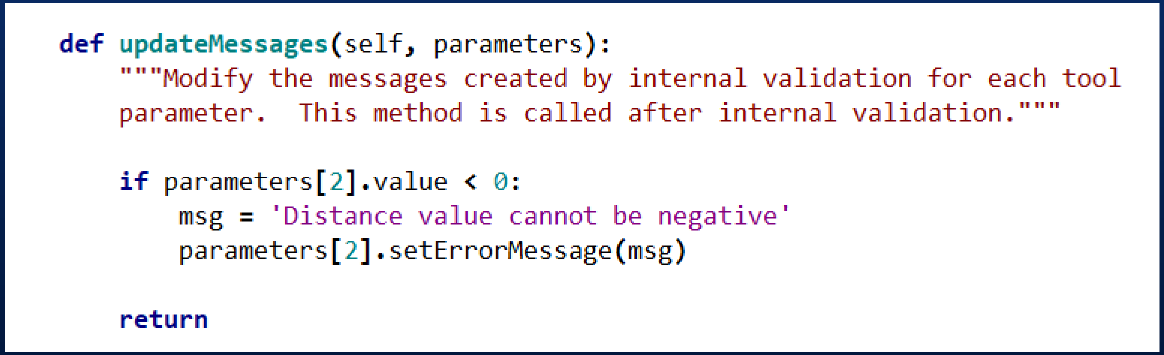
Validation

- **Mechanically, validation is about responding to changes in:**

- **value / valueAsText**
  - **Does a parameter have a value?**
  - **What is the value?**
  - **Properties of the data (arcpy.Describe)**
- **altered**
  - **Has the parameter been altered?**
- **hasBeenValidated**
  - **Has internal validation checked the parameter?**



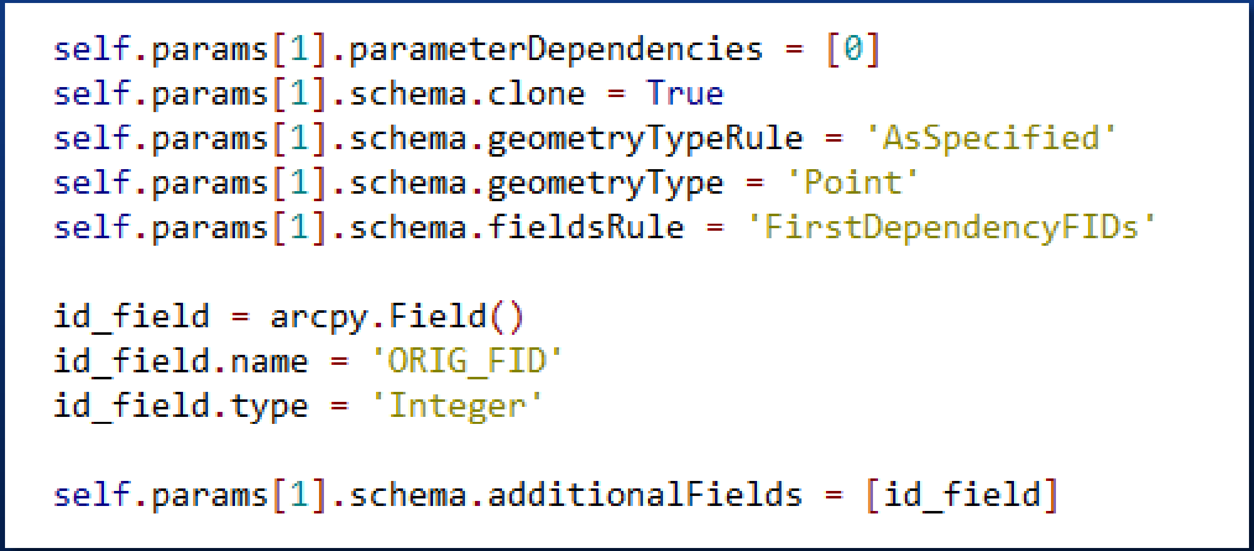
```python
def updateParameters(self, parameters):
    """Modify the values and properties of parameters before internal
    validation is performed.  This method is called whenever a parameter
    has been changed."""

    data_param = parameters[0]
    tolerance_param = parameters[2]

    if data_param.value:

        if not tolerance_param.altered and tolerance_param.hasBeenValidated:

            extent = arcpy.Describe(data_param).extent

            if extent.width > extent.height:
                tolerance_param.value = extent.width / 100
            else:
                tolerance_param.value = extent.height / 100

    return
```



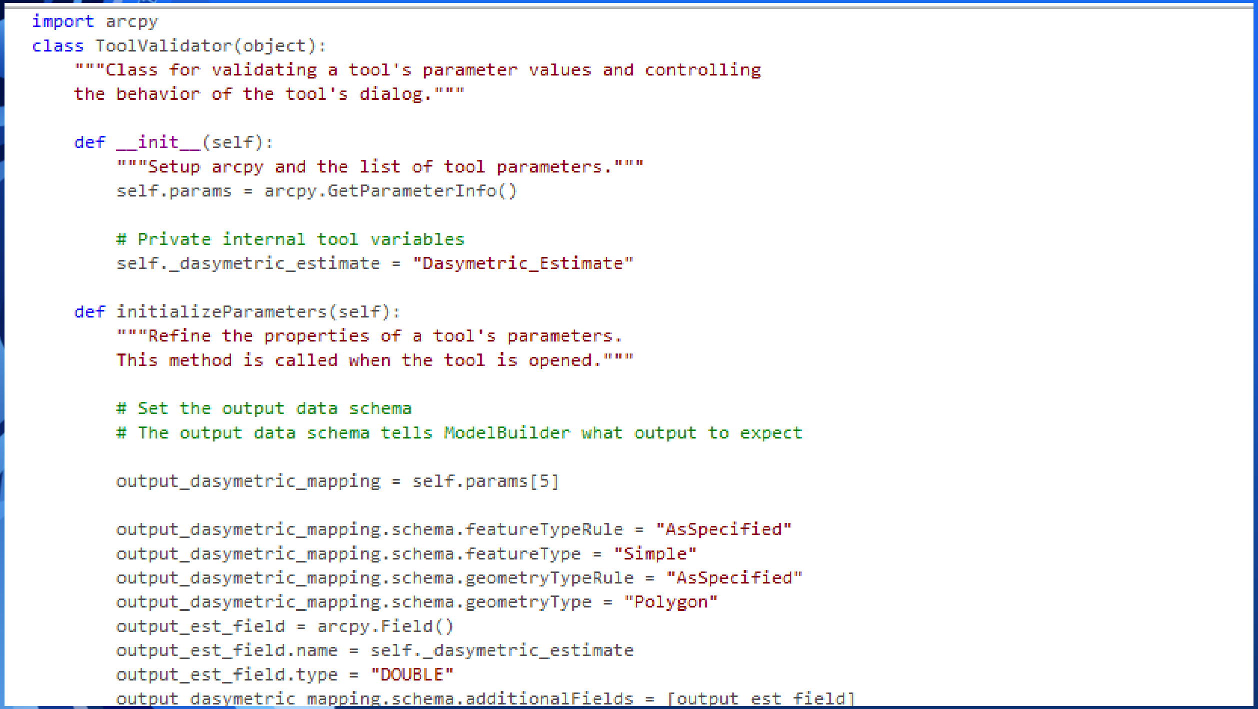
```python
def updateMessages(self, parameters):
    """Modify the messages created by internal validation for each tool
    parameter.  This method is called after internal validation."""

    if parameters[2].value < 0:
        msg = 'Distance value cannot be negative'
        parameters[2].setErrorMessage(msg)

    return
```

Validation: ModelBuilder

- Describe outputs for chaining
- By updating output `schema`, subsequent tools can see pending changes prior to execution



```python
self.params[1].parameterDependencies = [0]
self.params[1].schema.clone = True
self.params[1].schema.geometryTypeRule = 'AsSpecified'
self.params[1].schema.geometryType = 'Point'
self.params[1].schema.fieldsRule = 'FirstDependencyFIDs'

id_field = arcpy.Field()
id_field.name = 'ORIG_FID'
id_field.type = 'Integer'

self.params[1].schema.additionalFields = [id_field]
```



```
import arcpy
class ToolValidator(object):
    """Class for validating a tool's parameter values and controlling
    the behavior of the tool's dialog."""

    def __init__(self):
        """Setup arcpy and the list of tool parameters."""
        self.params = arcpy.GetParameterInfo()

        # Private internal tool variables
        self._dasymetric_estimate = "Dasymetric_Estimate"

    def initializeParameters(self):
        """Refine the properties of a tool's parameters.
        This method is called when the tool is opened."""

        # Set the output data schema
        # The output data schema tells ModelBuilder what output to expect

        output_dasymetric_mapping = self.params[5]

        output_dasymetric_mapping.schema.featureTypeRule = "AsSpecified"
        output_dasymetric_mapping.schema.featureType = "Simple"
        output_dasymetric_mapping.schema.geometryTypeRule = "AsSpecified"
        output_dasymetric_mapping.schema.geometryType = "Polygon"
        output_est_field = arcpy.Field()
        output_est_field.name = self._dasymetric_estimate
        output_est_field.type = "DOUBLE"
        output_dasymetric_mapping.schema.additionalFields = [output_est_field]
```

Demo:
Tool validation

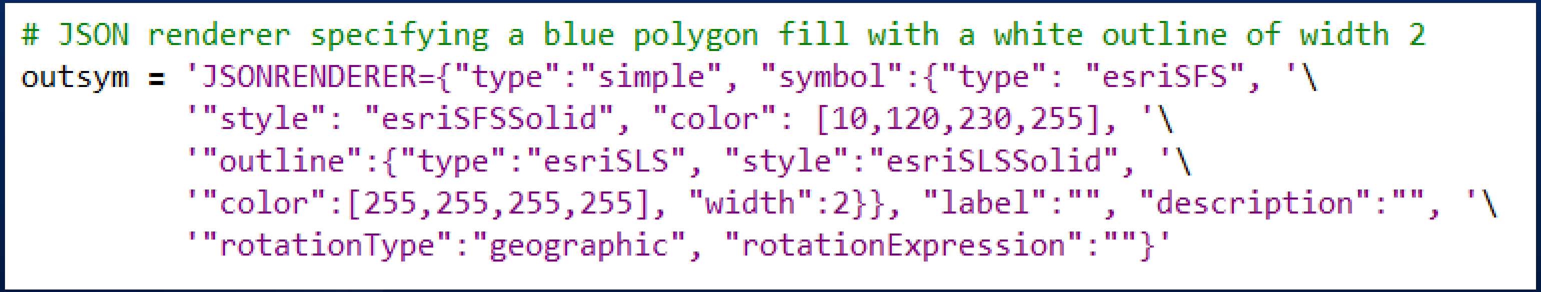
Hannes Ziegler

Symbolize outputs

- For tool output, can set symbology via:

1. Layer files

2. arcpy.SetParameterSymbology
   - New at ArcGIS Pro 2.5
   - Can set symbology using a JSON renderer object



```
# JSON renderer specifying a blue polygon fill with a white outline of width 2
outsym = 'JSONRENDERER={"type":"simple", "symbol":{"type": "esriSFS", '\
         '"style": "esriSFSSolid", "color": [10,120,230,255], '\
         '"outline":{"type":"esriSLS", "style":"esriSLSSolid", '\
         '"color":[255,255,255,255], "width":2}}, "label":"", "description":"", '\
         '"rotationType":"geographic", "rotationExpression":""}'
```

- https://pro.arcgis.com/en/pro-app/arcpy/functions/setparametersymbology.htm

THE SCIENCE OF WHERE®