Python: Building Geoprocessing Tools

David Wynne, Ghislain Prince

Python: Building Geoprocessing tools

Intermediate

Tuesday, 21 Jul 2015, 1:30pm - 2:45pm
Location: Ballroom 06 F

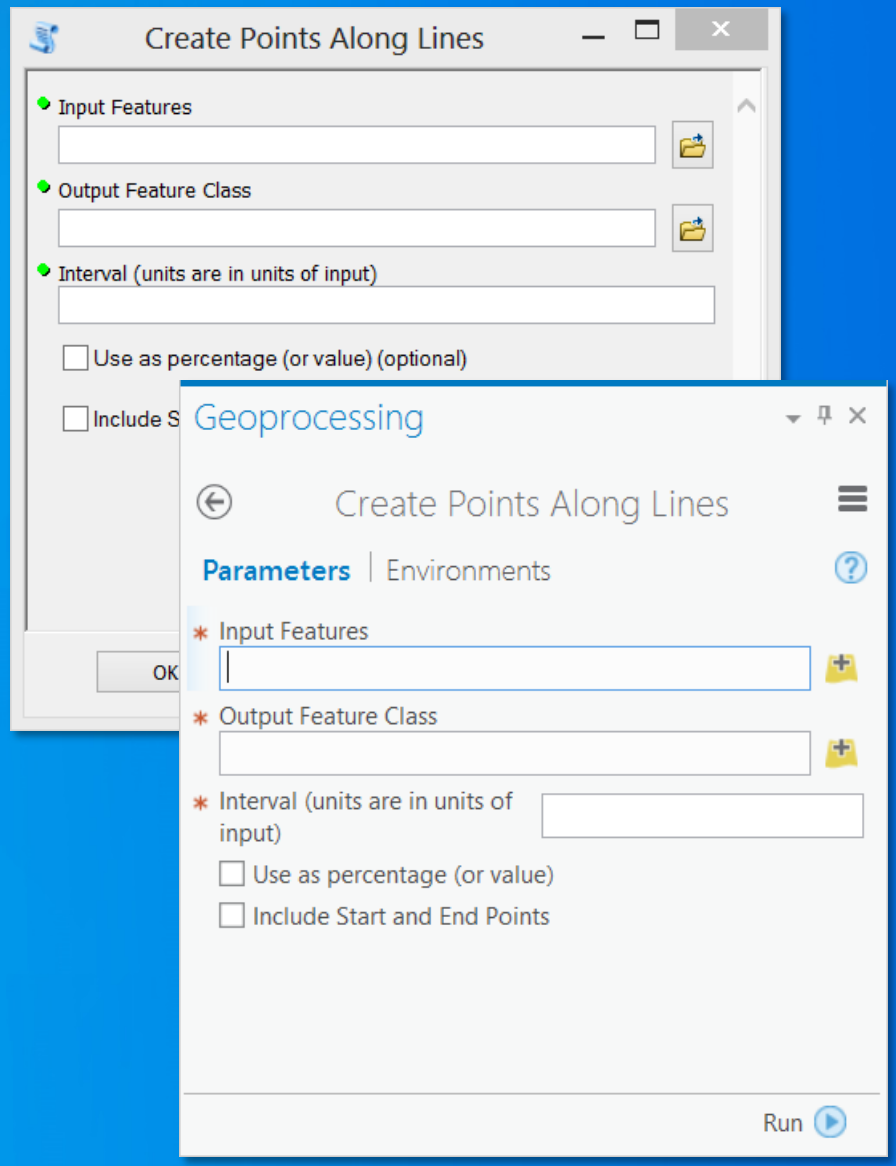
Thursday, 23 Jul 2015, 1:30pm - 2:45pm
Location: Ballroom 06 B

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.

Why we build geoprocessing tools

- Your work becomes part of the geoprocessing framework
  - Easy to access and run from within ArcGIS
  - Familiar look and feel
  - Make a mistake? Re-run from the previous result
  - Run from Python, ModelBuilder, as a service
  - Add to a tool bar

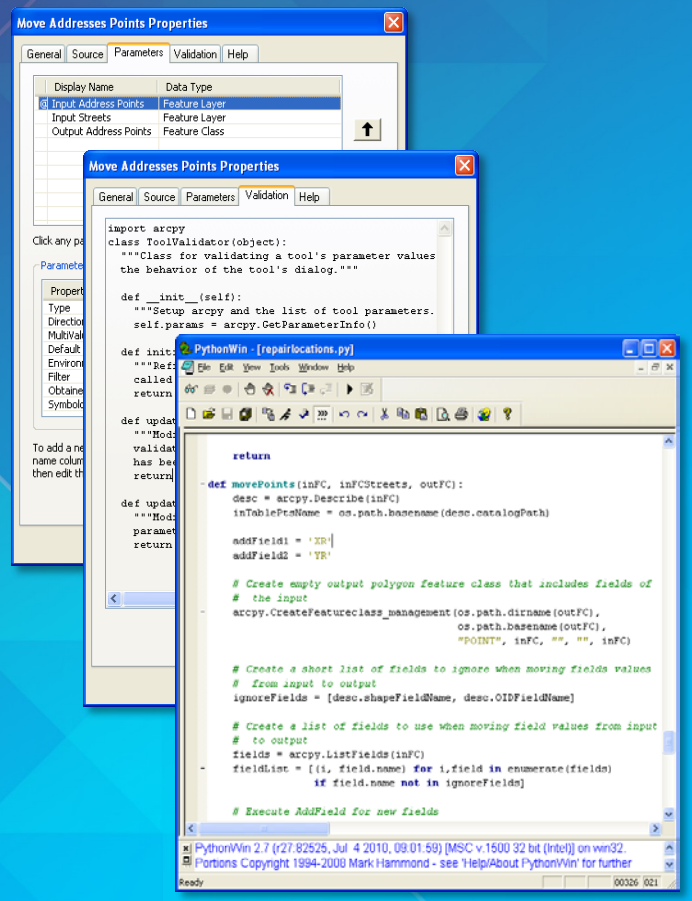
- No UI programming



What makes a good geoprocessing tool?

- Performs an essential and elemental operations on data
  - Can be combined with other tools to perform larger processes
  - Simple; isn't overcomplicated
- Looks and feels like other geoprocessing tools ('conventions')
  - Such as has an output ('required' or 'derived')
  - Parameter type: Required → optional → derived
  - Parameter direction: Inputs → outputs
- Includes documentation

Deconstructing a geoprocessing tool



- **A geoprocessing tool does three things**
  1. **Defines parameters**
  2. **Validates parameters**
  3. **Executes source**

```python
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
```

Turning Python code into geoprocessing tools

Demo

Script tools vs Python toolboxes

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

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

- **Which do I use?**
  - **“A tool is a tool”**
  - **From a development perspective, Python toolboxes are easier to manage/code***

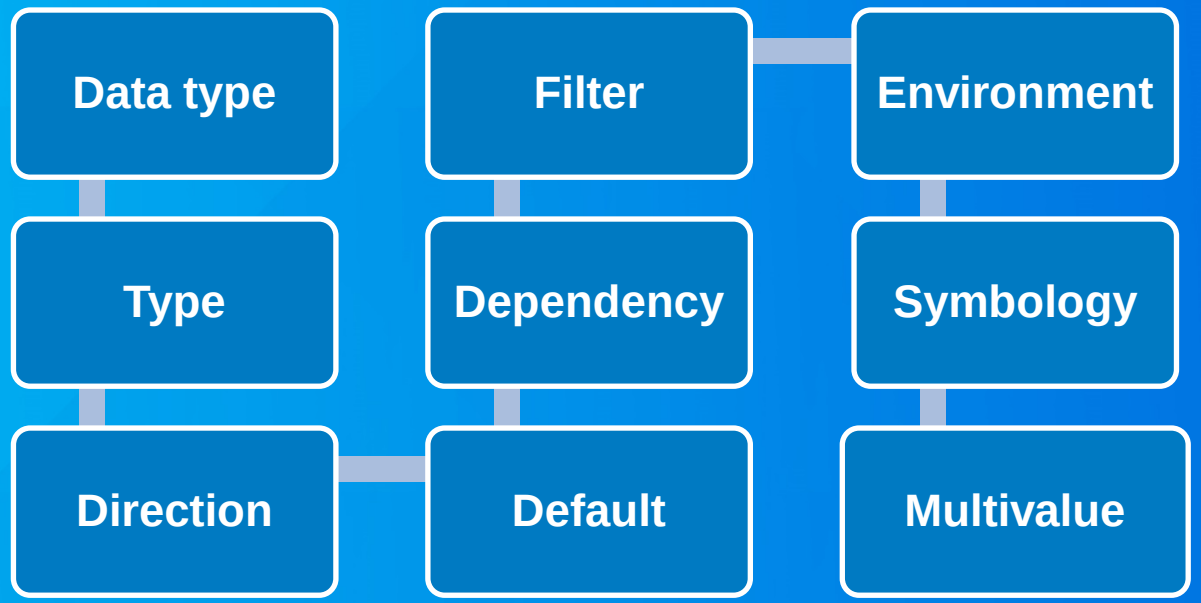
****Says Dave (use what suits you)***

Parameters

- A geoprocessing tool does three types of work:

1. Defines its parameters
2. Validates its parameters
3. Executes code that performs the actual work

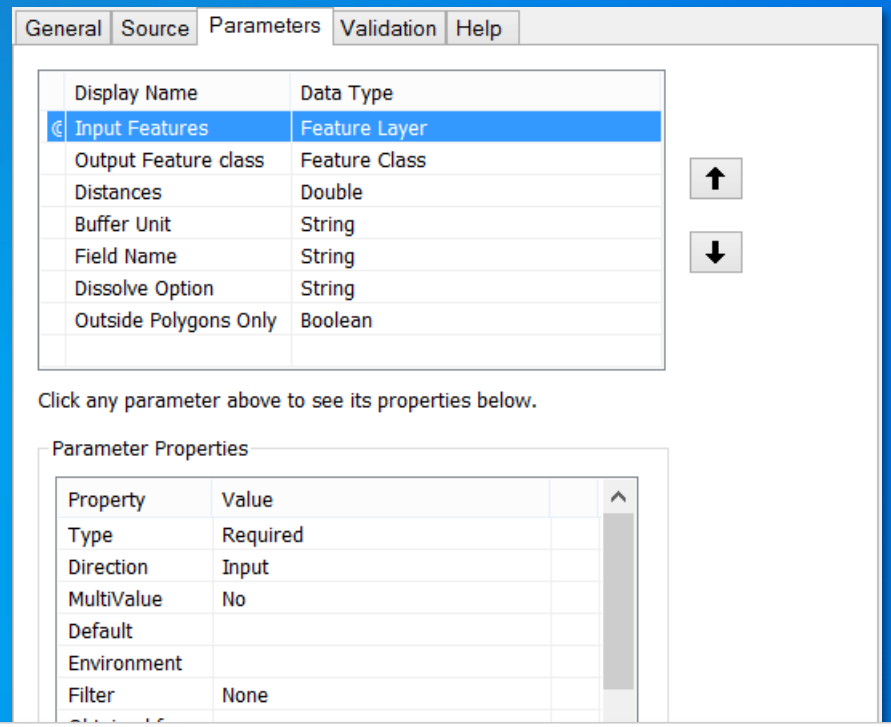
- Parameters are how you interact with a tool
- Simple rules to guide behaviors
  - Does an input exist?
  - What are valid fields for this data source?
  - Is this value an expected keyword?

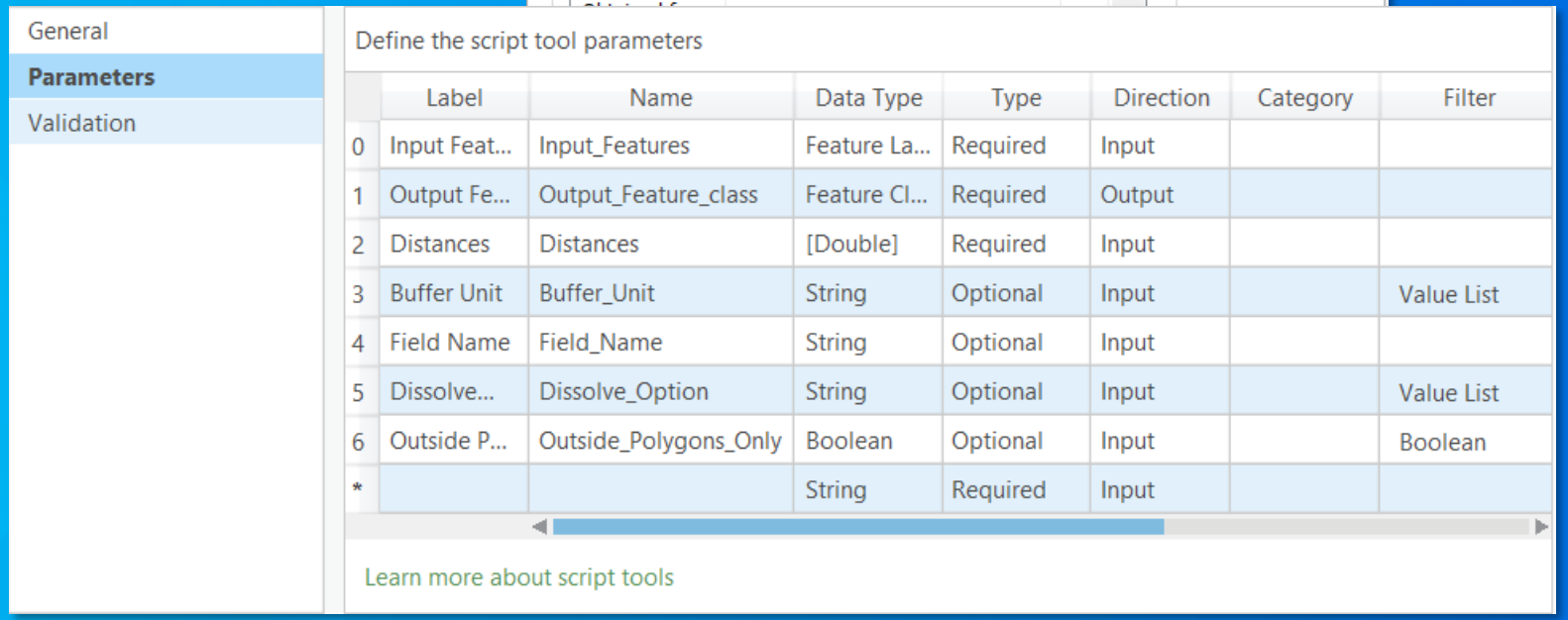


Parameters

- **Key parameter characteristics**

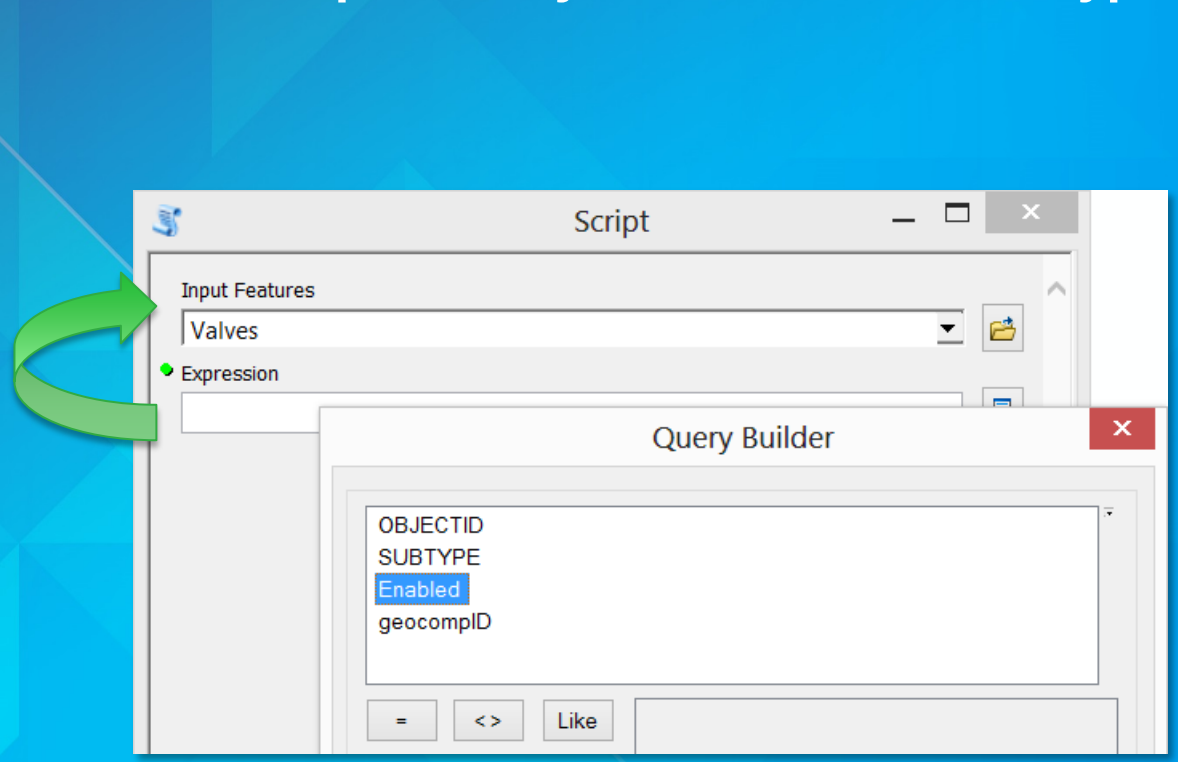
1. **Data type**
   - Feature layers, raster layers, table views, …
   - String, Boolean, long, double, …
2. **Direction**
   - Input, output
3. **Parameter type**
   - Required, optional, derived

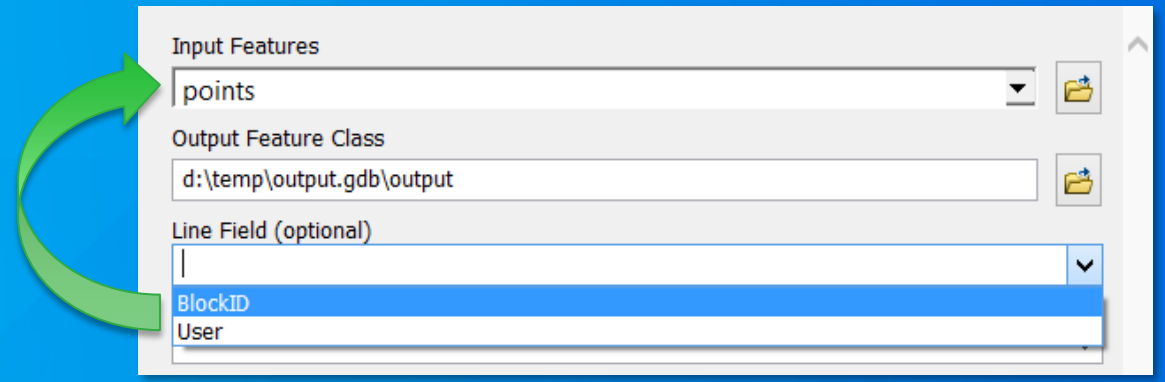




Parameters – Parameter Dependencies (‘*obtained from*’)

- **Some data types are dependent on other parameters**
  - **Field, SQL Expression, Field Info provide little value on their own**
- **Set a dependency to a different data type**

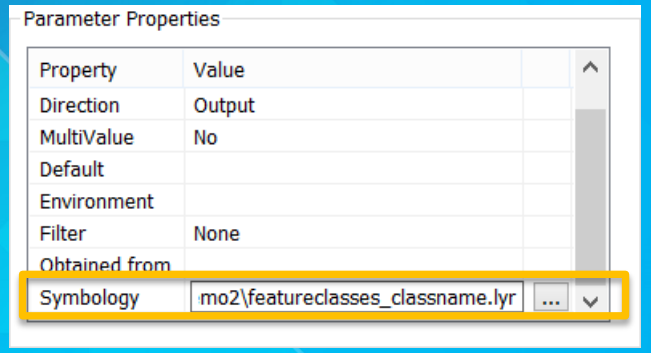




Parameters - Symbology

- Control the appearance of output by setting symbology
- Set using a layer file either a :

1. Parameter property



2. Or in your source code
   - Important if you want to vary the symbology based on logical criteria

```
# Set the symbology
params = arcpy.GetParameterInfo()
if len(params) > 0:
    params[1].symbology = os.path.join(layerfile_folder,
                                       'poly_symbology.lyr')
```

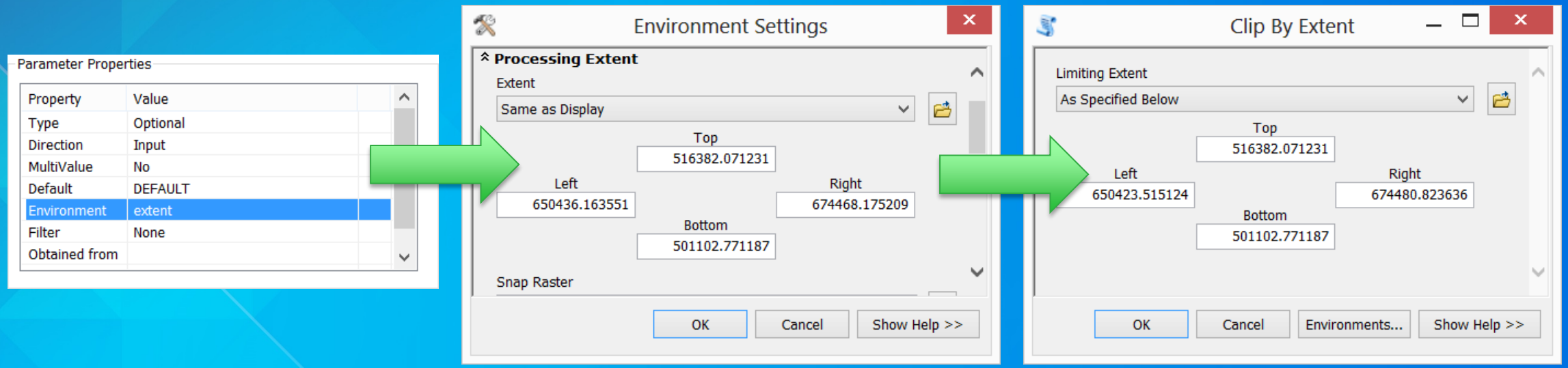
Parameters - Filters

- **Filters are used to limit choices to acceptable values**
- **For example:**
  - **A number between 1 and 10**
  - **A string parameter with keywords**
  - **A file parameter that will only accept files with a .txt extension**

Filter	Values	Relevant data types
Value List	String (**keywords**) or **numeric** values	String, Long, Double, Boolean
Range	Between a **minimum** and **maximum** value	Long, Double
Feature Class	Allowable feature class types	Feature Layer, Feature Class
File	File extensions	File
Field	Supported field types	Field
Workspace	Supported workspaces	Workspace

Parameters - Environments

- Some parameter data types match closely with geoprocessing environments
- Can configure a parameter to use an environment (if set) by default



Parameter-izing a tool

Demo

Validation

- A geoprocessing tool does three types of work:

1. Defines its parameters
2. Validates its parameters
3. Executes code that performs the actual work

- Provide additional behavior for how parameters respond and interact to values and each other
- With Python code that you write
  - Control parameter interaction
  - Calculate defaults
  - Enable or disable parameters
  - Messaging (to guide parameter choices and protect against invalid ones)

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 r

    # If using percentages, distance must be less
    elif parameters[3].value:
        if parameters[2].value > 1.0:
            parameters[2].setErrorMessage(
                'Percentages must be between 0.0 a
    return
```

```
def updateMessage(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
```

Validation

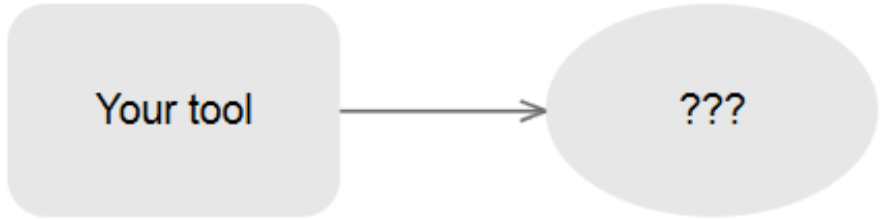
- Parameter definitions provide common ‘validation’ requirements
- For additional control, use updateParameters and updateMessages
- Validation is run every time a value changes
- Validation logic is usually based on:
  - value (or valueAsText)
  - altered
  - hasBeenValidated

```
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

- Messages are added in updateMessages
- A tool will not run while a parameter has an active error message
- setErrorMessage(<message>)
- setWarningMessage(<message>)

```
def updateMessages(self, parameters):
    """Modify the messages created by internal validation for
    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
```

Validation: Licensing

- Check if a tool is licensed to execute*
- If isLicensed returns False, the tool is blocked from executing 
- ** 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
```

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 parame
    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

- New at 10.3.1 and Pro 1.1
- Describe accepts parameter objects directly
  - For layers, using parameter.value is expensive
  - Using parameter directly is much faster (more noticeably in Pro)

```
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
```

Working with validation: Script tools and Python toolboxes

Demo

Execution

- A geoprocessing tool does three types of work:

1. Defines its parameters
2. Validates its parameters
3. Executes code that performs the actual work

- The ‘business logic’ of your tool

A. Parameters are received by your code
   - parameter.value or parameter.valueAsText
   - arcpy.GetParameter(n) or arcpy.GetParameterAsText(n)

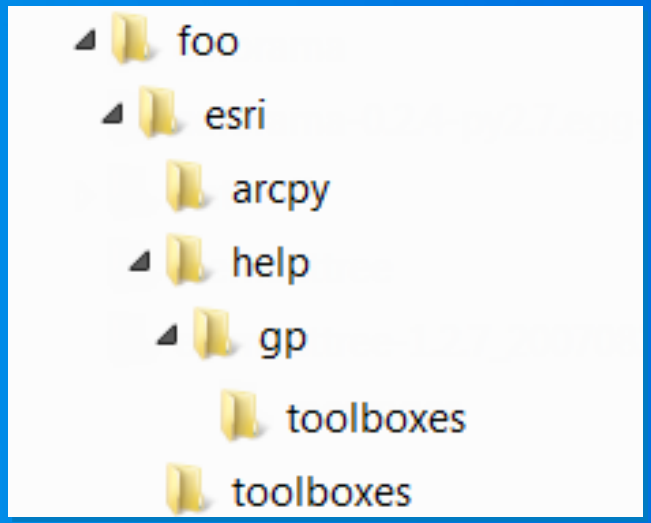
B. The logic of your tool is performed
   - Communication can be added with messages and progressor

C. Derived output values (if any) are pushed back

Distributing your tools

http://esriurl.com/extendingGP

- Easy to ‘share’ a toolbox.
  - Simply share the files or package a result
- But! To have your toolboxes blend seamlessly, distribute as a Python site-package
  - Are easily distributable
  - Toolboxes appear as System Toolboxes
  - Toolboxes become available through arcpy
  - Supports dialog side-panel help and localized messages
- Requires specific structure



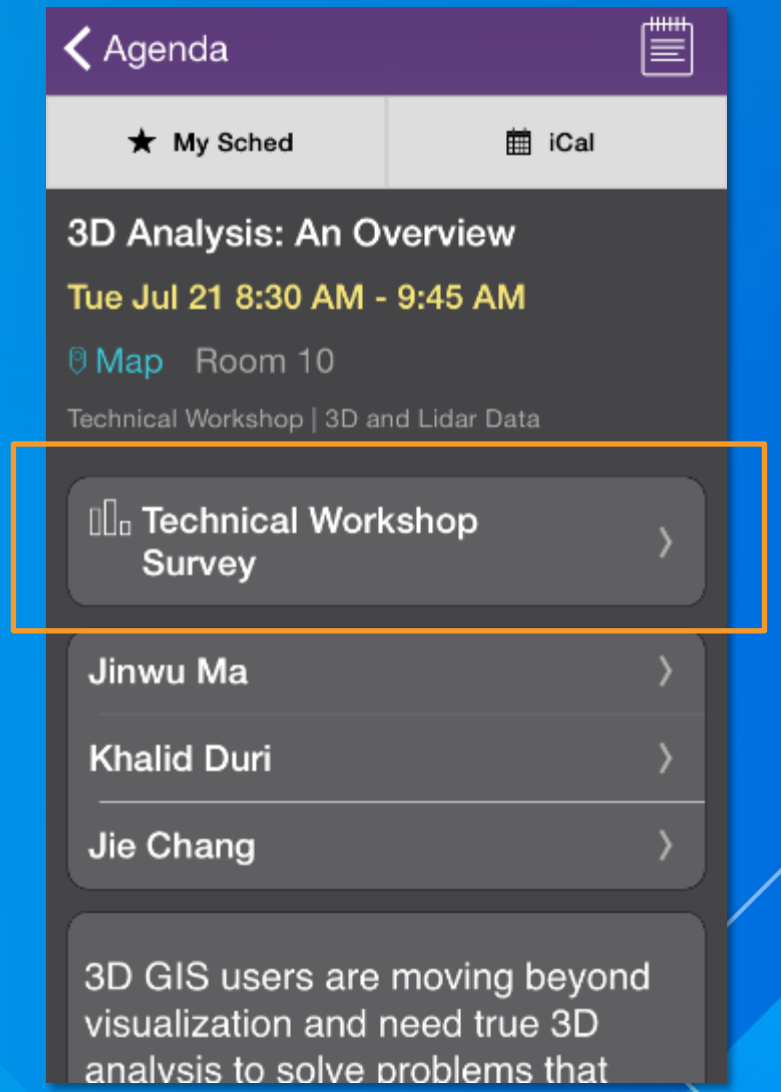
Python at User Conference

- **http://esriurl.com/UC2015Python (54 Python sessions)**

	Tuesday	Wednesday	Thursday
8:30	• Python: An Introduction	• Python Map Automation: Introduction to…	• Python: An Introduction • Python: Raster Analysis
10:15	• Python: Beyond the Basics		• Python: Beyond the Basics • Python Map Automation: Introduction to …
11:30		• ArcGIS Pro: A Quick Tour of Python	
1:30	• Python: Building Geoprocessing tools		• Python: Building Geoprocessing tools • Advanced Map Automation with Python
3:15	• Python: Raster Analysis	• Advanced Map Automation with Python	

Thank you…

- Please fill out the session survey in your mobile app
- Select Python: Building Geoprocessing Tools in the Mobile App
  - Use the Search Feature to quickly find this title
- Click “Technical Workshop Survey”
- Answer a few short questions and enter any comments



Understanding our world.