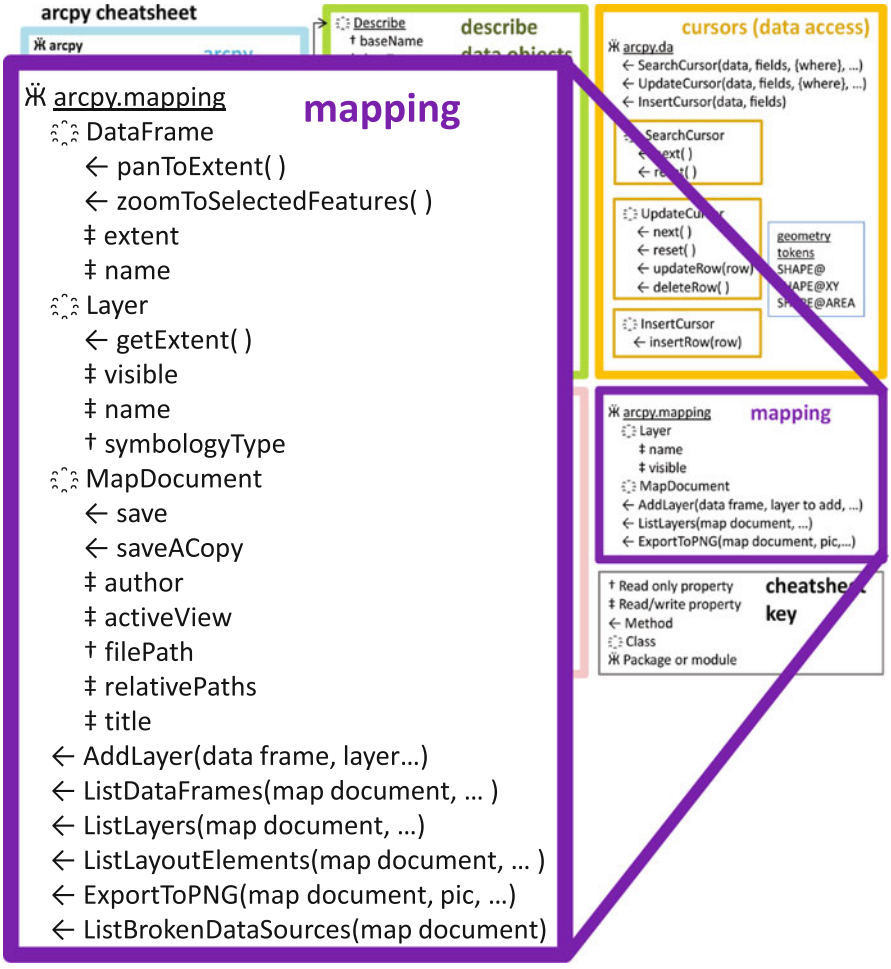
Chapter 24
Mapping Module

Abstract



Suppose you want to take a screen shot of dozens of maps to insert into a report. Or suppose you've been reorganizing your data and you want to ensure that you haven't

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_24

broken the data paths on a large collection of maps. Or suppose you've carefully selected a symbology for a layer and you'd like to apply this symbology to other maps that use that data. With the arcpy package mapping module, you can automate these and other tasks related to map documents and their layers. The mapping module is a script that resides within the arcpy package installed with ArcGIS. This chapter discusses how to use the mapping module to work with map documents, data frames, layers, symbology, and map layout elements.

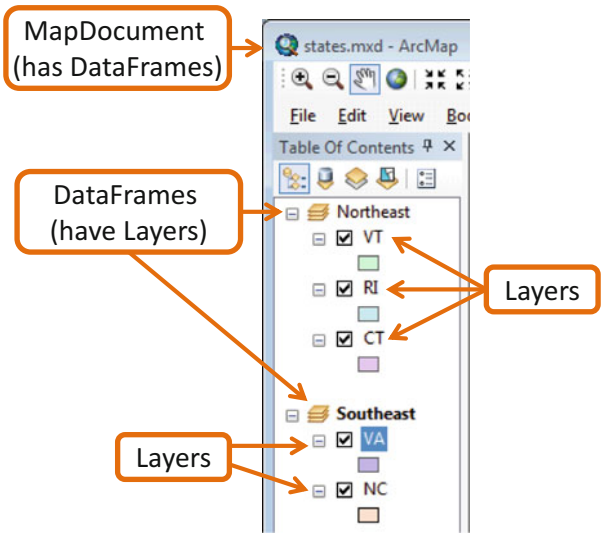
Chapter Objectives
After reading this chapter, you'll be able to do the following:

- Describe the limits and capabilities of the `arcpy mapping` module.
- Explain the hierarchy of map documents, data frames, and layers.
- Use `MapDocument`, `DataFrame`, and `Layer` properties and methods.
- Differentiate between `arcpy mapping` code for the ArcMap Python window and `arcpy mapping` code for stand-alone scripts.
- List the data frames in a map.
- List the layers in a map.
- Save a map document.
- Save a copy of a map document.
- Export a map as an image.
- Move map layers within the table of contents.
- Remove layers from the map.
- Add layers to a map.
- Modify the symbology of a layer.
- Modify map layout elements.

Suppose you want to take a screen shot of dozens of maps to insert into a report. Or suppose you've been reorganizing your data and you want to ensure that you haven't broken the data paths on a large collection of maps. Or suppose you've carefully selected a symbology for a layer and you'd like to apply this symbology to other maps that use that data. With the `arcpy` package `mapping` module, you can automate these and other tasks related to map documents and their layers. The `mapping` module is a script that resides within the `arcpy` package installed with ArcGIS. This chapter discusses how to use the mapping module to work with map documents, data frames, layers, symbology, and map layout elements.

To understand the `mapping` module, it's helpful to keep in mind the hierarchical structure of ArcMap. ArcMap opens map documents, containing data frames, and the date frames have layers. The layers themselves point to geographic data stored on a computer disk. The three central classes in the `mapping` module are the `MapDocument`, `DataFrame`, and `Layer` classes, which correspond to the structure of ArcMap. The mapping classes and functions are designed for managing existing map documents (not for creating new ones) and for automating map publication. The module provides access to the `DataFrame` and `Layer` objects with `ListDataFrames` and `ListLayers` methods that return lists of objects. The properties of these objects can then be modified. There are also a set of methods for exporting maps in image formats or PDF (portable document format).

Some examples in this chapter use the ArcMap Python Window. This is a departure from other chapters which work strictly with stand-alone IDEs. The ultimate goal of this chapter still aims for writing stand-alone scripts, ones that you can run outside of ArcMap. However, because the `mapping` module manipulates map document properties, it's useful to test `mapping` module code statements within an open map document. We'll start in the PythonWin Interactive Window and then move to the ArcMap Python Window as we explore the `MapDocument`, `DataFrame`, and `Layer` classes.



24.1 Map Documents

Manipulating map documents is the main focus of the `mapping` module. For these operations, you first need to create a `MapDocument` object specifying a map. Try the following code in an IDE (such as the PythonWin Interactive Window):

```
>>> myMap = 'C:/gispy/data/ch24/maps/dataSourceExample.mxd'
>>> mxd = arcpy.mapping.MapDocument(myMap)
>>> mxd
<MapDocument object at 0x161618b0[0x16161aa0]>
```

To call functions and classes in user-defined modules, you use dot notation (`moduleName.functionName` and `moduleName.ClassName`). Since the `mapping` is within the `arcpy` package, two dots are needed (`arcpy.mapping.ClassName` or `arcpy.mapping.FunctionName`). This double dot notation is already familiar from ArcGIS environment settings (e.g., `arcpy.env.workspace`). The following code calls a `mapping` function to list the broken data sources in the map document and uses the `MapDocument` object as an argument:

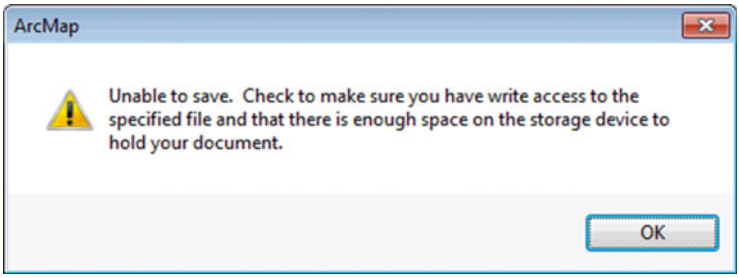


Figure 24.1 A message from ArcMap when the map is locked by Python.

```
>>> arcpy.mapping.ListBrokenDataSources(mxd)
[<map layer u'potHoles'>, <map layer u'township'>]
```

Two layers of 'dataSourceExample.mxd' are broken. Each layer stores an absolute or relative path to its data source. The data source connections are broken when the data are not in the location designated by the path. These data may not have been provided with the map or may not be in the right location relative to the map. Open 'dataSourceExample.mxd' in ArcMap to repair the broken data paths. If you the ran code that created a `MapDocument` object from PythonWin before opening the map in ArcMap, you won't be able to save the changes. You will get an error when you try to save the document (see Figure 24.1). The map document is locked by PythonWin until you delete the `MapDocument` object that points to it (or close the IDE where you created the object). Scripts should use the `del` keyword to delete the object. The following statement deletes the `MapDocument` object, but not the map document itself:

```
>>> del mxd
```

If the map document object has not been deleted when the map is opened, changes can not be saved, even after you run the deletion command. So you would need to close the map document and reopen it after executing the deletion command to successfully save changes to the map document in ArcMap.

Example 24.1 uses the `mapping` module to export an image of the map. The script imports arcpy, sets the workspace, creates a `MapDocument` object, and calls the `ExportToPNG` method. This method captures an image of the 'Layout view' of a map (as in Figure 24.2). The first argument is the `MapDocument` object; The second is the name of the output image file. After this, the script deletes the `MapDocument` object so that the map will remain editable. Try the code with the sample input to see that 'getty_map.png' is created. The exporting methods do not heed the `arcpy` geoprocessing workspace environment setting. If the full path output image file name is not used in the export statement, the image is placed in the map directory, even if the `arcpy` workspace variable is set to a different directory.

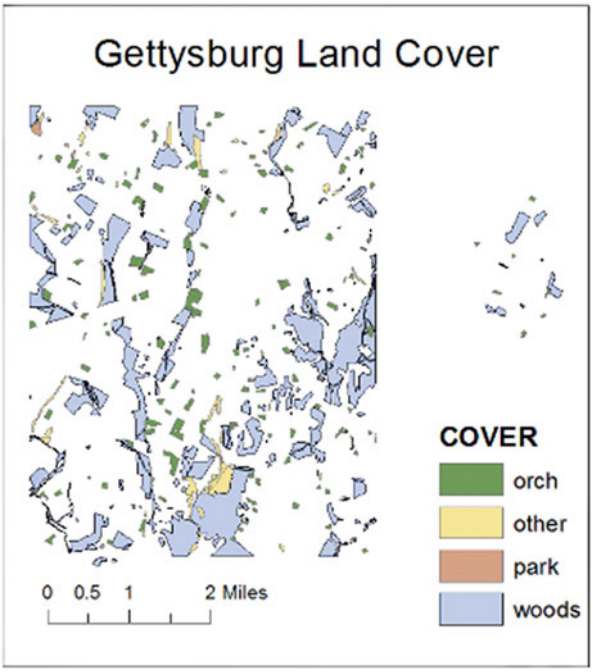


Figure 24.2 Output from Example 24.1, getty_map.png (reduced in size).

Example 24.1: Export the layout view of a map document to a PNG image (Figure 24.2).

```
# mapToPhoto.py
# Purpose: Export the 'Layout view' of a map as a PNG image.
# Usage: fullpath_mxd_filename fullpath_output_png_filename
# Sample input: C:/gispy/data/ch24/maps/landCover.mxd
#               C:/gispy/scratch/getty_map.png
# Note: Portable Network Graphic (PNG) is an image format used for
#       Internet content.
# Many other map export formats are available.

import arcpy, sys

# Full path names of an existing map and an image to create.
mapName = sys.argv[1]
imageName = sys.argv[2]

# Create a MapDocument object.
mxd = arcpy.mapping.MapDocument(mapName)

# Create an image of the map in 'Layout view'
arcpy.mapping.ExportToPNG(mxd, imageName)
print '{0} created.'.format(imageName)

# Delete the MapDocument object.
del mxd
```

24.1.1 Map Name or 'CURRENT' Map

Though it's usually more efficient to use a stand-alone programming IDE, the Python Window inside ArcMap can be quite helpful for learning about the `mapping` module. To open the ArcGIS Python Window, select the Geoprocessing > Python or click the 'Python' button on the standard menu. By using this Python window, you can see modifications to a map document as soon as you make them. However, there's one major difference that you should understand so that you can write scripts that will run outside of ArcMap. The difference pertains to the way you can initialize a `MapDocument` object.

If you're running code in the Python Window embedded in ArcMap, you can refer to this open map as `'CURRENT'`. In Figure 24.3, code is executed in the ArcMap Python scripting window. When `'CURRENT'` is used to create the `MapDocument` object, the filePath is set to 'C:\\gispy\\data\\ch24\\maps\dataSourceExample.mxd' because 'dataSourceExample.mxd' is the map that ArcMap is displaying.

Using `'CURRENT'` works only when you're running code within ArcMap; Scripts run outside of ArcMap do not have a current map defined, even if ArcMap is simultaneously open with a map displayed. Run the same code in an IDE outside of ArcMap, and a `RuntimeError` exception is thrown:

```
>>> mxd = arcpy.mapping.MapDocument('CURRENT')
RuntimeError: Object: CreateObject cannot open map document
```

Instead of referring to the map as `'CURRENT'` in stand-alone scripts, you must use the name of the map. In fact, you must specify the path of the map document (even if it's in your workspace). Otherwise, an error is thrown:

```
>>> arcpy.env.workspace = 'C:/gispy/data/ch24/maps/'
>>> mapName = 'landCover.mxd'
>>> mxd = arcpy.mapping.MapDocument(mapName)
AssertionError: Invalid MXD filename.
```

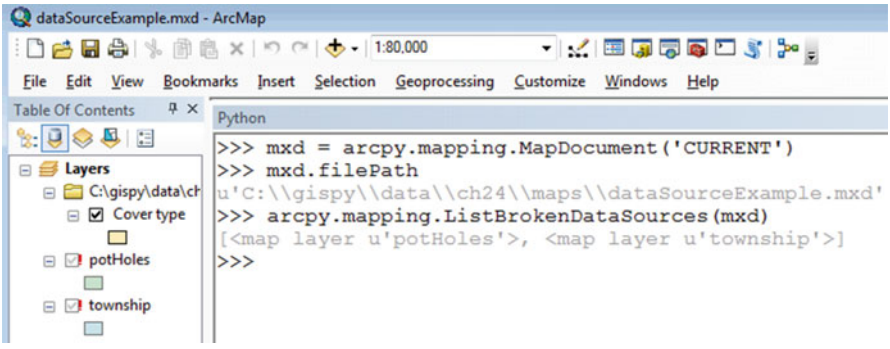


Figure 24.3 In the ArcMap Python scripting window, `'CURRENT'` can be used instead of a file name for the open map document.

The `MapDocument` class does not use the `arcpy` workspace environment to search for its input, so you need to add the map's path to its name:

```
>>> # File name of an existing map
>>> mapName = arcpy.env.workspace + '/landCover.mxd'
>>> # Create a MapDocument object.
>>> mxd = arcpy.mapping.MapDocument(mapName)
```

In stand-alone scripts designed to be run outside of ArcMap, you must specify the full file name of the map.

> `'CURRENT'` only works for code running inside ArcMap.

One other way to run code within ArcMap, is to use an ArcGIS Script Tool. You can use `'CURRENT'` within an ArcGIS Script Tool, with a couple of caveats. First, by default, ArcGIS Script Tools run in the background. Running a process in the background is just like running a stand-alone script. In order to use `'CURRENT'` in a Script Tool, you have to set the checkbox in the Script Tool properties to run the tool in the foreground. This connects the running code with ArcMap in the same way the ArcGIS Python Window does. Second, when you are operating on an open map (the `'CURRENT'` map), you need to look out for data control conflicts. Locking issues can arise if you're working with data that is open in ArcMap, so there may be certain operations you can't do.

Examples in the next section use the ArcMap Script Window. The keyboard shortcut in PythonWin for efficiently retrieving previous commands is ctrl + UpArrow. In the ArcGIS Python window, just use the UpArrow key to scroll through command history.

As you experiment with code in this environment, you may have to call the `RefreshActiveView` and `RefreshTOC` functions to see the updates appear within your map. Some changes appear automatically, but others require a refresh command to update the display. These are `arcpy` methods, not `mapping` module methods, meaning only one dot is needed. Refreshing the view is only needed when working within the `arcpy` Python window. These statements are usually not needed in stand-alone scripts. Refreshing takes time, so remove 'Refresh' statements from stand-alone scripts, where possible.

24.1.2 `MapDocument` Properties

You may already be familiar with the map documents properties you see when you click on the File menu in ArcMap and select 'Map Document Properties…'. Figure 24.3 uses the `mxd.filePath` to check the file path of the map document.

Python can access additional map document properties, such as `author`, `activeView`, and `title`. To see this in action, open 'states.mxd' in ArcMap, open the ArcMap Python window, and type Python commands in this window.

```
>>> mxd = arcpy.mapping.MapDocument('CURRENT')
```

When you type mxd followed by the dot, you see a menu of `MapDocument` properties and methods. You can select these from the menu with the up and down arrows and tab completion. Some `MapDocument` properties control metadata, such as the map's author:

```
>>> mxd.author
u'Me'
>>> mxd.author = 'Wonderful me!'
>>> mxd.author
u'Wonderful me!'
```

Other properties pertain to viewing the map. The buttons in the bottom left corner of the map modify the active view ('Data' or 'Layout'). The 'Layout' view displays the map surrounds (title, legend, scale, etc.) along with the data. ArcMap will switch between these views if you modify the `activeView`:

```
>>> mxd.activeView
u'Southeast'
>>> mxd.activeView = 'PAGE_LAYOUT'
```

The value of the `activeView` property is either `'PAGE_LAYOUT'` or the name of the active data frame in the data view. The map title can also be changed:

```
>>> mxd.title
u'Hey hey'
>>> mxd.title = 'Eastern US'
>>> arcpy.RefreshActiveView()
```

The `RefreshActiveView` method updates the title on the map. Again, this is an `arcpy` method, not a mapping method, so only one dot is needed.

Another `MapDocument` property, `relativePaths`, is important for portability. Maps can use a relative path or an absolute path to the data sources. When `relativePaths` is set to `True`, the map and data can be moved to a new location without breaking the links to the data, as long as the data stay in the same location, relative to the map. The `relativePaths` property can be set to `True` or `False`:

```
>>> mxd.relativePaths
True
>>> mxd.relativePaths = False
```

The author, activeView, title, and relativePaths can be changed within Python because these are 'read and write' properties. Some MapDocument properties are 'read only'. For example, the filePath property can only be read. Trying to change its value throws a NameError exception:

```
>>> mxd.filePath
u'C:\\gispy\\data\\ch24\\maps\\states.mxd'
>>> mxd.filePath = 'C:\\gispy\\data\\ch24\\maps\\US.mxd'
NameError: The attribute 'filePath' is not supported on this
instance of MapDocument.
```

You can check the value of a 'read only' property, but you can't change the value. You can't change the file path in the 'File > Map Document Properties…' window either. Search the ArcGIS Resources site for 'MapDocument (arcpy.mapping)' for a list of the MapDocument properties and their read/write capabilities.

24.1.3 Saving Map Documents

There are two MapDocument methods for saving changes to map documents, saveACopy and save. The saveACopy method saves a copy. It takes one argument, the new map name. If a full path is not specified, the copy will be saved in the current workspace:

```
>>> mxd.saveACopy('modifiedMap.mxd')
```

To automatically launch ArcMap and view the saved copy, you can use the os module startfile command, which opens (or tries to open) the document passed in as an argument with the default program associated with that file type. Since this is an os command, not an arcpy command, the arcpy workspace is not used; therefore, you may need to provide the full path file name. The following example launches ArcMap and loads 'landCover.mxd':

```
>>> import os
>>> os.startfile('C:/gispy/data/ch24/maps/landCover.mxd')
```

The save method does not require any arguments, but use it with care, as it will overwrite the existing map, even if the overwriteOutput environment variable is set to False:

```
>>> mxd.save()
```

Notice that there is no method for creating a new map document. The `mapping` module does not allow you to create new maps from scratch. You have to start with an existing map. If you desire to create maps entirely programmatically, you can create an empty map to use as a template, save a copy of the map (instead of overwriting the template), and then add content.

24.2 Working with Data Frames

Since data frames belong to the map document, a map document must be specified to access them. Continuing the ArcMap Python window session associated with the 'states.mxd' map, we'll use our existing `MapDocument` object, `mxd`, to get a list of data frames in the map. The `ListDataFrames` method requires one argument, a `MapDocument` object and it returns a Python list of the `DataFrame` objects in the map document:

```
>>> dfs = arcpy.mapping.ListDataFrames(mxd)
>>> dfs
[<DataFrame object at 0x27b7b950[0x216e2a98]>, <DataFrame object at
0x27b7b8f0[0x216e2a48]>]
```

Each `DataFrame` object has properties. You can use zero-based indexing to access individual `DataFrame` objects. Try the following code to get the first `DataFrame` object in the list and use some `DataFrame` properties:

```
>>> # Get the first data frame.
>>> df = dfs[0]
>>> df.name
u'Northeast'
>>> # Get the second data frame.
>>> df2 = dfs[1]
>>> df2.name
u'Southeast'
>>> df.description
u''
>>> df2.description = "My very cool data frame"
>>> df2.description
u'My very cool data frame'
>>> df2.displayUnits
u'Feet'
```

The `extent` property is an `Extent` object, which stores the corners of the data frame's boundaries as `XMin`, `YMin`, `XMax`, and `YMax` properties:

```
>>> ext = df2.extent
>>> ext.YMin
29.99437527717719
```

You can also zoom to selected features. The following code selects one of the layers, using a Data Management tool and zooms to the selected layer:

```
>>> arcpy.SelectLayerByAttribute_management('VA')
>>> df2.zoomToSelectedFeatures()
```

Since `dfs` is a Python list, you can also navigate it with a `FOR`-loop:

```
for df in dfs:
    # Print the name of the current data frame.
    print df.name
```

`DataFrame` objects can be used to access data layers. We'll next look at `Layer` object properties and some examples that use `DataFrame` objects and `Layer` objects together.

24.3 Working with Layers

The `mapping` module has a `ListLayers` method for listing the layers in a map. It requires one argument, a `MapDocument` object and it returns a list of `Layer` objects. `Layer` objects have properties that Python can access. Continue typing in the ArcMap Python window of the 'states.mxd' map to try the following code statements:

```
>>> layers = arcpy.mapping.ListLayers(mxd)
>>> myLayer = layers[0]
>>> myLayer.dataSource
u'C:\\gispy\\data\\ch24\\USstates\\VT.shp'
>>> myLayer.isFeatureLayer
True
>>> myLayer.isRasterLayer
False
```

The `getExtent` method returns an `Extent` object. This object provides extent boundaries for the `Layer` object, which may differ from the data frame's boundaries:

```
>>> e = myLayer.getExtent()
>>> e
```

```
<Extent object at 0x146d41d0[0x1472a9e0]>
>>> e.YMin
1219145.8956193451
```

You can zoom to a layer's extent by setting the data frame's extent to the layer's extent:

```
>>> df.extent = myLayer.getExtent()
```

You can also select a set of features and set the data frame extent to the value returned by `getSelectedExtent()`, to zoom to those features.

The `ListLayers` method can return all of the layers in the map or a subset of the layers. A `MapDocument` object is the only required argument. If this is the only argument used, all of the layers are listed:

```
# List ALL of the Layer objects
>>> layers = arcpy.mapping.ListLayers(mxd)
>>> for myLayer in layers:
...     print myLayer.name
...
VT
RI
CT
VA
NC
```

This method also has two optional arguments, a wild card and a `DataFrame` object. The wild card argument can restrict the results based on a substring (just like the `arcpy` methods for listing datasets) and the `DataFrame` object can restrict the list to a particular data frame. To specify a data frame, but not utilize the wild card, you need to use a placeholder (`'*'`) for the wild card.

```
>>> # Get a list of DataFrame objects.
>>> dfs = arcpy.mapping.ListDataFrames(mxd)

>>> # Get the first DataFrame object.
>>> df = dfs[0]

>>> # Get a list of Layer objects in this data frame.
>>> layers = arcpy.mapping.ListLayers(mxd, '*', df)
>>> for myLayer in layers:
...     print myLayer.name
...
VT
RI
CT
```

The mapping module methods that use Layer objects to manipulate layers (MoveLayer, RemoveLayer, AddLayer, and InsertLayer) are discussed next.

24.3.1 Moving, Removing, and Adding Layers

The mapping module can move, remove, and add layers. Because layers are nested within data frames (which are within a map document), to use any of these layer methods, you must first instantiate MapDocument and DataFrame objects. These are then used to specify the Layer object. To see how it works, open 'layerManipExample1.mxd'. In this map, a layer containing points is hidden by the polygon layers. To make the points visible, we'll use the MoveLayer method to move this layer to the top of the table of contents. The MoveLayer method allows a layer to be moved before or after a reference layer within the same data frame. To try this, begin by initializing a MapDocument object, a DataFrame object, and a list of Layer objects in the ArcMap Python Window:

```
>>> mxd = arcpy.mapping.MapDocument('CURRENT')
>>> dfs = arcpy.mapping.ListDataFrames(mxd)
>>> df = dfs[0]
>>> lyrs = arcpy.mapping.ListLayers(mxd)
```

Now that we have a list, we need to select the specific Layer objects to use for calling MoveLayer. Moving the 'centers' layer before the 'cover' layer brings it to the top of the table of contents, so we'll use the 'cover' layer as the reference layer. We need to use numbers (not names) to index into the list. We could find the correct indices by looping through the list and checking against the names, but here we'll just hard-code the numbers. The ListLayers function returns the layers in the order they appear in the table of contents from top to bottom. The layer to move is the third layer (index 2) and the reference layer is the first layer (index 0):

```
>>> layerToMove = lyrs[2]
>>> layerToMove.name
u'centers'
>>> referenceLayer = lyrs[0]
>>> referenceLayer.name
u'cover'
```

The MoveLayer method requires a DataFrame object and the two Layer objects as arguments. The optional fourth parameter specifies the position relative to the reference layer. Enter the following code to move the 'centers' layer to the top of the table of contents:

```
>>> arcpy.mapping.MoveLayer(df, referenceLayer, layerToMove, 'BEFORE')
```

The code for removing and adding layers is similar. The difference comes in specifying the `Layer` objects (layers to be removed or added). Now that you have seen `MoveLayer` work in the ArcMap Python Window, you can become familiar with code for manipulating layers in stand-alone scripts, as shown in Examples 24.2 and 24.3.

Example 24.2 removes the first layer in the first data frame. To remove data from an open map by hand, you first need to right click on the layer in the table of contents and then you can select 'Remove'. Similarly, to remove a layer with a script, you first need to create a `Layer` object that is pointing to the layer you want to remove. Then you can call the `RemoveLayer` method on this object. The map in Example 24.2, 'layerManipExample2.mxd', has three layers, all in one data frame. The code gets a `MapDocument` object, lists the `DataFrame` objects, and lists the `Layer` objects from the first data frame. The script sets `layerToRemove` to the first `Layer` object in the list and then calls the `RemoveLayer` method. This method has two required arguments, a `DataFrame` object (for the data frame where the layer resides) and a `Layer` object (referencing the one to be removed). The script finally saves a copy of the map that now has two layers and releases the lock on the map by deleting the `MapDocument` object. The copy of the map only has the two remaining layers.

Example 24.2: Remove the first layer from a map.

```
# removeLayers.py
# Purpose: Remove the first layer in the table of contents.
# Input: No arguments required.
import arcpy

# Get a MapDocument object.
mxdName = 'layerManipExample2.mxd'
mapPath = 'C:/gispy/data/ch24/maps/'
mxd = arcpy.mapping.MapDocument(mapPath + mxdName)

# Get a list of the DataFrame objects.
dfs = arcpy.mapping.ListDataFrames(mxd)

# Get the first DataFrame object.
df = dfs[0]

# Get a list of Layer objects in this data frame.
lyrs = arcpy.mapping.ListLayers(mxd, '' , df)

# Get the first Layer object.
layerToRemove = lyrs[0]

# Remove the layer.
arcpy.mapping.RemoveLayer(df, layerToRemove)

# Save a copy of the map.
copyName = 'C:/gispy/scratch/' + mxdName[:-4] + '_V2.mxd'
mxd.saveACopy(copyName)
```

```
# Delete the MapDocument object to release the map.
del mxd
```

Example 24.3 adds data to a map. To add data to a map by hand, you click on the 'Add Data' button in the standard toolbar and browse to the data to add. To add data with a script, you need to specify the data path and create a `Layer` object pointing to the data. This layer is then passed into the AddLayer method. For adding layers, you again need to get a `MapDocument` and `DataFrame` objects to specify where the layer will be added. Since the layer doesn't exist yet, instead of pointing to an existing layer, you need to create a `Layer` object by calling the `Layer` class. The `Layer` class takes the name of a data file as an argument and returns a `Layer` object. Example 24.3 creates a `Layer` object named `layerObj` by pointing to the `fileName` variable. Next, it calls `AddLayer`, passing in two required arguments, `DataFrame` and `Layer` objects.

By default the `AddLayer` method adds a layer in the same way the 'Add Data' button works, using an automatic arrangement scheme. You can also specify 'TOP' or 'BOTTOM' of the data frame using the third optional argument. The `InsertLayer` method is similar to the `AddLayer` method, but it gives you tighter control on the placement by requiring a reference layer (like `MoveLayer`).

Example 24.2 and 24.3 manipulate vector data (a shapefile); The procedures for removing and adding vector and raster files are the same.

Example 24.3: Adding a shapefile layer to a map.

```
# addLayer.py
# Purpose: Add a data layer to a map.
# Input: No arguments required.
import arcpy

# Initialize data variables.
arcpy.env.workspace = 'C:/gispy/data/ch24/maps/'
fileName = '../USstates/MA.shp'
mapName = 'layerManipExample3.mxd'

# Instantiate MapDocument and DataFrame objects.
mxd = arcpy.mapping.MapDocument(arcpy.env.workspace + '/' + mapName)
dfs = arcpy.mapping.ListDataFrames(mxd)

# Get the first data frame.
df = dfs[0]

# Instantiate a Layer object.
layerObj = arcpy.mapping.Layer(fileName)

# Add the new layer to the map.
arcpy.mapping.AddLayer(df, layerObj)
# Save a copy of the map.
```

```
copyName = 'C:/gispy/scratch/' + mapName[:-4] + '_V2.mxd'
mxd.saveACopy(copyName)

# Delete the MapDocument object to release the map.
del mxd
```

24.3.2 Working with Symbology

The `mapping` module provides some capability for changing layer symbology using another layer management method, `UpdateLayer`. To manually change the symbology for a map layer, right click on a map layer name in the table of contents, select 'Properties' and click on the 'Symbology' table. Select a symbology type from the list on the left and then you can update the properties for your visualization preferences. For example, if you select 'Graduated colors' under 'Quantities', you can select a value field, a number of classes, and a color ramp. The available properties depend on the symbology type, e.g., the 'Graduated Symbols' type has a symbol size property but no color ramp, since it varies glyph size, instead of color, for each class. The `mapping` module allows you to modify some of these properties.

Python can only modify the properties associated with certain symbology types. A `Layer` object has a `symbologyType` property which has one of following values: `'GRADUATED_COLORS'`, `'GRADUATED_SYMBOLS'`, `'UNIQUE_VALUES'`, `'RASTER_CLASSIFIED'`, or `'OTHER'`. Each of the first four values has an `arcpy` class and properties that can be modified in Python. The `'OTHER'` symbology type is a catch-all for the remaining symbology types. Python can not modify the symbology of layers that have a symbology type of `'OTHER'`.

Modifying symbology in Python starts by getting the `Layer` object and checking the `symbologyType`, which tells you which symbology properties apply. To see how it works, open 'symbologyExample.mxd' in ArcMap and try the following code in the ArcMap Python window:

```
>>> mxd = arcpy.mapping.MapDocument('CURRENT')
>>> dfs = arcpy.mapping.ListDataFrames(mxd)
>>> df = dfs[0]
>>> lyrs = arcpy.mapping.ListLayers(mxd)
>>> layerToModify = lyrs[0]
>>> layerToModify.symbologyType
u'OTHER'
```

Data files that have a '.lyr' file extension are referred to as 'layer files'. These files store the symbology for a dataset. When you add a layer file to a map, it uses the stored `symbologyType`; Otherwise, when data is added to a map, the default `symbologyType` is `'OTHER'`. For example, when a shapefile is added to a map,

it is assigned the default symbology type. Depending on its `symbologyType`, a `Layer` object can have a `symbology` class.

```
>>> layerToModify.symbology
NameError: The attribute 'symbology' is not supported on this
instance of Layer.
```

The `symbology` class is not supported for `layerToModify`. Map layers with a `symbologyType` of `'OTHER'` do not support the `symbology` class. Only map layers with one of the four other symbology types support the symbology class. You can't change the `symbologyType` of a layer using an assignment statement, because this property is read-only:

```
>>> layerToModify.symbologyType = 'GRADUATED_COLORS'
RuntimeError: LayerObject: Set attribute symbologyType does not exist
```

Instead, modifying the `symbologyType` can be done using a previously created '.lyr' file. A '.lyr' file can act as training data that has the desired `symbologyType`. For our example, we'll use the sample dataset, 'gradColorNE.lyr', from the 'C:\gispy\data\ch24\symbolTraining' directory. This dataset has a graduated color symbology. To apply the symbology from this file, use the `UpdateLayer` method, as in the following code:

```
>>> srcLay = 'C:/gispy/data/ch24/symbolTraining/gradColorsNE.lyr'
>>> srcLayObj = arcpy.mapping.Layer(srcLay)
>>> arcpy.mapping.UpdateLayer(df, layerToModify, srcLayObj)
>>> layerToModify.symbologyType
u'GRADUATED_COLORS'
```

Now we can modify graduated color symbology properties, such as the value field for calculating the colors and the number of color classes:

```
>>> layerToModify.symbology.valueField
u'Category'
>>> layerToModify.symbology.valueField = 'AVE_FAM_SZ'
>>> arcpy.RefreshActiveView()
>>> layerToModify.symbology.numClasses
5
>>> layerToModify.symbology.numClasses = 3
>>> arcpy.RefreshActiveView()
```

Once the view is refreshed, you can see the resulting map that visualizes the average family size in three classes (Figure 24.4). Though Python's access to symbology is somewhat limited, by developing a set of training layers, you can modify a number of visual properties automatically.

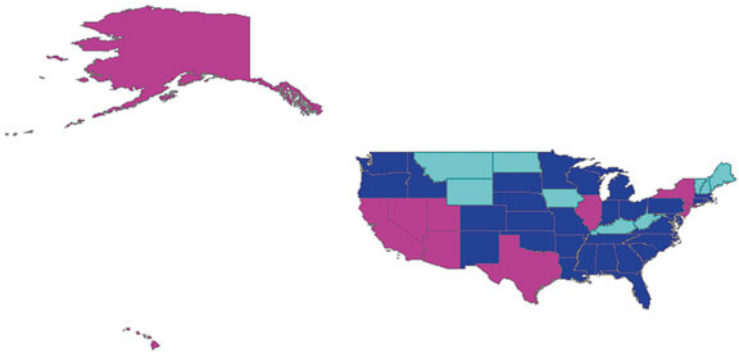


Figure 24.4 Graduate color symbology for average family size.

24.4 Managing Layout Elements

Map layout elements, such as legends, scale bars, and north arrows can also be repositioned and resized with the `mapping` module. The properties of a layout element can be changed manually in ArcMap by double-clicking on the element to launch the properties dialog. The element properties vary, depending on the type of element. Some (though not all) of these properties can be modified with a script.

The `mapping` module has a `ListLayoutElements` method to list the layout elements in the map. This returns a list of element objects, which have `name`, `type`, `position`, and `size` properties. Depending on the layout element, the object can have other properties or methods. For example, the title text can be modified using a `text` property, the style of a layer in the legend can be updated using the `updateItem` method, and a `sourceImage` property can be modified for picture elements. The 'layoutElemsExample.mxd' map document has layout elements visible to the left of the map in layout view. To show how this works, we'll modify the title element. First, close any ArcMap Python windows and then open 'layoutElemsExample.mxd' in ArcMap. Next, try the following code in the ArcMap Python window to get a list of layout element objects and print their names:

```
>>> mxd = arcpy.mapping.MapDocument('CURRENT')
>>> elems = arcpy.mapping.ListLayoutElements(mxd)
>>> for e in elems:
...     print e.name

Text Box
Scale Text
Alternating Scale Bar
North Arrow
Legend
Title
Layers
```

The loop prints the name of each element using the `name` property. Double-click on the title element in ArcMap to open its properties window and look at the 'Size and Positions' tab. The 'Element Name' is set to 'Title'. To get a specific element object, you need to know the element's name. For example, since the title layout element has the name `'Title'`, you can loop through the list to get this object as follows:

```
>>> for e in elems:
...     if 'Title' in e.name:
...         title = e
>>> title
<TextElement object at 0x2a1e8330[0x251d8a20]>
```

Alternatively, you can get a specific element by using the optional wild card parameter when you call the `ListLayoutElements` method, though this requires you to know both the name and the element type. The first parameter for this method, a `MapDocument` object, is required. The second parameter, the element type, is optional but will not accept a place holder, so you must specify one of the six predefined element types, `'DATAFRAME_ELEMENT'`, `'GRAPHIC_ELEMENT'`, `'LEGEND_ELEMENT'`, `'MAPSURROUND_ELEMENT'`, `'PICTURE_ELEMENT'`, or `'TEXT_ELEMENT'`. The third argument is a wild card for the element name. As an example, the north arrow is a `'MAPSURROUND_ELEMENT'` type element and it is named `'North Arrow'`. The following code uses these two pieces of information and indexes into the list to get the north arrow element:

```
>>> arrow = arcpy.mapping.ListLayoutElements(mxd,
     'MAPSURROUND_ELEMENT', '*Arrow*')[0]
>>> arrow.name
u'North Arrow'
```

Returning to our `title` variable, this is pointing to a text element object which has properties such as `elementWidth`, `elementHeight`, `elementPositionX`, `elementPositionY`, `fontSize`, `name`, `text`, and so forth. The following code uses the `text` and `fontSize` properties to change the map's title and refreshes the view to show the changes:

```
>>> title.text
u'Map_Title'
>>> title.text = 'USA'
>>> arcpy.RefreshActiveView()
>>> title.fontSize
42.0
>>> title.fontSize = 72
>>> arcpy.RefreshActiveView()
```

Currently, the map title is sitting in a staging area to the left of the main map view. We can move it to the center of the map using the position and dimensions of its data frame. `DataFrame` objects have `elementWidth`, `elementHeight`, `elementPositionX`, and `elementPositionY` properties which we can enlist for this task. Use the following code to get the `DataFrame` object for the map (which has only one data frame) and move the title to the center of the map:

```
>>> dfs = arcpy.mapping.ListDataFrames(mxd)
>>> df = dfs[0]
>>> title.elementPositionX = df.elementPositionX + \
     (df.elementWidth*0.5) - (title.elementWidth*0.5)
>>> arcpy.RefreshActiveView()
```

The title is now in the center (horizontally) of the frame. To move it to the top of the frame, modify the y-position:

```
>>> title.elementPositionY = df.elementPositionY + \
     df.elementHeight - title.elementHeight
>>> arcpy.RefreshActiveView()
```

You can not create new layout elements with the `mapping` module. However, you can use a template map document with map elements in a staging area, outside the data frame. Only the elements inside the data frame will be visible when the map is exported. For example, only the title of 'layoutElemsExample.mxd' has been moved inside the data frame. The rest of the elements are in a staging area, so they are not visible on export. Export 'layoutElemsExample.mxd' with the following code and confirm that the resulting document shows the title, but no other layer elements:

```
>>> myExport = 'C:/gispy/scratch/noSurrounds.pdf'
>>> arcpy.mapping.ExportToPDF(mxd, myExport)
```

For a template map document, you could start with something like the sample map, 'layoutElemsExample.mxd'. You may want to remove the data, move the title back to the staging area and add other elements (such as pictures or text boxes) in the staging area. Getting a layout element to modify its properties requires you to know the element name. When you add an element to a map document, it gets a name. However, for some elements, the default name is an empty string. For this reason, you need to make sure each element in your map template has a meaningful name. For example, add a text element to 'layoutElemsExample.mxd' (Insert > Text), double click on it and view the 'Size and Position' table The 'Element Name' box is empty; rename it as 'Text1', so that you can refer to this element by name. Once you have named all of the elements and placed them in a staging area, the template is ready. A script can save a copy of the template map, and then automatically add data

> **Note** New layout elements can not be created with the `mapping` module, but a pre-prepared map can be used as a template for automating layout element manipulation.

to the map and modify the symbology and map surrounds to streamline map production workflow.

24.5 Discussion

In this chapter, we used the `arcpy mapping` module to modify map, data frame, and layer properties, to add, move, and delete layers, and to modify symbology and layout elements. These operations involve working with map documents, data frames, layers, symbology, and layout element object properties and methods. Most mapping scripts begin by creating a `MapDocument` object, listing the `DataFrame` objects, and listing the `Layer` objects. To explore these objects further, reference the `'CURRENT'` map document in the ArcGIS Python window. When porting code to stand-alone applications, be sure to reference the full path file name of the map document and be aware of data locking conditions. This chapter demonstrated many of the `mapping` module capabilities, but not all. If you are working with temporal data attributes, you may want to investigate additional `mapping` module functionality, such as the `DataFrameTime` and `LayerTime` classes that enable dynamic map displays. One final tip, when you're using the `mapping` module extensively, it may be helpful to print the alphabetical lists of `mapping` classes, functions, and constants found in the ArcGIS Resources help pages.

24.6 Key Terms

`MapDocument` objects
`DataFrame` objects
`Layer` objects
`'CURRENT'` vs. map name
`RefreshActiveView` method
Layer file (.lyr extension)
`SymbologyType` property
`Layer` object symbology class
Layout element objects

24.7 Exercises

1. **exportToJPG.py** Write a script that exports a map to a JPEG image with the same base name as the map and places the image in 'C:/gispy/scratch'. The script should take one argument: the full path filename of a map document file.

   Example input: C:/gispy/data/ch24/maps/states.mxd

   Example output:
   ```
   >>>> C:/gispy/scratch/states.jpg created.
   ```

2. **listLayers.py** Write a script that takes the full path file name of a map document and prints the '.mxd' file path as well as its data frame and layer names of each layer. The script should take just one argument, the full path filename of the '.mxd' file. Use nested looping and format the output with tabs to achieve the indentation shown in the example.

   Example input: C:/gispy/data/ch24/maps/states.mxd

   Example output:
   ```
   >>> Map: C:/gispy/data/ch24/maps/states.mxd
   Frame 0: Northeast
       Layer 0: VT    C:\gispy\data\ch24\USstates\VT.shp
       Layer 1: RI    C:\gispy\data\ch24\USstates\RI.shp
       Layer 2: CT    C:\gispy\data\ch24\USstates\CT.shp
   Frame 1: Southeast
       Layer 0: VA    C:\gispy\data\ch24\USstates\VA.shp
       Layer 1: NC    C:\gispy\data\ch24\USstates\NC.shp
   ```

3. **addRaster.py** Write a script that adds a raster layer to an existing map and saves the results as a copy of the mxd in 'C:/gispy/scratch'. The script should take two arguments: the full path filename of the input '.mxd' file and the full path file name of the raster data to be added.

   Example input:
   C:/gispy/data/ch24/maps/testAdd.mxd C:/gispy/data/ch24/otherData/getty_rast

   Example output:
   ```
   >>> C:/gispy/scratch/testAdd.mxd created.
   ```

   The output map should have the same basename as the input map, but there should now be a 'getty_rast' layer in the table of contents.

4. **removeRastLayers.py** Write a script that removes any raster layers from a map and saves the results as a copy of the mxd in 'C:/gispy/scratch'. The script should take one argument, the full path filename of the '.mxd' file. The output map should have the same basename, but will have no rasters as layers in the table of contents.

Example input: C:/gispy/data/ch24/maps/testRemove.mxd

Example output:

```
>>> getty_rast_discrete layer removed.
getty_rast layer removed.
pic1.JPG layer removed.
C:/gispy/scratch/testRemove.mxd created.
```

5. **activeZoom.py** Practice using `mapping` properties and objects. Write a script that performs the following steps:

   Step 1: Change the active view of a map to the data frame with the name specified by the user.
   Step 2. Set only the layers in the active data frame to visible (This can by done by setting the visibility of all layers to `False` and then turning on only the needed layers.)
   Step 3. Zoom to the extent of the first layer in the active data frame.
   Step 4. Export a PDF of the map with the same base name as the data frame and in the same directory as the map document.

   Do not save the changes to the map document within the script. The script should take three arguments, the full path filename of a map document, the name of a data frame in that document, and an output directory for the PDF.

   Example input:
   C:/gispy/data/ch24/maps/USregions.mxd Central C:/gispy/scratch

   Example output:

```
>>> Map active view set to Central frame
southAtlantic layer hidden.
midAtlantic layer hidden.
newEngland layer hidden.
eastNorthCentral layer hidden.
eastSouthCentral layer hidden.
westSouthCentral layer hidden.
westNorthCentral layer hidden.
mountain layer hidden.
pacific layer hidden.
eastNorthCentral layer visible.
eastSouthCentral layer visible.
westSouthCentral layer visible.
westNorthCentral layer visible.
Data frame extent set to the extent of eastNorthCentral. C:/
  gispy/scratch/Central.pdf created.
```

   The output '.pdf' document, C:/gispy/scratch/Central.pdf, looks something like Figure 24.5:

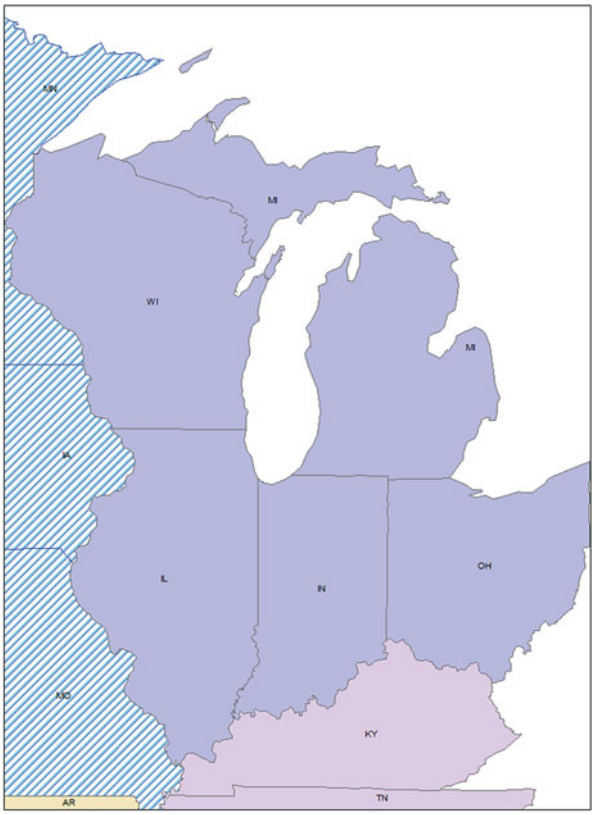


Figure 24.5 PDF exported by activeZoom.py for the sample input.

6. **modSymbol.py** Write a script that modifies the symbology of the first layer in the first data frame in a map and export the results using the following steps:

   Step 1. Use a training layer to modify the symbology to graduated symbols.
   Step 2. Change the field value of the graduated symbols to 'FEMALES', the number of classes to 3, and the normalization field to 'POP2012'.
   Step 3. Export a JPEG of the resulting map.

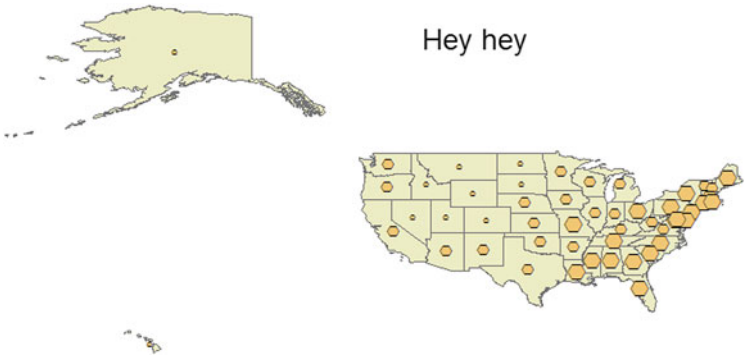
   Do not save the changes to the map document within the script. The script should take three arguments, a full path filename of the '.mxd' file, a full path training layer name, and a full path JPEG name.

   Example input: C:/gispy/data/ch24/maps/symbologyExample.mxd C:/gispy/data/ch24/symbolTraining/gradSymbolsNE.lyr C:/gispy/scratch/Females2010.jpg

   Example output:

```
>>> C:/gispy/scratch/Females2010.jpg created.
```

The output image should look similar to this one:



7. **modLayout.py** Write a script that modifies the north arrow in a map. Specifically, it will move the north arrow to the bottom left corner of the data frame and triple the size of the arrow. Export a TIFF of the resulting map to the 'C:/gispy/scratch' directory. Do not save the changes to the map document within the script. The script should take two arguments, a full path map name and a TIFF base name, in that order. Assume the input map has an existing north arrow layout element named 'North Arrow'.

   Example input:
   C:/gispy/data/ch24 /maps/layoutElemsExample.mxd northArrow.tif

   Example output:

```
>>> C:/gispy/scratch/northArrow.tif created.
```

The output image should look similar to this one: