

Esri Developer Summit

March 26–29, 2012 | Palm Springs, California

esri.com/events/devsummit



Developing Geoprocessing Tools in a Python Toolbox

Dave Wynne

Dale Honeycutt

A decorative graphic at the bottom of the slide. It features a curved orange border at the top. Below it is a semi-transparent map of a geographical area. Overlaid on the map is Python code in a light blue font. The code includes symbols like 'esri.symbol.SimpleLineSymbol', 'new dojo.Color([0,0,0,0.5])', and 'feature.setSymbol'.

```
esri.symbol.SimpleLineSymbol  
new dojo.Color([0,0,0,0.5])  
polySymbol =  
feature.setSymbol(polySymbol)  
}  
else if(f == 1) {  
var polySymbolGreen =  
polySymbolGreen.setOutlineColor([0,0,0,0.5]);  
polySymbolGreen.setSymbol(polySymbolGreen);  
feature.setSymbol(polySymbolGreen);  
}  
else if(f == 2) {  
var polyBlue = new esri.symbol.SimpleLineSymbol([0,0,0,0.5]);  
polyBlue.setOutlineColor([0,0,0,0.5]);  
polyBlue.setSymbol(polyBlue);  
feature.setSymbol(polyBlue);  
}
```

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?

- **First Dev Summit?**
- **General geoprocessing?**
 - **Python?**
 - **Script tools?**

Why we create tools

- **Becomes part of the geoprocessing framework**
 - Run from a tool dialog, ModelBuilder, Python
- **Familiar interface**
- **Basic validation saves coding checks**
 - Valid type, Exists checks, etc.

Why we create tools *cont...*

- **Easy to share**
- **Generic**
 - **Can be used with different data and varied scenarios**
- **Communicates with the application**
 - **Layers from the map**
 - **Messages**

Geoprocessing tools

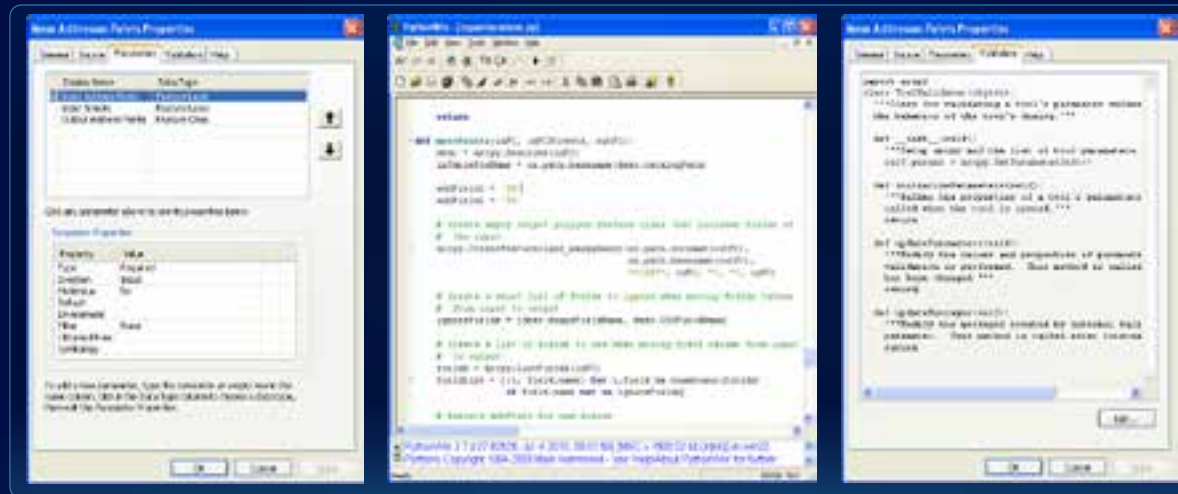
- **A geoprocessing tool does three types of work:**
 - **Defines its parameters**
 - **Validates its parameters**
 - **Executes some code that performs the actual work**

Geoprocessing tools

- **A geoprocessing tool does three types of work:**
 - **Defines its parameters**
 - **Validates its parameters**
 - **Executes some code that performs the actual work**

Creating script tools pre-10.1

- Parameters defined through a wizard
- Validation code that lives in the toolbox
- Separate source code



Demo: Script tool basics

Adding a script tool

```
esri.symbol.SimpleLineSymbol,
new dojo.Color([0,0,0,0.5]),
polySymbol,
feature.setSymbol(polySymbol);
} else if(f == 1) {
var polySymbolGreen = new
polySymbolGreen.setOutline(
symbol.SimpleLineSymbol(esri.symbol.
Color([0,0,0,0.5]), 1));
polySymbolGreen.setSymbol(polySymbolGreen);
} else if(f == 2) {
var polyBlue = new esri.symbol.SimpleLineSymbol(
new dojo.Color([0,0,255,0.5]), 1);
polyBlue.setOutline(new esri.symbol.SimpleLineSymbol(
new dojo.Color([0,0,255,0.5]), 1));
polyBlue.setSymbol(polyBlue);
}
```

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"
- **Tranform to python list with:**
 - `mylist = string.split(";")`
 - `Fields = arcpy.GetParameterAsText(1).split(";")`

Parameter filters

Value List	A list of string or numeric values. Used with String, Long, Double, and Boolean parameter data types.
Range	A minimum and maximum value. Used with Long and Double data types.
Feature Class	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.
File	A list of file suffixes. Example: "txt; e00; ditamap".
Field	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.
Workspace	A list of allowable workspace types: File System, Local Database, Remote Database. More than one value can be supplied.

Validation

Programming the ToolValidator class

```
esri.symbol.SimpleLineSymbol,
new dojo.Color([0, 0, 1]),
polySymbol,
feature.setSymbol(polySymbol);
} else if(f == 1) {
var polySymbolGreen = new
polySymbolGreen.setOutline(
symbol.SimpleLineSymbol(esri.symbol.
Color([0, 0, 0.5]), 1));
polySymbolGreen.setSymbol(polySymbolGreen);
feature.setSymbol(polySymbolGreen);
} else if(f == 2) {
var polyBlue = new esri.symbol.SimpleLineSymbol(
new dojo.Color([0, 0, 1]), 1);
polyBlue.setOutline(new esri.symbol.SimpleLineSymbol(
new dojo.Color([0, 0, 1]), 1));
polyBlue.setSymbol(polyBlue);
feature.setSymbol(polyBlue);
}
```

Validation is everything that happens...

- Before the OK button is pushed

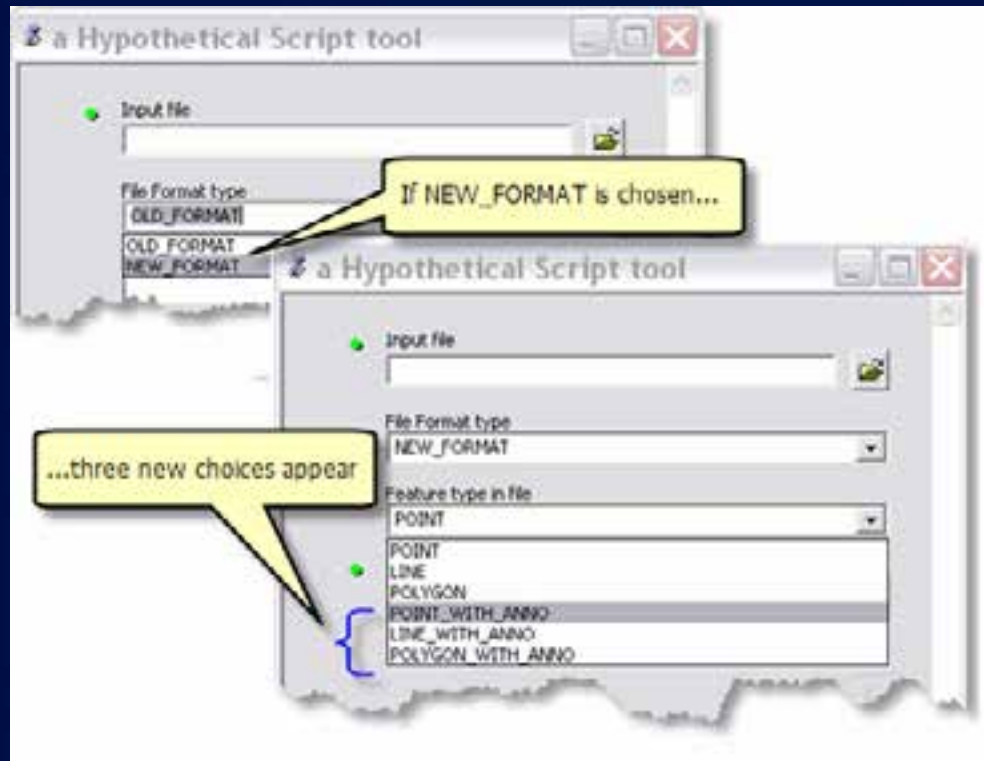
Purpose of validation

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?

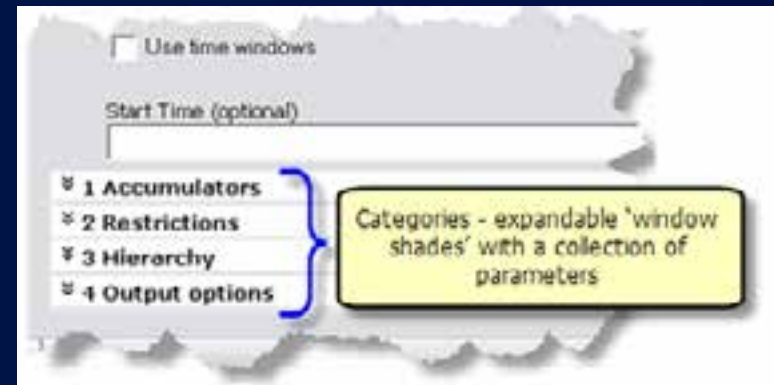
Full validation – dynamic choice lists



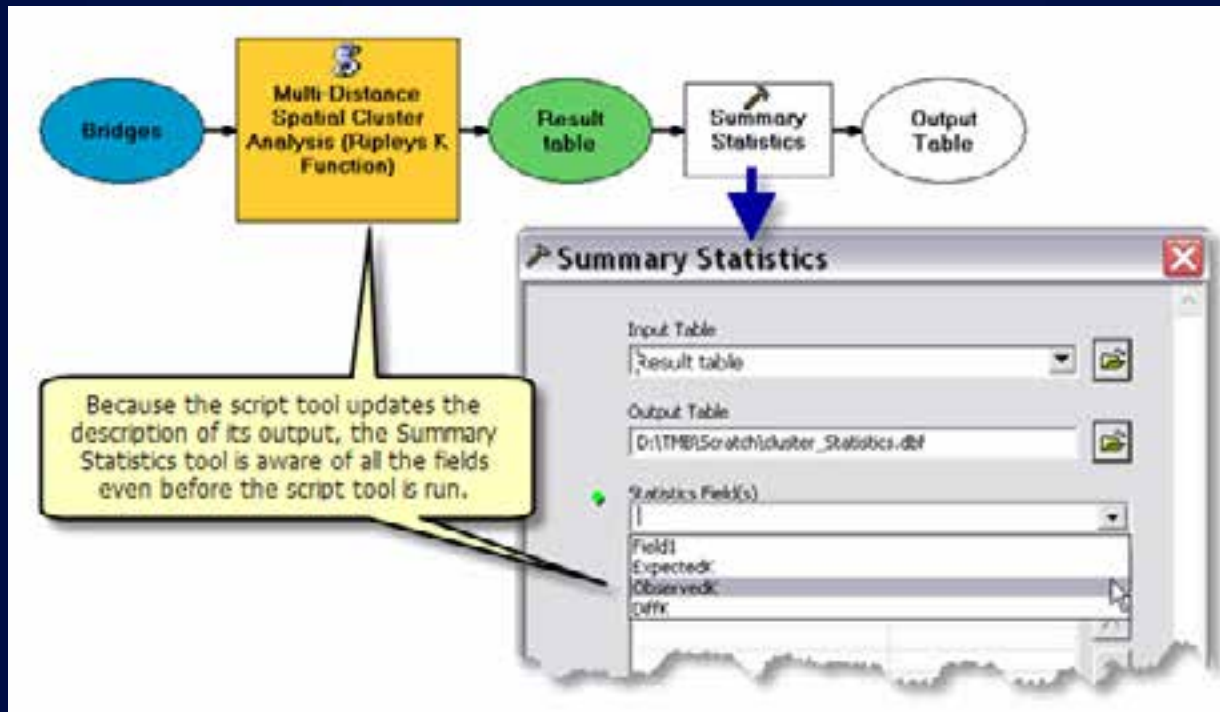
Full validation – calculate values based on other parameters



Full validation – more goodies



Full validation -- describing the output

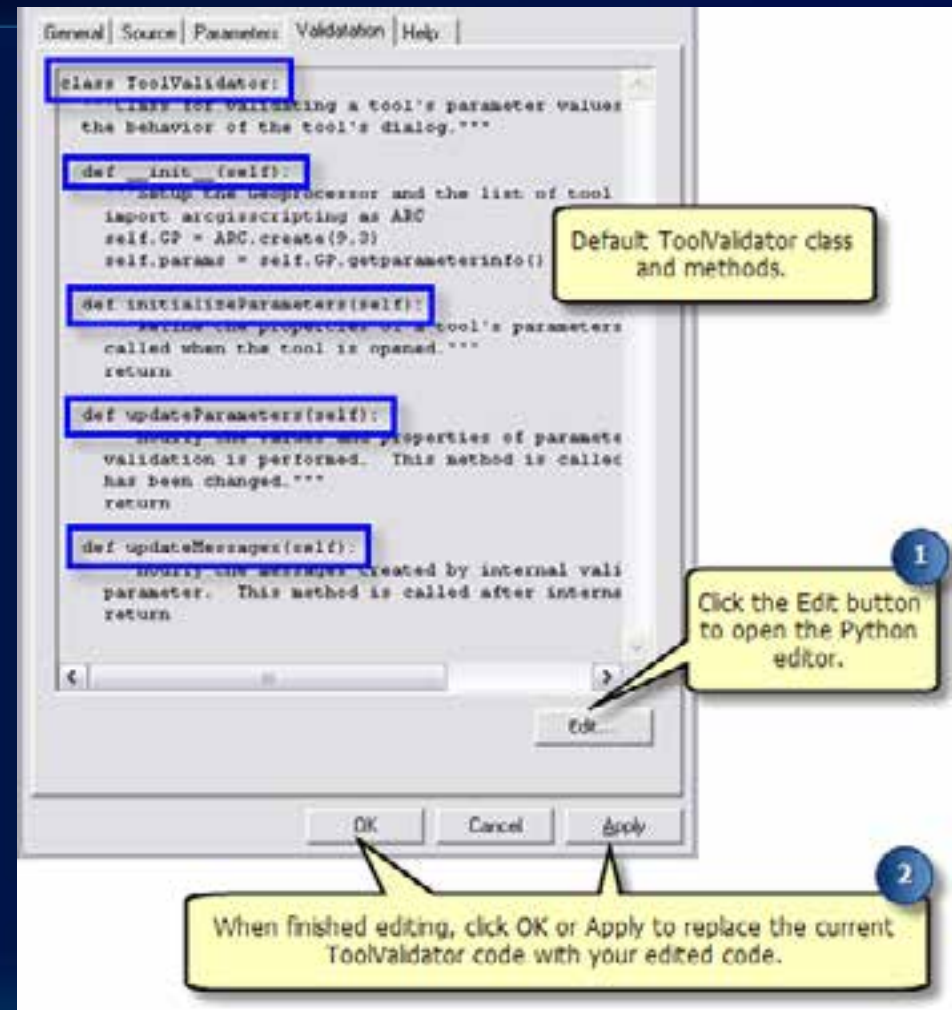


ToolValidator Class

- **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: controlling the UI

```
esri.symbol.SimpleLineSymbol,
new dojo.Color([0, 0, 1]),
polySymbol,
feature.setSymbol(polySymbol);
} else if(f == 1) {
var polySymbolGreen = new
polySymbolGreen.setOutline(
symbol.SimpleLineSymbol(esri.symbol.
Color([0, 0, 0.5]), 1));
polySymbolGreen.setSymbol(polySymbolGreen);
} else if(f == 2) {
polySymbolBlue = new esri.symbol.SimpleLineSymbol(
polySymbolBlue.setOutline(new
esri.symbol.SimpleLineSymbol(
new dojo.Color([0, 0, 1]), 1));
polySymbolBlue.setSymbol(polySymbolBlue);
}
```

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)

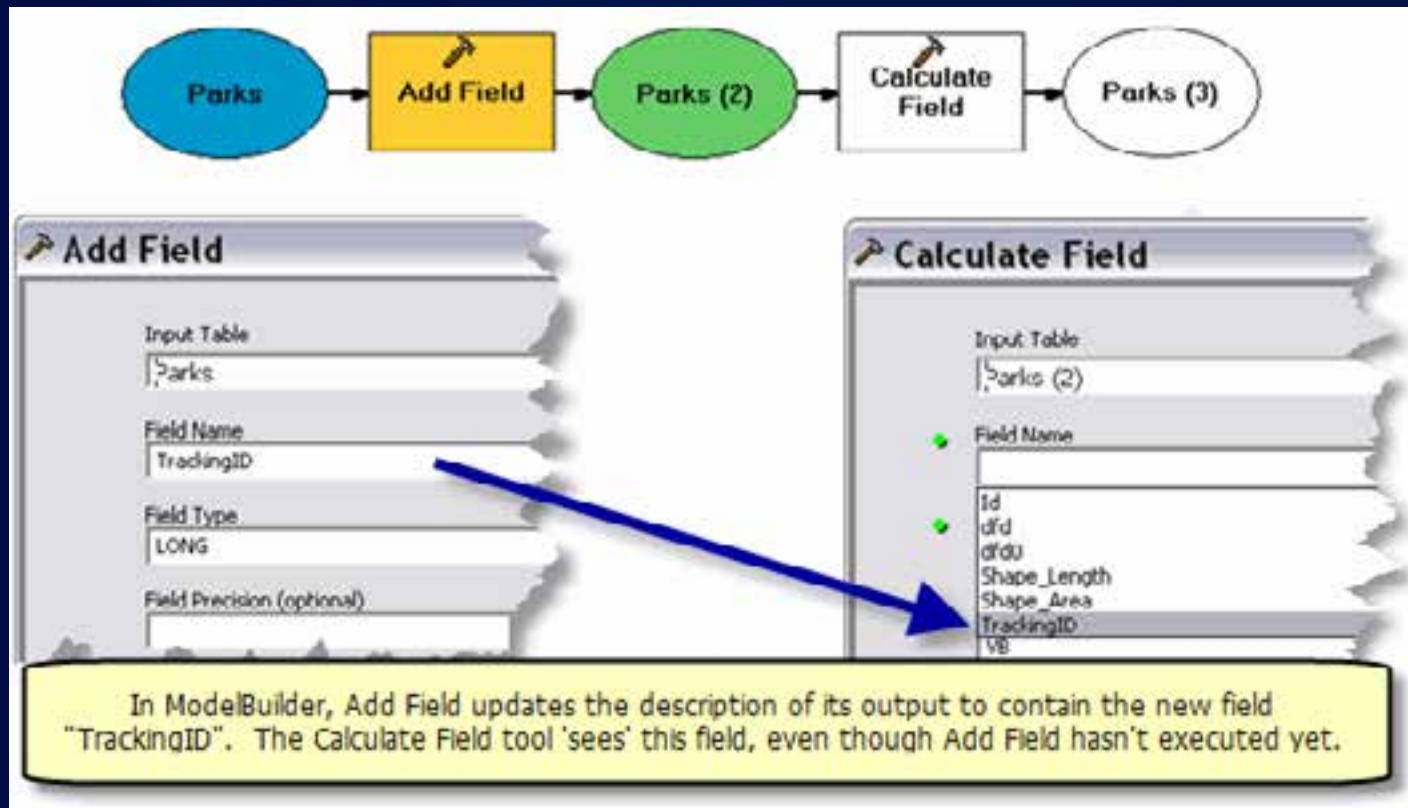
Describing the output

Using the schema object

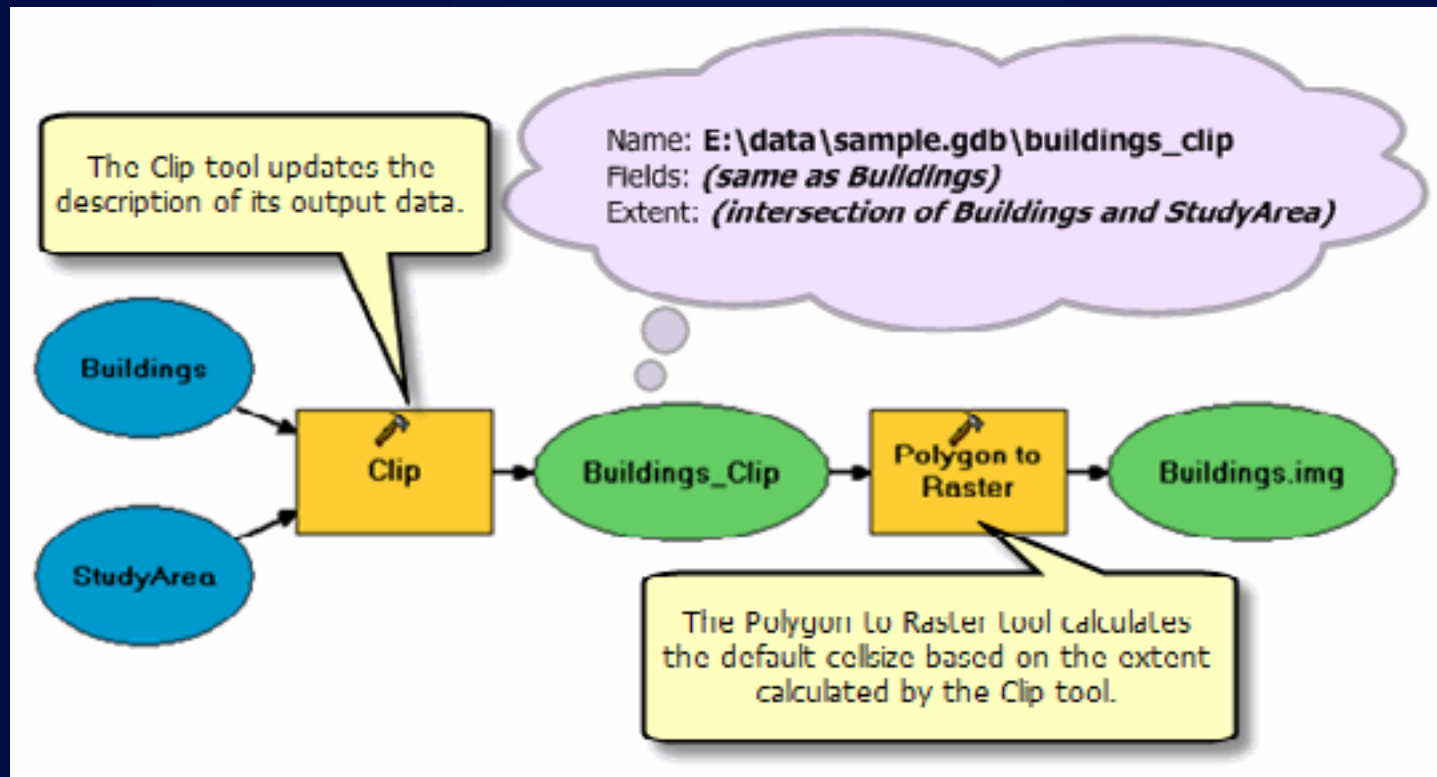


```
esri.symbol.SimpleLineSymbol({
  color: [0, 0, 255],
  width: 2
});
feature.setSymbol(new esri.symbol.SimpleLineSymbol({
  color: [0, 0, 255],
  width: 2
}));
feature.setSymbol(new esri.symbol.SimpleLineSymbol({
  color: [0, 0, 255],
  width: 2
}));
feature.setSymbol(new esri.symbol.SimpleLineSymbol({
  color: [0, 0, 255],
  width: 2
}));
```

Describing the output



Describing the output



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 2nd input dataset”**
 - **Example: “use the fields of the input, but add these new fields”**

Schema object methods (rules)

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

Python toolboxes

```
esri.symbol.SimpleLineSymbol([0,0,0,0.5]),
new dojo.Color([0,0,0,0.5]),
feature.setSymbol(symbol);
} else if(f == 1) {
var polysymbolGreen =
polySymbolGreen.setOutline(
symbol.SimpleLineSymbol(esri.symbol.
Color([0,0,0,0.5]), 1));
polySymbolGreen.setcolor(new dojo.
Color([0,0,0,0.5]));
feature.setSymbol(polysymbolGreen);
} else if(f == 2) {
var polysymbolBlue = new esri.symbol.SimpleLineSymbol(
[0,0,0,0.5]),
new dojo.Color([0,0,0,0.5]),
feature.setSymbol(polysymbolBlue);
}
```

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 of more tool classes**
 - **With methods that**
 1. **Define the tool**
 2. **Establish parameters**
 3. **Validation methods**
 4. **'execute'**

Toolbox class

```
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]
```

- The **Toolbox** class is always named *Toolbox*

```
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

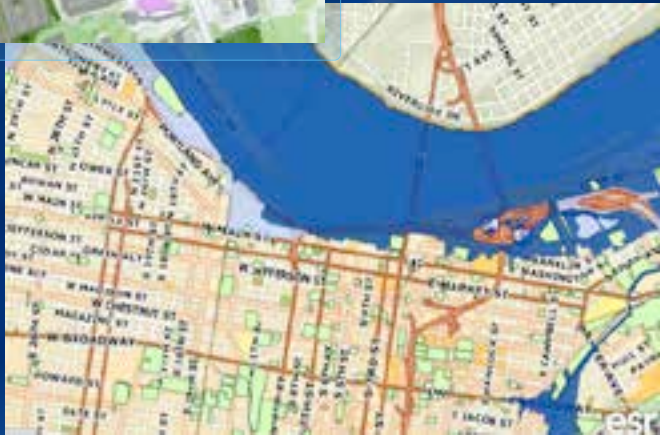
    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before internal
        validation is performed. This method is called whenever a parameter
        has been changed."""
        return

    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
```



Demo: Getting started



Parameters

- Defined using Parameter objects in `getParameterInfo`

```
def getParameterInfo(self):  
    """Define parameter definitions"""  
    in_features = arcpy.Parameter(  
        displayName="Input features",  
        name="in_features",  
        datatype="Feature Layer",  
        parameterType="Required",  
        direction="Input")
```

- parameterType (Required, Optional, Derived)
- direction (Input, Output)
- When tool is executed, parameter values are sent to **execute**

Data types

- **Core concept of geoprocessing**
- **Every parameter has a data type**
 - **String, Double, Boolean, Feature Layer, and Raster Dataset, etc.**
- **Every data type has:**
 - **Built-in validation logic**
 - **A UI control**
 - **A string representation**

Schema

- Output datasets are described by a Schema object
 - Important for using your tool in **ModelBuilder**
- parameterDependencies set initial schema
 - You apply extra rules to the schema

```
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

- Everything that happens **before** pushing OK
- Customize how parameters respond and interact to values and each other
- Controls tool behavior

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

```
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

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

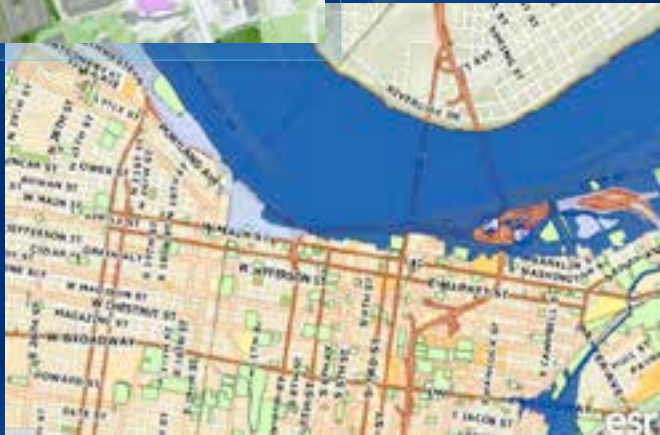
Execution

- 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

```
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
```



Demo: Python toolboxes



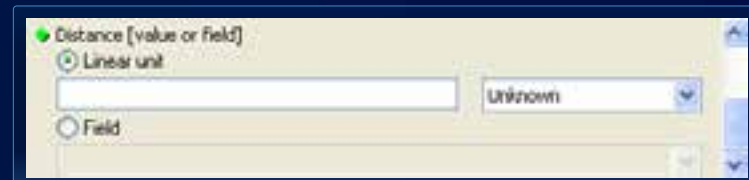
Organizing

- The entire toolbox *can* be organized in one file
- With a large toolbox, can make sense to organize into separate files

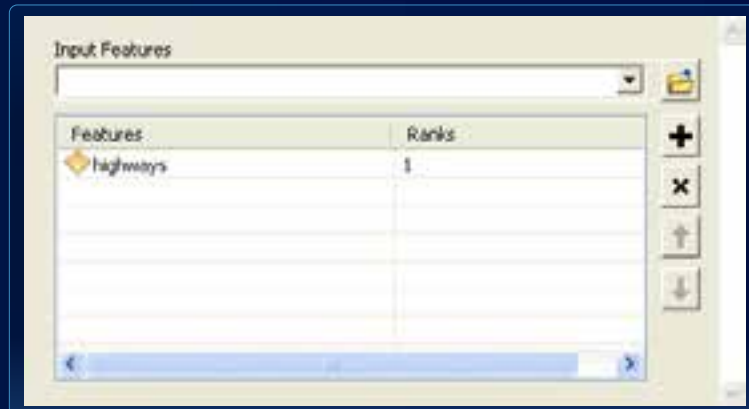
```
def execute(self, parameters, messages):  
    import tbx_utils  
    tbx_utils.do_work(parameters[0].valueAsText,  
                      parameters[1].valueAsText)  
    return
```

Bonuses

- Composite data types



- Value tables

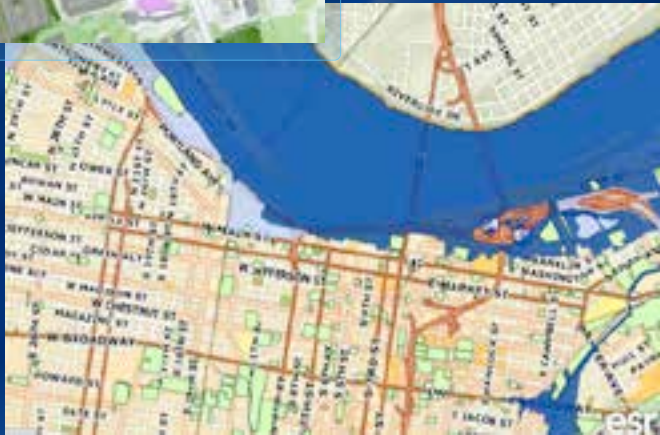


- Custom license checking





Demo: Python toolboxes



Python toolbox caveats

- **Doesn't support other tool types**
 - **Can't *add* models, or other tool types**
- **Doesn't support embedding and encryption of source code**



esri