



# Advanced Python

- **Lists, Dictionaries, Tuples**
- **Geometry objects**
- **Data Access**
- **Mapping Module**
- **Python Toolboxes**

# Key Python data structures

- **Lists**

- Flexible
- Ordered

- **Tuples**

- Immutable
- Ordered

- **Dictionary**

- Key/value pairs

```
• l = ['10 feet', '20 feet', '50 feet']
```

```
• t = ('Thurston', 'Pierce', 'King')
```

```
• d = {'ProductName': 'desktop',  
      'InstallDir': 'c:\\ArcGIS\\Desktop10.1'}
```

# Working with Lists

- **Zero-indexed - first value is at location 0**  
**a = [5, 3, 6, 9, 0, 30]**
- **a[2] → 6**
- **val1, val2, val3, val4 = a → 1, 2, 3, 4**
- **a[3:] [9,0,30]**
- **length: len(a) → 6**
- **a.append(4) → [5, 3, 6, 9, 0, 30, 4]**

# Working with Dictionaries

- `employees = { 1000 : "John Smith", 1001 : "Kevin Jones", 1002 : "Mary McMurray" }`
- `employees[1001] → "Kevin Jones"`
- `employees.keys → [1000, 1001, 1002]`
- `employees.values → ["John Smith", "Kevin Jones", "Mary McMurray"]`
- `employees.items → a list of the items as a tuple (similar to a list in a list)`

# List comprehension

- Compact way of mapping a list into another

```
>>> distances = [10, 50, 200]
>>> new_distances = ['{} feet'.format(d) for d in distances]
>>> print(new_distances)
['10 feet', '50 feet', '200 feet']

>>> field_names = [f.name for f in arcpy.ListFields(table)]
>>> print(field_names)
[u'OBJECTID', u'NAME', u'ADDRESS']
```

# Defining Functions

- Organize and re-use functionality

```
• import arcpy

def increase_extent(extent, factor):
    """Increases the extent by the given factor"""

    XMin = extent.XMin - (factor * extent.XMin)
    YMin = extent.YMin - (factor * extent.YMin)
    XMax = extent.XMax + (factor * extent.XMax)
    YMax = extent.YMax + (factor * extent.YMax)

    return arcpy.Extent(XMin, YMin, XMax, YMax)

• oldExtent = arcpy.Describe("boundary").extent
• newExtent = increase_extent(oldExtent, .1)
```

Define your  
function

Return a  
result

Call the  
function

# Geometry



# Geometry and cursors

- Can create geometry in different ways
  - Geometry objects
  - List of coordinates
  - Using other formats
    - **JSON**, WKT, WKB

```
• line = arcpy.Polyline(  
•     arcpy.Array([arcpy.Point(2,3),  
•                 arcpy.Point(3,5)])  
•  
• line = [(2,3), (3,5)]  
•  
• line = {  
•     "type": "LineString",  
•     "coordinates": [[2, 3], [3,5]]  
•  
• cursor.insertRow([line])
```

# Geometry and cursors

- Can create geometry in different ways
  - Geometry objects
  - List of coordinates
  - Using other formats
    - **JSON**, WKT, WKB

```
• line = arcpy.Polyline(  
•     arcpy.Array([arcpy.Point(2,3),  
•                 arcpy.Point(3,5)])  
•  
• line = [(2,3), (3,5)]  
• line = '{"paths": [[ [2,3], [3,6] ] ]}'  
• cursor.insertRow([line])
```

# Working with geometry

- **Relational:**

- Is a point within a polygon?

```
• point.within(polygon)
```

- **Topological**

- What is the intersection of two geometries?

```
• poly1.intersect(poly2)
```

- **Others (mid-point, geodesic area)**

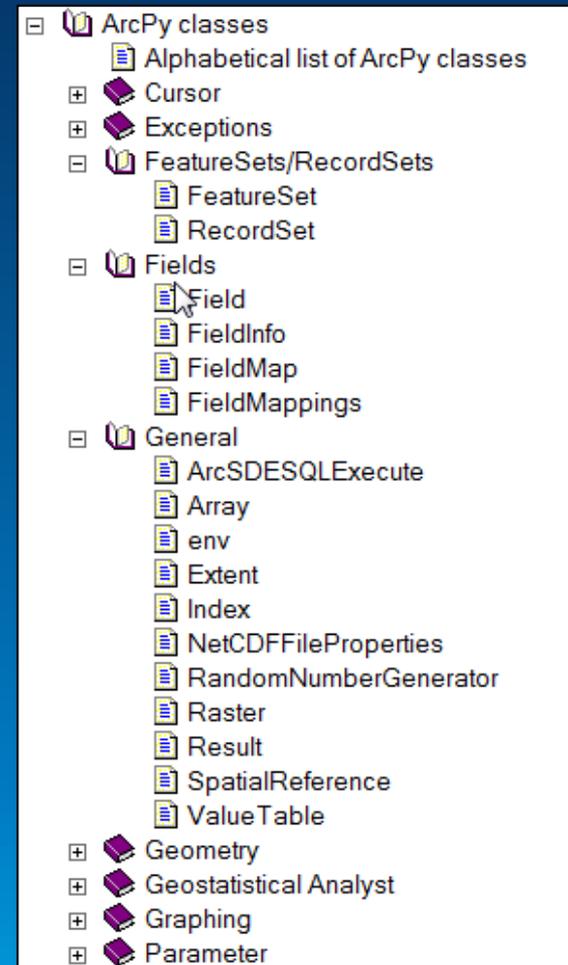
```
boundary  
buffer  
clip  
contains  
convexHull  
crosses  
difference  
disjoint  
distanceTo  
equals  
getArea  
getLength  
getPart  
intersect  
overlaps  
positionAlongLine  
projectAs  
symmetricDifference  
touches  
union  
within
```

**arcpy classes**



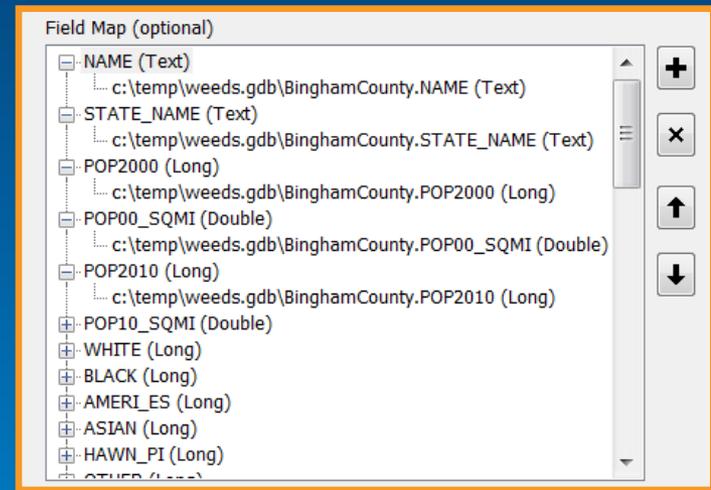
# arcpy classes

- **Classes are used to create objects**
- **Frequently used for...**
  - **Tool parameters**
  - **Working with geometry**



# arcpy Classes

- Most tool parameters are easily defined with a string or number
- Some are not:
  - Spatial reference
  - Field map
- Classes can be used to define these parameters



# Using classes for parameters

- Extent

```
• extent = arcpy.Extent(-117.1, 14.5, -86.7, 32.7)
• arcpy.CreateRandomPoints_management(
•     arcpy.env.workspace,
•     'samplepoints',
•     constraining_extent=extent)
```

- SpatialReference

```
• arcpy.CreateFeatureclass_management(
•     arcpy.env.workspace, 'hydrology', 'POLYGON',
•     spatial_reference=arcpy.SpatialReference(32145))
```

# Cursors



# Cursors

- Use cursors to access records and features

SearchCursor	Read-only
UpdateCursor	Update or delete rows
InsertCursor	Insert rows

- Two varieties
  - 'Classic' cursors
  - 'Data access' cursors (10.1+)

# Cursor mechanics

- **Data access cursors use lists and tuples**
  - Values are accessed by index

```
• cursor = arcpy.da.InsertCursor(table, ["field1", "field2"])  
• cursor.insertRow([1, 10])
```

- **Classic cursors use row objects**
  - Values are accessed by setValue/getValue

```
• cursor = arcpy.InsertCursor(table)  
• row = cursor.newRow()  
• row.setValue("field1", 1)  
• row.setValue("field2", 10)  
• cursor.insertRow(row)
```

## With statements

- **arcpy.da Cursors support with statements**

```
• with arcpy.da.SearchCursor(table, field) as cursor:  
•     for row in cursor:  
•         print row[0]
```

# Cursor performance

- Use only those fields you need

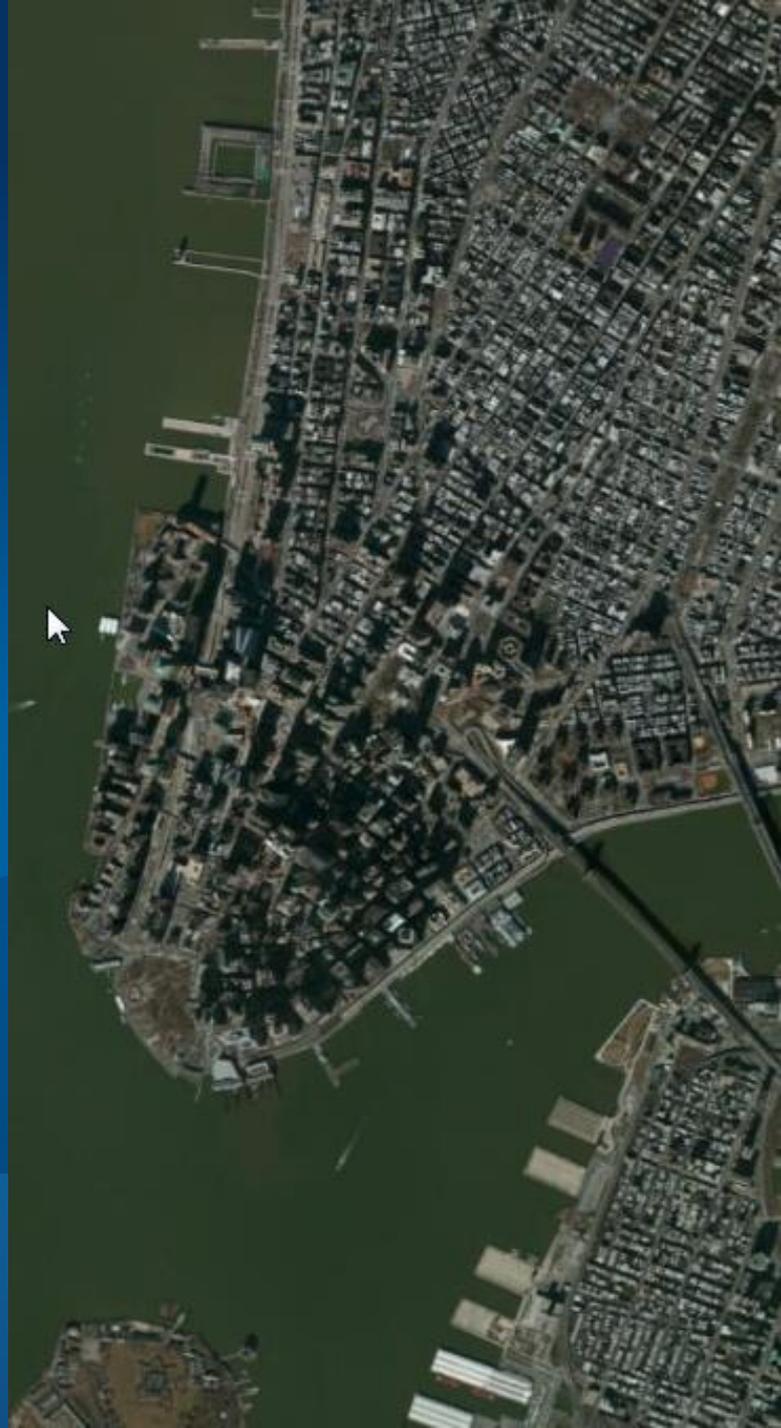
- Use tokens

- Get only what you need
- *Full geometry is expensive*

- **SHAPE@XY** —A tuple of the feature's centroid x,y coordinates.
- SHAPE@**SHAPE@XY** —A tuple of the feature's true centroid x,y coordinates.
- SHAPE@X —A double of the feature's x-coordinate.
- SHAPE@Y —A double of the feature's y-coordinate.
- SHAPE@Z —A double of the feature's z-coordinate.
- SHAPE@M —A double of the feature's m-value.
- SHAPE@JSON — The esri JSON string representing the geometry.
- SHAPE@WKB —The well-known binary (WKB) representation for OGC geometry. It provides a portable representation of a geometry value as a contiguous stream of bytes.
- SHAPE@WKT —The well-known text (WKT) representation for OGC geometry. It provides a portable representation of a geometry value as a text string.
- SHAPE@ —A [geometry](#) object for the feature.
- SHAPE@AREA —A double of the feature's area.
- SHAPE@LENGTH —A double of the feature's length.
- OID@ —The value of the [ObjectID](#) field.

Exercise

# Data Access



# arcpy.mapping



MD Wetlands  
1.png



MD Wetlands  
2.png



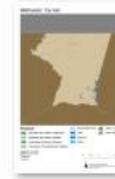
MD Wetlands  
3.png



MD Wetlands  
4.png



MD Wetlands  
5.png



MD Wetlands  
6.png



MD Wetlands  
7.png



MD Wetlands  
8.png



MD Wetlands  
9.png



MD Wetlands  
10.png



MD Wetlands  
11.png



MD Wetlands  
12.png



MD Wetlands  
13.png



MD Wetlands  
14.png



MD Wetlands  
15.png

## What is arcpy.mapping?

- A map scripting environment introduced at 10.0
- Python mapping module that is part of the ArcPy site-package

# What can you do with arcpy.mapping?

- **Manage map documents, layer files, & their contents**
  - Find a layer with data source X and replace with Y
  - Update a layer's symbology in many MXDs
  - Generate reports that lists document information
    - data sources, broken layers, spatial reference info, etc.
- **Automate the exporting and printing of map documents**
- **Automate map production and create map books**
  - Extend Data Driven Pages capabilities

## Referencing Map Documents (MXDs)

- Opening Map Documents (MXD) with `arcpy.mapping`
- Use the `arcpy.mapping.MapDocument` function
- Takes a path to MXD file on disk or special keyword "CURRENT"
- Reference map on disk  
`mxd = arcpy.mapping.MapDocument(r"C:\some.mxd")`
- Get map from current ArcMap session  
`mxd = arcpy.mapping.MapDocument("CURRENT")`

## Referencing Map Documents (MXDs), cont.

- **When using CURRENT**

- Always run in foreground (checkbox in script tool properties)
- Be wary of open conflicts, file contention
- May need to refresh the application

`arcpy.RefreshActiveView()`

- **Limitations and pre-authoring**

- No "New Map" function, so keep an empty MXD available
- Can't create new objects (e.g., north arrow, data frame)

# MapDocument Properties/Methods

- **Save / saveAsCopy**
- **Author, description, dateSaved**
- **ReplaceWorkspaces**
- **dataDrivenPages**

# Map Layers and Data Frames

- The “List” functions

- ListLayers
- ListDataFrames
- Watch the list indexes (you may often forget to use [0])

```
df = arcpy.mapping.ListDataFrames(MXD) [0]
```

- Layer properties

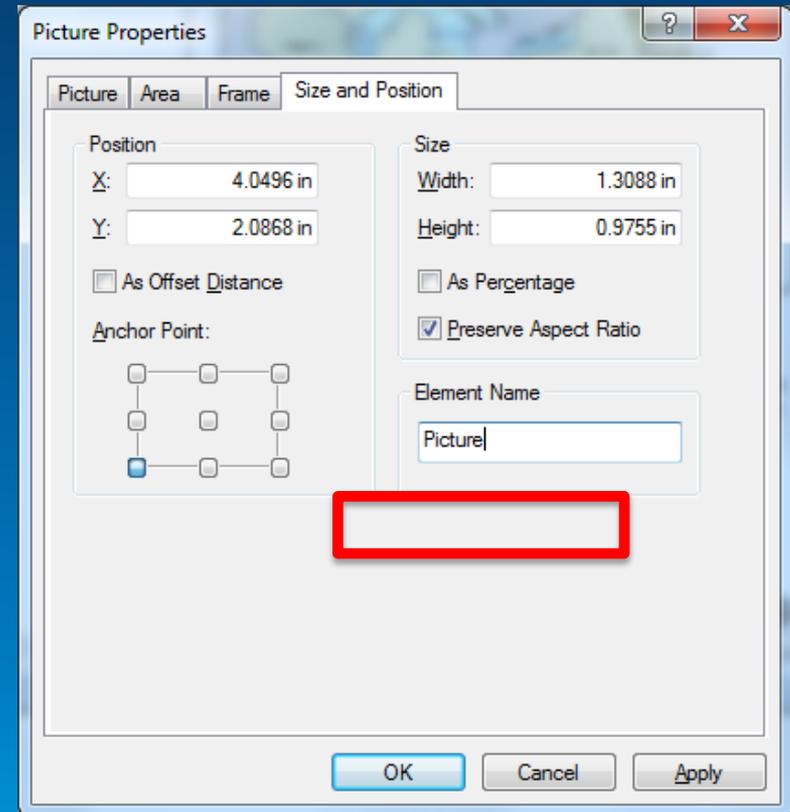
- Common properties are available (e.g., def query, visible)
- All properties can be updated via layer (.lyr) files

- DataFrame properties and methods

- Basic map navigation and settings

# arcpy.mapping for the Page Layout

- **When and what to pre-author for layout manipulation scenarios**
  - Name your layout elements
  - Set the appropriate anchor
  - Cannot add new elements, so pre-author and hide



# arcpy.mapping for Printing and Exporting

- **PDFDocument and DataDrivenPages classes**
- **Export and print functions**
- **Map server publishing**
- **Map book generation**

## FUNCTIONS

```
ExportToAI  
ExportToBMP  
ExportToEMF  
ExportToEPS  
ExportToGIF  
ExportToJPEG  
ExportToPDF  
ExportToPNG  
ExportToSVG  
ExportToTIFF  
PDFDocumentCreate  
PDFDocumentOpen  
PrintMap  
PublishMSDToServer  
...
```

## Updating Data Sources

- Use `arcpy.mapping` for migrating Map Documents and Layer files to new data sources
- Fancier scripts can help mitigate migration pain: SQL syntax changes, field name changes, etc
  - “Updating and fixing data sources with `arcpy.mapping`”
  - <http://esriurl.com/4628>
- Many capabilities:
  - Update all layers in an MXD or specific tables and layers
  - Works with all file and GDB types
  - Update joins and relates
  - Migrate from different workspace types

# Python Toolboxes

```
class Tool(object):
    def __init__(self):
        """Define the tool (tool name)"""
        self.label = "Tool"
        self.description = ""
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definition"""
        params = None
        return params

    def isLicensed(self):
        """Set whether tool is licensed"""
        return True

    def updateParameters(self, parameters):
        """Modify the values and parameters of the tool.
        validation is performed.  This method is called
        whenever the user has changed the tool parameters.
        """
        return

    def updateMessages(self, parameters):
        """Modify the messages created by the tool.
        parameter.  This method is called whenever the
        user changes the tool parameters.
        """
        return

    def execute(self, parameters, messages):
        """The source code of the tool.
        """
        return
```

# Python Toolboxes



- **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**

# Benefits

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

# Toolbox Structure

A tool does 3 types of work:

- Parameters

- Validation

- Source

```
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
```