Esri International Developer Summit

Palm Springs, CA

Developing Tools with Python

David Wynne, Jon Bodamer

Abstract

- Developing Tools with Python

- Join us as we step through the process of creating geoprocessing tools in Python. We will demonstrate how to create Script Tools and Python Toolboxes. This workshop will highlight the important decisions in making fully functional geoprocessing tools.

Why we create tools

- *Tools solve problems*
- Tools are:
  - Easy to share
  - Generic
  - Becomes part of the geoprocessing framework
    - Python, Dialogs, ModelBuilder, Services
  - Basic validation saves coding checks

What makes a good tool?

- Performs essential and elemental operations on GIS data
- Has an output (derived or not)
- Has a reasonable number of parameters
- Is documented
- Follow geoprocessing conventions

Make your tool familiar

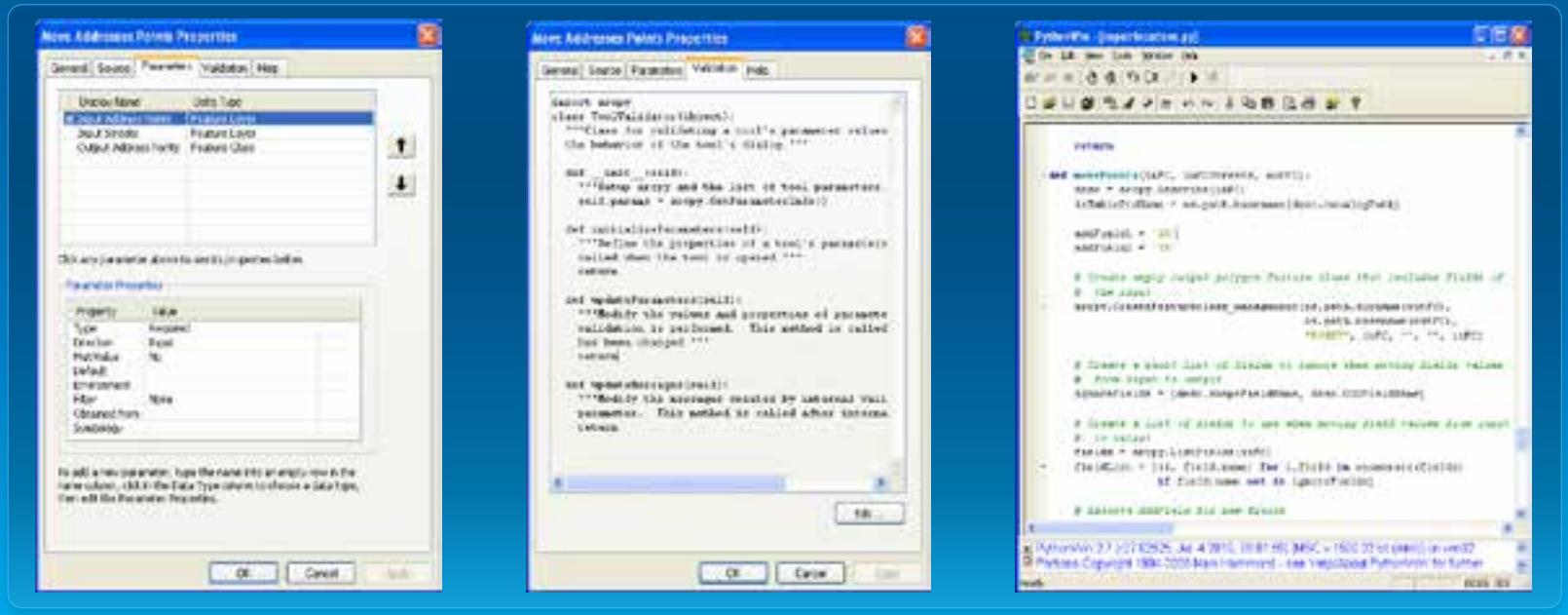
- System geoprocessing tools follow this ordering convention

1. Required input datasets
2. Required output datasets
3. Required modifiers
4. Optional inputs
5. Optional modifiers
6. Optional outputs
7. Derived outputs

Geoprocessing tools

- A geoprocessing tool does three types of work:

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



Define 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
- Data type
  - Layer types: Feature Layer, Raster Layer, Table View
  - Scalar types: String, Boolean, Long, Double

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

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

Execution

- A geoprocessing tool does three types of work:

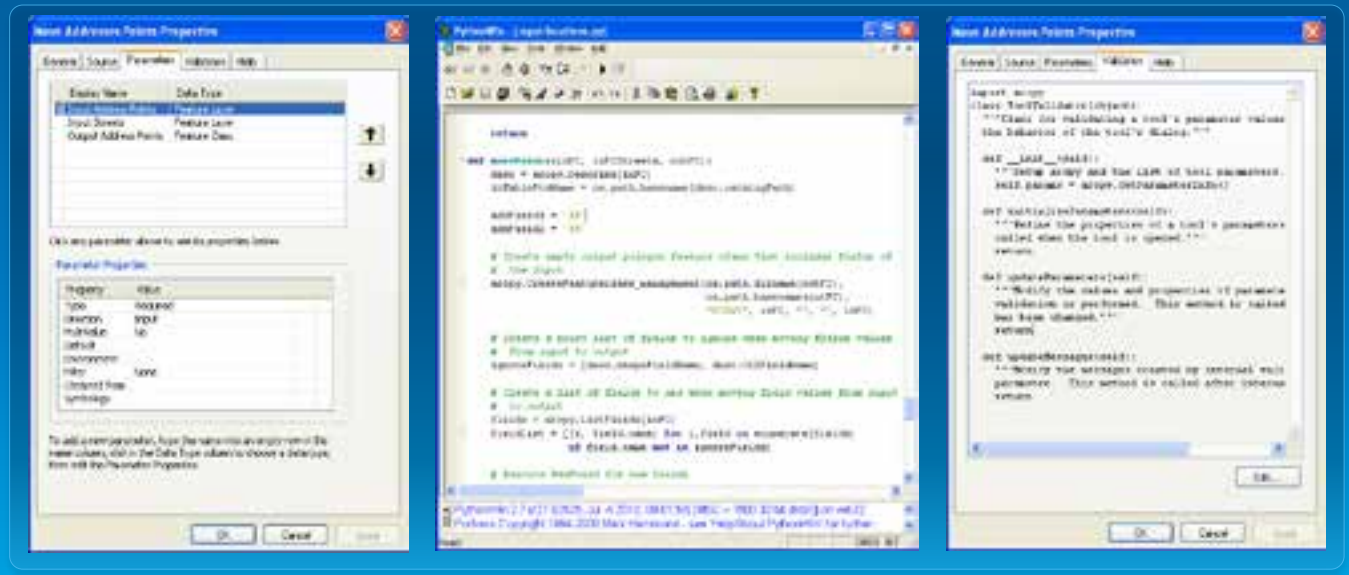
1. Defines its parameters
2. Validates its parameters
3. Executes code that performs the actual work

- The ‘business logic’ of your tool

Script tools

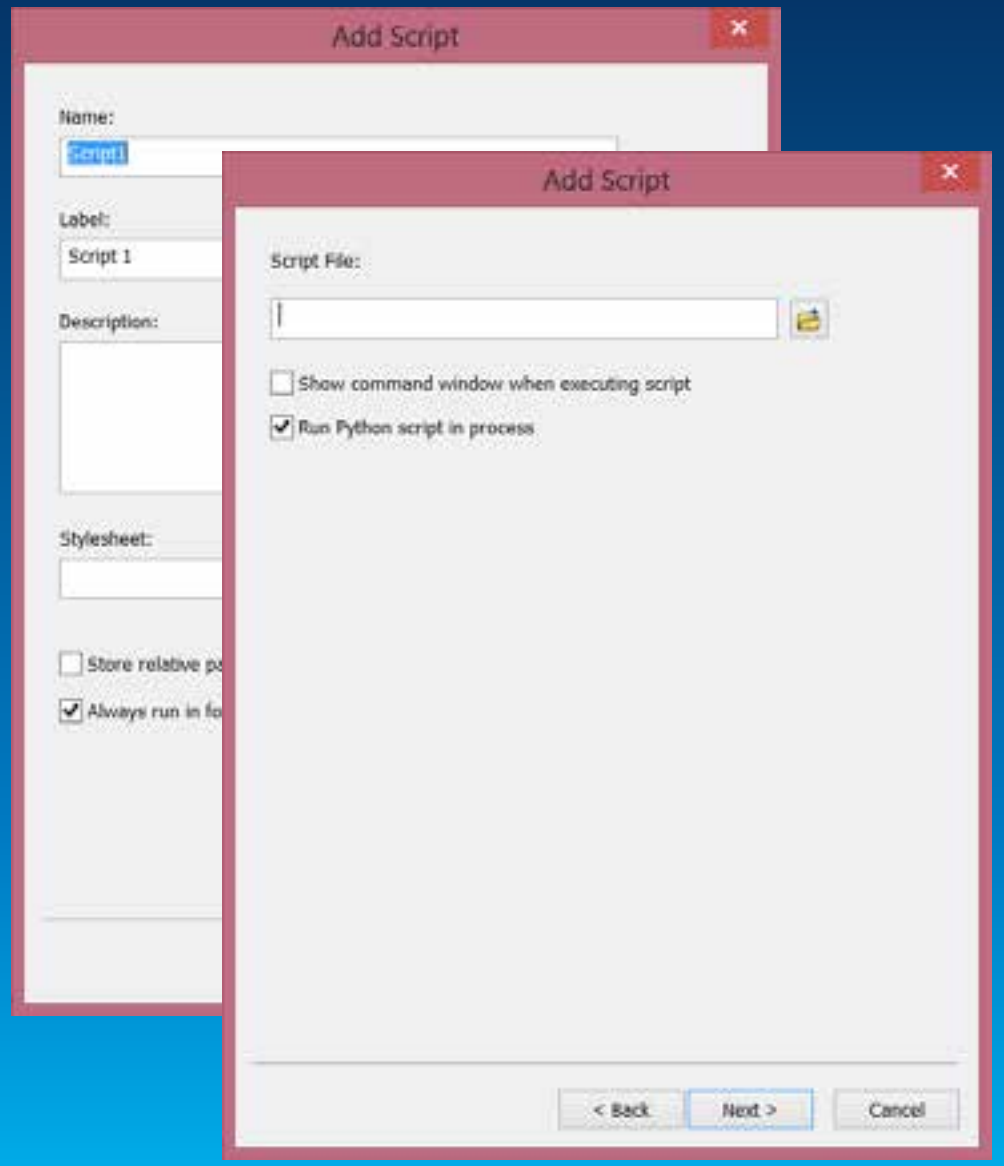
Creating script tools

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



Script tools

- **Script tool wizard**
- **Tools created using the Script tool wizard run a file on disk**
  - ***Doesn't have to a .py file!***
- **Adding a source file is not required at the beginning**



Script tools – Getting and setting parameters

- Parameters are received using either:
  - arcpy.GetParameterAsText : value is a string
  - arcpy.GetParameter : appropriate object type

- You can also send a parameter value back for a derived output/value with:
  - arcpy.SetParameterAsText
  - arcpy.SetParameter

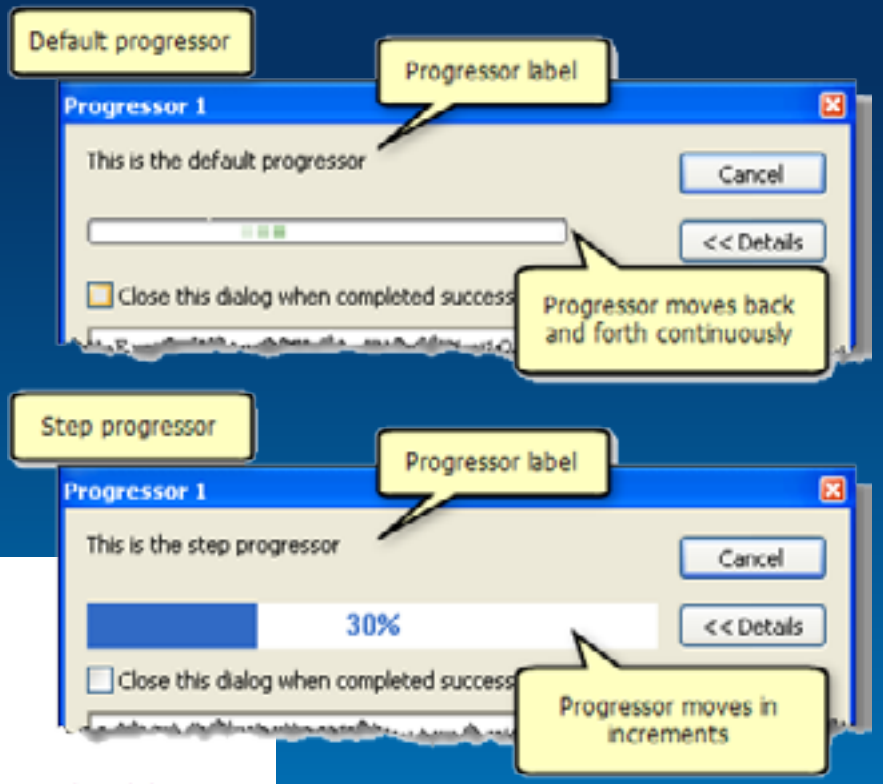
Messages in a script tool

- Add custom messages into the tool's messages
  - AddError
  - AddMessage
  - AddWarning
  - AddIDMessage

- *Note*: Error messages do not raise an Exception
- For unhandled exceptions, all exception messages are added to the tool messages

Progressor

- **Using SetProgressor is another way to indicate progress**



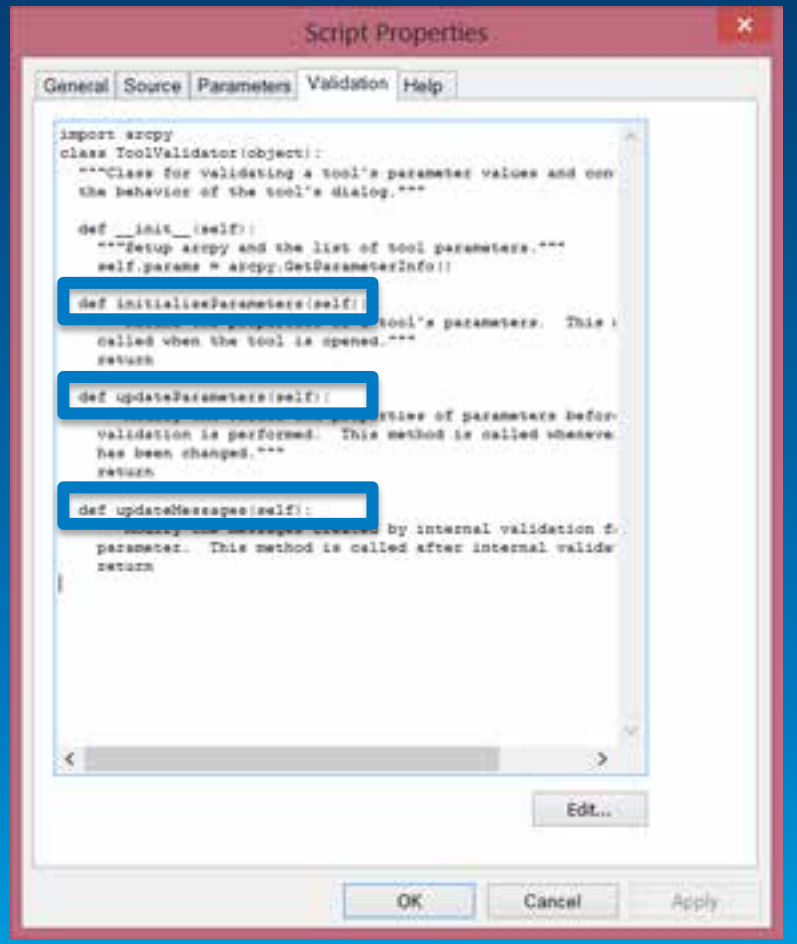

```
arcpy.SetProgressor('step', 'Copying data', 0,
                    len(feature_classes), 1)

for fc in feature_classes:
    arcpy.SetProgressorLabel('Loading {0}...'.format(fc))

    # < some data handling >

    arcpy.SetProgressorPosition()
```

ToolValidator Class

- A Python class that you write
- Allows more 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

ToolValidator Class

- initializeParameters()
  - whenever a tool's signature is requested
- updateParameters()
  - called whenever a parameter value is changed
- updateMessages()
  - called UpdateParameters()



Demo

Demo: Script tools

Jon Bodamer

Python toolboxes

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 type of tool

Python toolbox benefits

- All Python
- Frees you from Desktop
  - Frees you from the Script tool wizard
- Can easily make changes and refresh

A Python toolbox is defined by classes

- Every Python toolbox has a toolbox class
- And one or more tool classes
  - With methods that
    1. Define the tool
    2. Establish parameters
    3. Validation methods
    4. ‘execute’

```
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
```

Parameters

- **Tool parameters are defined using Parameter objects in getParameterInfo**

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

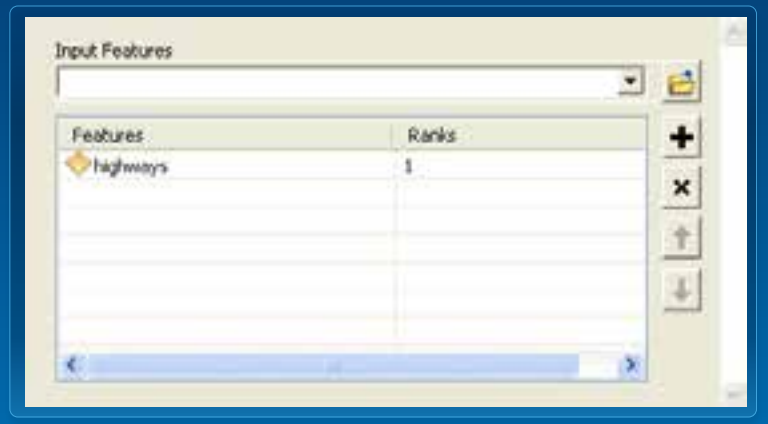
  - **parameterType (Required, Optional, Derived)**
  - **direction (Input, Output)**
  - **Just like in a script tool, every parameter has a data type**

Extra Python toolbox parameters

1. **Value Table**

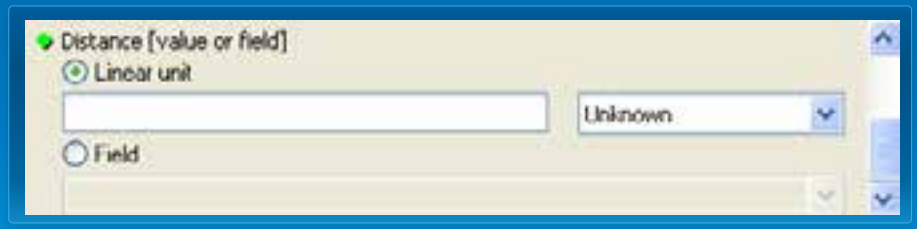
   Set *columns* property to:

   - **A list of lists or parameter objects**



2. **Composite data types**

```
param.datatype = ["DERasterDataset",
                  "DERasterCatalog"]
```



Schema

- Output datasets are described by a Schema object
  - Needed to connect tools 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 - 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):
    """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*
- 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("Using input {}".format(in_features))

    ## additional code
```

Messages in a Python toolbox

- All arcpy message functions are respected
- Can also use messages object in execute

```
def execute(self, parameters, messages):
```

  - addMessage
  - addErrorMessage
  - addWarningMessage
  - addIDMessage
  - addGPMessages

Demo

Demo: Python toolboxes

Jon Bodamer

Python toolbox organization

- **Yes—you can put all your code in one .pyt**
- **But will often make sense to organize as separate files**

```
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
```

- http://esriurl.com/SpatialAnalystSupplementalTools

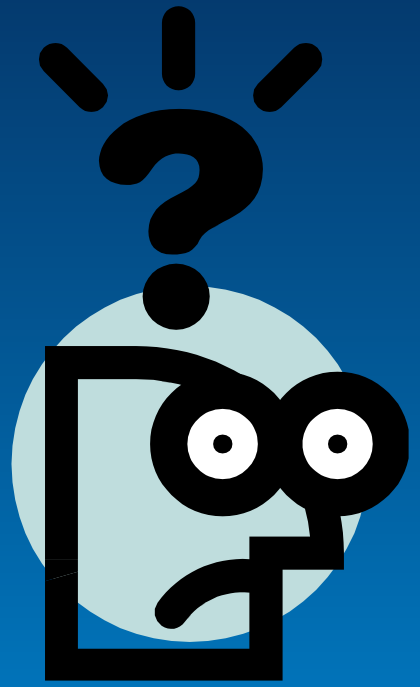
Important differences Python toolbox and script tools

- Python toolboxes don’t support other tool types
  - Can’t *add* models, or other tool types
- Python toolboxes don’t support embedding and encryption of source code

More Python

- Monday
  - Building Live Data Feeds Using Python (2:15 pm – 2:45 pm)
  - Creating Geoprocessing Tools in a Python Toolbox (4:45 pm – 5:15 pm)
- Tuesday
  - Administering Your Enterprise Geodatabase through Python (2:30 pm – 3:30 pm)
  - Deploying Your Geoprocessing Tools as Python Modules (2:30 pm – 3:30 pm)
  - Administering ArcGIS for Server with Python (4:00 pm – 4:30 pm)
  - Using Python with ArcGIS Runtime Desktop SDKs (4:30 pm – 5:00 pm)
- Wednesday
  - Using Geometric Networks with Python and Geoprocessing Services (10:30 am – 11:30 am)
  - Accessing C Type Libraries with Python Script Tools (1:00 – 2:00 pm)
  - Python Map Automation—Beyond the Basics of arcpy.mapping (2:30 pm – 3:30 pm)
- Thursday
  - Working with Raster Data Using ArcPy (8:30 am – 9:30 am)
  - Working with Feature Data Using ArcPy (10:00 am – 11:00 am)
  - Creating Mosaic Datasets and Publishing Image Services Using Python (10:00 am – 11:00 am)

Thank you. Questions?



- http://www.esri.com/events/devsummit/session-rater


esri
Understanding our world.