Developing Geoprocessing Tools in a Python Toolbox

Dave Wynne
Dale Honeycutt
Abstract

Join us as we step through the entire process of creating tools in a Python toolbox and highlight the important decisions in making a fully functional geoprocessing tool. Find out about additional capabilities not supported by the Script tool wizard, such as creating value table parameters, composite data types, and tool licensing.
Questions for you

• First Dev Summit?
• How much general geoprocessing experience?
• How much Python experience?
• Develop your own script tools?
3 types of geoprocessing tools

- “Built-in” tools
  - Developed with a compiled language and ArcObjects
- ModelBuilder tools
- Script tools
  - (Includes Python toolbox)
2 ways to deliver script tools

• In a custom toolbox
  - Right-click toolbox and choose “Add > Script”
  - Opens the “Add Script Tool Wizard”

• In a Python toolbox
  - Right-click a folder and choose “New > Python Toolbox”
  - Introduced at 10.1

• Which delivery mechanism do you choose?
  - We’ll get to that
Today’s agenda

• Introduce basic concepts of geoprocessing tools using the “Add Script tool wizard”
  - It’s just easier and faster and may be more familiar to you

• Once basics are covered, show how a tool is implemented in a Python toolbox
A geoprocessing tool does 3 things:

- Defines its parameters
- Validates its parameters (before it executes)
- Executes some code that performs the actual work
READ THE DOC

• Everything we cover today (and much more) is available in the documentation

• Today => highlight the important stuff, give you enough information to be dangerous
1 Defining parameters

Demo: Adding a script tool
Demo review – Table to html

• **GetParameterAsText(index)**
  - How parameters are received by the script

• **Add Script Tool Wizard**
  - Properties can be updated later using Properties dialog

• **Parameters**
  - Label
  - Data Type
  - Direction (input, output, derived)
  - Required vs. Optional
  - Filters
Demo – Table to html with field choices

- **Field data type** – setting *Obtained From*
- **MultiValues** are passed to script as a semi-colon delimited string
  - “aaa;bbb;ccc;ddd”
- **Transform to python list with:**
  - `mylist = string.split(";")`
  - `Fields = arcpy.GetParameterAsText(1).split(";")`
Parameter filters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value List</td>
<td>A list of string or numeric values. Used with String, Long, Double, and Boolean parameter data types.</td>
</tr>
<tr>
<td>Range</td>
<td>A minimum and maximum value. Used with Long and Double data types.</td>
</tr>
<tr>
<td>Feature Class</td>
<td>A list of allowable feature class types: Point, Multipoint, Polyline, Polygon, MultiPatch, Sphere, Annotation, Dimension. More than one value can be supplied to the filter.</td>
</tr>
<tr>
<td>File</td>
<td>A list of file suffixes. Example: &quot;txt; e00; ditamap&quot;.</td>
</tr>
<tr>
<td>Field</td>
<td>A list of allowable field types: Short, Long, Single, Double, Text, Date, OID, Geometry, Blob, Raster, GUID, GlobalID, XML. More than one value can be supplied to the filter.</td>
</tr>
<tr>
<td>Workspace</td>
<td>A list of allowable workspace types: File System, Local Database, Remote Database. More than one value can be supplied.</td>
</tr>
</tbody>
</table>
Derived output parameters
Rule #1: All tools MUST have output

- ModelBuilder – a tool’s output is used as input to another tool
  - This is how tools are chained together
- Two types of output:
  - Required:
    - user specifies the output dataset
  - Derived:
    - Output is a modification of input or
    - Output is a not a dataset (a number) or
    - Output dataset determined by the tool
    - Executable may need to call `SetParameterAsText()`
Derived data

• The output parameter is set to Derived
• Optionally, Obtained From is set
• Examples of derived output:
  - **Get Count**: output = a number
  - **Calculate Field**: output = exact copy of input
  - **Add Field**: output = exact copy of input + a new field
  - **Create Feature Class**: new output as defined by inputs (workspace, name, schema)

See the help topic **Setting script tool parameters**
Derived data: Scalar values

- All scalar values (numbers, text, etc) are derived
  - example: Get Count
- In script tool wizard, set output to Derived
- Must use `setParameterAsText()` or `setParameter()` in your executable
Derived data – Output same as input

- Example: **Calculate Field**
- In script tool wizard, set output to **Derived** and **Obtained From** to the input parameter
- **SetParameterAsText()** not needed
  - But good habit to do anyway
Derived data – Modify input schema

- Example: **Add Field**
- In script tool wizard, set output to **Derived** and **Obtained From** to the input parameter
- In ToolValidator (coming up next), you’ll modify the input schema
  - `SetParameterAsText()` not needed
    - But good habit to do anyway
Derived data – new output

• Example: *Create Feature Class*
• In script tool wizard, set output to *Derived, Obtained From* is blank (empty)
• Optionally, set the schema in ToolValidator (coming up next)
• Use *SetParameterAsText()* in your executable
2 Validating parameters
Purpose of validation

1. Better user experience
   - Check validity of inputs *before tool is executed*

2. Describe the output of the tool (the schema) before it is executed
   - For ModelBuilder chaining
Internal validation – what you get for free

- 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?
  - *And a few more things*
Full validation goodies – dynamic choice lists

If NEW_FORMAT is chosen...

...three new choices appear
More full validation goodies
Full validation – calculate values based on other parameters

*Image of a software interface with a note: The default cell size is calculated from the extent of the Input Features parameter.*
Full validation – describing the output before tool is executed
ToolValidator Class

• Introduced at 9.3
• A Python class that you program
• Allows full control of dialog
  - Better UI, validating relationships between parameters, messaging
• Allows you to fully describe outputs for chaining in ModelBuilder
  - Through the use of a schema object
  - This is where you define the schema (fields, etc) of derived outputs
ToolValidator Class

- `initializeParameters()` – whenever a tool’s signature is requested
- `updateParameters()` – called whenever a parameter value is changed
- `updateMessages()` – called after `updateParameters()`
Demo:
ToolValidator
Class
Demo review -- Basic ToolValidator

• The basics of editing a ToolValidator class
• Setting up keyword lists, categories
• Dynamic update of keyword lists
  - Keyword list changes based on values in another parameter
  - (The kind of stuff basic validation cannot do)
Do not open datasets in tool validation

- You may be tempted to open and scan an input dataset in your tool validation code
  - For example, read some field values to determine range of data in order to populate another parameter
- The problem with this is that the input may not exist
  - Because your tool is being validated as part of a ModelBuilder chain
Describing the output
Using the schema object
Describing the output

In ModelBuilder, Add Field updates the description of its output to contain the new field "TrackingID". The Calculate Field tool 'sees' this field, even though Add Field hasn't executed yet.
Describing the output

The Clip tool updates the description of its output data.

Name: \data\sample.gdb\buildings_clip
Fields: \textit{(same as Buildings)}
Extent: \textit{(intersection of Buildings and StudyArea)}

The Polygon to Raster tool calculates the default cell size based on the extent calculated by the Clip tool.
Output parameters have a schema object

- The schema object contains the description of the output dataset
- You set up rules about how you want this description to be constructed
  - Example: “use the extent of the 2\textsuperscript{nd} input dataset”
  - Example: “use the fields of the input, but add these new fields”
## Schema object methods (rules)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>String: &quot;Feature&quot;, &quot;Table&quot;, &quot;Raster&quot;, &quot;Container&quot; (for workspaces and feature datasets). (Read-only property.)</td>
</tr>
<tr>
<td>clone</td>
<td>Boolean</td>
</tr>
<tr>
<td>featureTypeRule</td>
<td>String: &quot;AsSpecified&quot;, &quot;FirstDependency&quot;</td>
</tr>
<tr>
<td>featureType</td>
<td>String: &quot;Simple&quot;, &quot;Annotation&quot;, &quot;Dimension&quot;</td>
</tr>
<tr>
<td>geometryTypeRule</td>
<td>String: &quot;Unknown&quot;, &quot;FirstDependency&quot;, &quot;Min&quot;, &quot;Max&quot;, &quot;AsSpecified&quot;</td>
</tr>
<tr>
<td>geometryType</td>
<td>String: &quot;Point&quot;, &quot;Multipoint&quot;, &quot;Polyline&quot;, &quot;Polygon&quot;</td>
</tr>
<tr>
<td>extentRule</td>
<td>String: &quot;AsSpecified&quot;, &quot;FirstDependency&quot;, &quot;Intersection&quot;, &quot;Union&quot;, &quot;Environment&quot;</td>
</tr>
<tr>
<td>extent</td>
<td>Extent object</td>
</tr>
<tr>
<td>fieldsRule</td>
<td>String: &quot;None&quot;, &quot;FirstDependency&quot;, &quot;FirstDependencyFIDsOnly&quot;, &quot;All&quot;, &quot;AllNoFIDs&quot;, &quot;AllFIDsOnly&quot;</td>
</tr>
<tr>
<td>additionalFields</td>
<td>Python list of field objects</td>
</tr>
<tr>
<td>cellSizeRule</td>
<td>String: &quot;AsSpecified&quot;, &quot;FirstDependency&quot;, &quot;Min&quot;, &quot;Max&quot;, &quot;Environment&quot;</td>
</tr>
<tr>
<td>cellsize</td>
<td>double</td>
</tr>
<tr>
<td>rasterRule</td>
<td>String: &quot;FirstDependency&quot;, &quot;Min&quot;, &quot;Max&quot;, &quot;Integer&quot;, &quot;Float&quot;</td>
</tr>
<tr>
<td>rasterFormatRule</td>
<td>String: &quot;Img&quot;, &quot;Grid&quot;</td>
</tr>
<tr>
<td>additionalChildren</td>
<td>Python list of datasets to add to a workspace schema.</td>
</tr>
</tbody>
</table>
Summary
A tool does 3 things

- Defines parameters
  - Including the output dataset schema (if necessary)
- Validates parameters before it executes
  - Default validation (what you get from the framework)
  - Custom validation (ToolValidator)
  - (Output dataset schema defined during validation)
- Executes some code that performs the actual work
Python toolboxes

esriurl.com/pythontoolbox
The Python toolbox

• Everything is done in Python
  - Easier to create
  - Easier to maintain

• An ASCII file (.pyt) that defines a toolbox and tool(s)

• Tools look and behave like any other tool
A Python toolbox is defined by classes

- Every Python toolbox has a toolbox class
- And one or more tool classes
  - Parameters
  - Validation
  - Source code
Getting started
Parameters

- Defined using Parameter objects in getParameterInfo

- Every parameter has a data type
  - String, Double, Boolean, Feature Layer, Raster Dataset, ...

*Note: 10.1 SP1 added new datatype keywords*

http://esriurl.com/5959
http://esriurl.com/5960

```python
def getParameterInfo(self):
    in_features = arcpy.Parameter(
        displayName="Input features",
        name="in_features",
        datatype="GPFeatureLayer",
        parameterType="Required",
        direction="Input")
```
Data types

- Python toolbox have 2 extra data types

1. Value Table

- Set **columns** to:
  - A list of lists
  - A list of parameter objects

```python
param0.datatype='GPValueTable'
param0.columns = [['Feature Layer', 'Features'], ['Long', 'Ranks']]```
Data types

2. Composite

• Assign a list of datatypes to `datatype`

```python
param.datatype=['DERasterDataset', 'DERasterCatalog']
```
Validation

• Everything that happens before pushing OK

• Customize how parameters respond and interact to values and each other

• Controls tool behavior
Schema

• Output datasets are described by a Schema object
  - Necessary for using your tool in **ModelBuilder**

• `parameterDependencies` set initial schema
  - You apply extra rules to the schema

```python
def getParameterInfo(self):
    # -- parameters defined here -- #
    param1.parameterDependencies = [param0.name]
    param1.schema.clone = True

    add_field = arcpy.Field()
    add_field.name = "Status"
    add_field.type = "String"
    param1.schema.additionalFields = [add_field]
```
Validation - updateParameters

- Parameter interaction
- Calculate default values
- Enable/disable parameters

```
def updateParameters(self, parameters):
    """Set the default distance threshold to 1/100 of the largest of the
    width or height of the extent of the input features.""
    if parameters[0].value:
        if not parameters[6].altered:
            extent = arcpy.Describe(parameters[0].value).extent
            if extent.width > extent.height:
                parameters[6].value = extent.width / 100.0
            else:
                parameters[6].value = extent.height / 100.0
```

- Called whenever a parameter value is altered
Validation - updateMessages

- Called after returning from internal validation
- Provide custom error/warning messages

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

    # If input is not versioned add error to parameter
    if parameters[0].value and parameters[0].altered:
        if not arcpy.Describe(parameters[0].value).isVersioned:
            parameters[0].setErrorMessage("Input must be versioned")
    return
Validation - isLicensed

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

```python
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

• The ‘business logic’ of the tool is found in `execute`
• This is where any analysis, conversion, and data creation occurs
• Execute has arguments for dealing with parameters and messages

```python
def execute(self, parameters, messages):
    in_features = parameters[0].valueAsText
    out_feature_class = parameters[1].valueAsText
    interval = parameters[2].value

    messages.addMessage("Performing analysis...")
    ## Your analysis
```
Python toolboxes
Python toolbox organization

- Yes—you can put all your code in one .pyt
- But do you really want to?
  - Can make sense to organize into separate files

```python
import arcpy
from mytool import MyTool

class Toolbox(object):
    def __init__(self):
        self.label = "Tools"
        self.alias = "alias"
        self.tools = [MyTool]  # list of tool classes
```

Resources

- **Python toolbox help**
  - esriurl.com/pythontoolbox

- **Working examples**
  - esriurl.com/pyt1
  - esriurl.com/pyt2
  - esriurl.com/pyt3

- **Python for ArcGIS**
  - resources.arcgis.com/en/communities/python/

@arcpy
Rate This Session
Your feedback is important

- Quick Link from www.esri.com/devsummit
Developing Geoprocessing Tools in a Python Toolbox
David Wynne, Dale Honeycutt

Enter Offering ID: 283
Rate This Session

Presentation Skills
Poor: 1 2 3 4 5
Excellent

Presentation Content
Poor: 1 2 3 4 5
Excellent

Overall Value
Poor: 1 2 3 4 5
Excellent

Comments
Please provide suggestions as to how the presenter can improve this workshop. (500 character limit)

SEND FEEDBACK