McIDAS-V Tutorial

An Introduction to Jython Scripting

updated October 2015 (software version 1.5)

McIDAS-V is a free, open source, visualization and data analysis software package that is the next generation in SSEC's 40-year history of sophisticated McIDAS software packages. McIDAS-V displays weather satellite (including hyperspectral) and other geophysical data in 2- and 3-dimensions. McIDAS-V can also analyze and manipulate the data with its powerful mathematical functions. McIDAS-V is built on SSEC's VisAD and Unidata's IDV libraries. The functionality of SSEC's HYDRA software package is also being integrated into McIDAS-V for viewing and analyzing hyperspectral satellite data.

McIDAS-V version 1.2 included the first release of fully supported scripting tools. Running scripts with McIDAS-V allows the user to automatically process data and generate displays for web pages and other environments. The McIDAS-V scripting API is written in java implementation of python called Jython. The McIDAS-V scripting library system library is still under development and new tools will be added with future releases of McIDAS-V. You will be notified at the start-up of McIDAS-V when new versions are available on the McIDAS-V webpage - http://www.ssec.wisc.edu/mcidas/software/v/.

If you encounter any errors or would like to request an enhancement, please post questions to the McIDAS-V Support Forums - http://www.ssec.wisc.edu/mcidas/forums/. The forums also provide the opportunity to share information with other users.

This tutorial assumes that you have McIDAS-V installed on your machine, and that you know how to start McIDAS-V. If you cannot start McIDAS-V on your machine, you should follow the instructions in the document entitled *McIDAS-V Tutorial – Installation and Introduction*. More training materials are available on the McIDAS-V webpage and in the "Getting Started" chapter of the ***McIDAS-V User's Guide***, which is available from the Help menu within McIDAS-V.

Terminology

There are two windows displayed when McIDAS-V first starts, the **McIDAS-V Main Display** (hereafter **Main Display**) and the **McIDAS-V Data Explorer** (hereafter **Data Explorer**).

The **Data Explorer** contains three tabs that appear in bold italics throughout this document: ***Data Sources, Field Selector***, and ***Layer Controls***. Data is selected in the ***Data Sources*** tab, loaded into the ***Field Selector***, displayed in the **Main Display**, and output is formatted in the ***Layer Controls***.

Menu trees will be listed as a series (e.g., ***Edit -> Remove -> All Layers and Data Sources***). Mouse clicks will be listed as combinations (e.g., *Shift+Left Click+Drag*).

Python vs. Jython

Fact: You will do all of your McIDAS-V programming in the Python programming language.

Fact: McIDAS-V uses an implementation of the Python programming language called Jython.

Fact: The original and most widely used implementation of the Python programming language is not Jython; the full and proper name for that one is actually CPython. (In case you're wondering, Jython is implemented in Java, and CPython is implemented in C!). In day-to-day usage, CPython is often referred to as just "Python."

What does all that mean?

As you learn McIDAS-V scripting, you should know that all the documentation you'll find across the web for the Python *programming language* (specifically, version 2.7) is relevant and accurate. Jython is very careful to retain compatibility with the Python language. This includes almost the entire Python standard library, which is why we had access to the functions we needed to import, like **glob** and **basename**.

However, the most important distinction between Jython and CPython is the availability of libraries. Because Jython is Java-based, we do **not** generally get to use the Python libraries that depend on "native" code. As a result, there is a large list of libraries we **cannot** use in Jython, including:

- SciPy
- NumPy
- matplotlib
- netcdf4-python
- h5py

So, once you are comfortable with the basics of the Python programming language, the remainder of learning McIDAS-V scripting is largely about learning the "McIDAS-V way of doing things": instead of NumPy, matplotlib, and netcdf4-python, we will use the VisAD data model, VisAD displays, and NetCDF-Java to get our work done.

In summary, Jython's language and standard library are the same as CPython, but many non-standard Python libraries such as NumPy are not available in Jython. However, McIDAS-V has good alternatives for data access, plotting, and numerical computation.

Using the Jython Shell

The **Jython Shell** consists of an *output window* on top and an *input field* on the bottom. The user enters Jython into the *input field*. When the Enter key or "**Evaluate**" is pressed, the Jython input is evaluated and output is shown in the *output window*. The **Jython Shell** is a great tool to begin writing scripts that can be run from the background. When inputting commands, the **Jython Shell** runs in single or multi-line mode. You can switch modes by using the double down arrows or with the shortcut **Ctrl+/**. The **Evaluate** button also has a shortcut **Shift+Enter**.

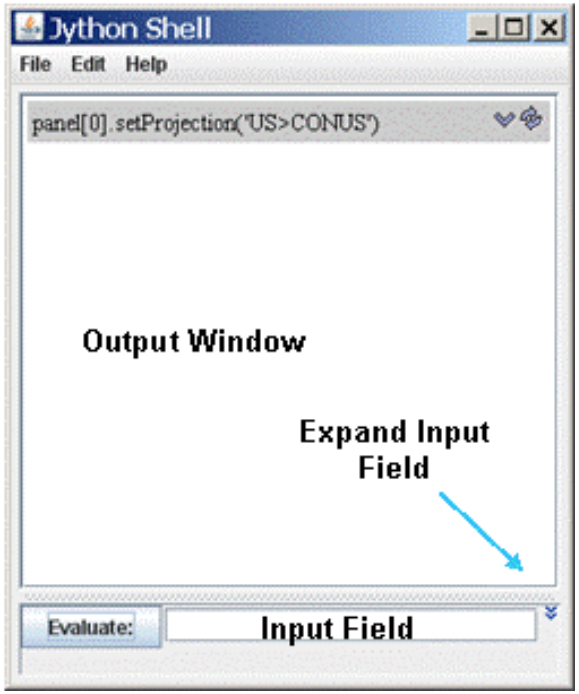
Here is a chart containing keyboard shortcuts for the **Jython Shell**:

Enter	Evaluate command in single-line input mode
Shift+Enter	Evaluate command in multi-line input mode
Ctrl+/	Switch between single and multi-line input modes
Ctrl+p	Recall a previously evaluated command
Ctrl+n	Recall the next run command, assuming Ctrl+p was already used

1. Using the **Jython Shell**, create a window with a single panel **Map Display**.

   a. In **Main Display**, select ***Tools -> Formulas -> Jython Shell*** to open the **Jython Shell**.

   b. In the *input field*, type:
   panel = buildWindow( )
   Click **Evaluate**.



buildWindow is the function used to create an object that contains an array of panels. This creates a window as you would using the GUI with ***File -> New Display Window…***.

2. Now create another window, this time with a **Globe Display**. Using the same **Jython Shell**, in the *input field*, type:
globePanel = buildWindow(height=600, width=600, panelTypes=GLOBE)
Click **Evaluate**.

You now have two single paneled displays, each of which can be modified.

3. Turn off the wireframe box on the Map Display and then rotate the Globe Display.

In the *input field*, type:
panel[0].setWireframe(False)
Click **Evaluate**.
In the *input field*, type:
globePanel[0].setAutoRotate(True)
Click **Evaluate**.

setWireframe and **setAutoRotate** are methods which operate on an object. In these examples, the objects are **panel** and **globePanel**.

Basic Jython Terminology

In the above examples we introduced the terms *function*, *method* and *object*. In most general terms, an object is returned from a function and a method operates on an object and may return a new object.

In steps 1 and 2, the **buildWindow** function is used to create an object, in this case an array of panels. Objects can have one or more attributes and these attributes are defined by a class. In later examples of this tutorial, you will see the importance of knowing these attributes. Methods are used to operate on an object. In step 3, **setWireframe** operates on the panel object by turning off the wireframe box.

The word *object* can be intimidating because it hints at the topic of *object-oriented programming*, which can be complex and confusing. However, as McIDAS-V programmers, we just need to know that an object is a special kind of variable that has *functions* associated with it. These special kinds of *functions* are referred to as *methods*.

4. It is important to understand how to interact with the kinds of objects encountered frequently. For example, the **list** (described later) is an object. For this particular kind of object, methods like **append** and **remove** can be used. Click the **Expand Input Field** icon to the right of the *input field* so multiple lines can be entered into the **Jython Shell**.

   In the *input field*, type:
   myList = [1, 2, 3]
   print(myList)
   Click **Evaluate**.
   This code results in:
   [1, 2, 3]

   In the *input field*, type:
   myList.append(4)
   print(myList)
   Click **Evaluate**.

This code results in:
[1, 2, 3, 4]

In the *input field*, type:
myList.remove(2)
print(myList)
Click **Evaluate**.
This code results in:
[1, 3, 4]

5. Defining new types of objects is possible but it is outside the scope of this tutorial. As a McIDAS-V programmer, most of the objects needed are already defined. For example, in McIDAS-V, there is a **Window** object, and its size can be adjusted with the method **setSize**.

In the *input field*, type:
panel2 = buildWindow()
panel2[0].setSize(800,800)
Click **Evaluate**.
This code results in a window being built with a size of 800x800.
This window can now be closed.

It is important to know the input parameters for each of the functions and methods. McIDAS-V Jython functions and methods are documented in the scripting section of the ***McIDAS-V User's Guide***:
http://www.ssec.wisc.edu/mcidas/doc/mcv_guide/current/misc/Scripting/Scripting.html

Any documentation for the core Python (2.7) language and standard library will be valid for Jython and McIDAS-V. Python has become a favorite learning language, so there is a lot of information available. The syntax is case sensitive and adheres to strict indentation practices. Here are a few good sources of information:

- Learn Python The Hard Way (http://learnpythonthehardway.org/book/index.html)
- Standard Python documentation (https://docs.python.org/2/), especially the tutorial (https://docs.python.org/2/tutorial/index.html)
- Python Style Guide (https://www.python.org/dev/peps/pep-0008/)

Using the Jython Shell (continued)

6. The Map Display will be used in the remaining examples, so at this time, close the Globe Display.

7. Change the projection and center point of the display. The projection name will follow the structure of the menu tree of the Main Display.

a. In the *input field*, type: **panel[0].setProjection('US>States>Midwest>Wisconsin')**
Click **Evaluate**.

b. In the *input field*, type: **panel[0].setCenter(43.0,-89.0)**
Click **Evaluate**.

setProjection changes the projection of a panel. The syntax for setting a projection is similar to the menu structure you see when you change the projection using the GUI. Note, Jython is a case sensitive language, and you must type things exactly as documented here.

8. Add some annotations to the display.

a. Determine the available fonts for your OS. In the *input field*, type (the 4 spaces before **print** are necessary):
for fontname in allFontNames( ):
** print fontname**
Click **Evaluate.**

b. From the results, pick a font for the next commands. In these examples, SansSerif is used. In the *input field*, type:
here = panel[0].annotate('You Are Here', size=20, font='SansSerif', lat=43.5, lon=-89.2, color='Red')
Click **Evaluate**.

The bottom left corner of the text is located at the specified latitude/longitude coordinates. Line and element coordinates are also available in **annotate**. Color can be specified using RGB values or the color name. html tags can also be used to do things like making the font bold.

c. In the *input field*, type:
plus = panel[0].annotate('+', size=20, font='SansSerif', line=200, element=295, color=[1.0,0.0,1.0])
Click **Evaluate**.

d. When you are through adding annotations to the display, close the window created with **buildWindow**.

Indentation in Python

In the previous step, it was required that the **print** line be indented 4 spaces. Python syntax is focused on code readability. The Python programming language requires specific, consistent indentation of source code and uses block indentation to control the flow. This tutorial, as well as any documentation of McIDAS-V scripting, will use indentation of 4 spaces (https://www.python.org/dev/peps/pep-0008/#tabs-or-spaces). Not all programming languages require this specific indentation.

For example, consider the following example of valid IDL code:

myList = [1, 2, 3]	This code results in:
for i=0, n_elements(myList)-1 do begin	*1*
print, myList[i]	*2*
endfor	*3*

IDL control blocks (if, for, foreach, while, etc.) do not need indentation to work properly. The IDL code above is not formatted well, but it still runs as expected.

9. Run commands in the Jython Shell to become accustomed to the indentation required for scripts to run.

   a. In the *input field*, type:
   myList = [1, 2, 3]
   for thing in myList:
   print(thing)
   Click **Evaluate**.

   The Jython Shell raises an error:
   SyntaxError: ("mismatched input 'print' expecting INDENT", ('<string>', 3, 0, 'print(thing)\n'))
   The only thing wrong with the Python code in step 9a is the missing 4-space indentation in front of **print**.

   b. Add a 4-space indentation to the **print** statement. In the *input field*, type:

myList = [1, 2, 3]	This code results in:
for thing in myList:	*1*
** print(thing)**	*2*
Click **Evaluate**.	*3*

   c. The indentation of **print thing** is required by Python. However, there is no need for an **ENDFOR**. Every indented line is considered to be "inside" the **for** loop. In the *input field*, type:

```
myList = [1, 2, 3]
endingMessage = "Loop is finished"

for thing in myList:
  print(thing)

print endingMessage
```

   Click **Evaluate**.

This code results in:
1
2
3
Loop is finished

d. "This is junk" from the previous example was only printed a single time after the **for** loop. This is because **print junk** is one indentation level to the left of those inside the **for** loop, indicating the end of the **for** block. The same is true for **if/else**. In the *input field*, type:

```
mcv_is_cool = True
if mcv_is_cool:
   print 'McIDAS-V is great!'
else:
   print 'McIDAS-V is horrible!'
```

Click **Evaluate**.

This code results in:
McIDAS-V is great!

Again, notice the lack of **ENDIF** or **ENDELSE**. Or, comparing to C-style languages, note the absence of anything like a closing curly brace. In Python, the end of a control block is indicated by a "dedent" (i.e. the next line starts one indentation level to the left).

e. In review, the control flow in Python is indicated with indentation, as in the following code. In the *input field*, type:

```
condition = True
if condition:
   print 'beginning of if block'
else:
   print 'beginning of else block'
print 'end of if/else block'
```

Click **Evaluate**.

This code results in:
beginning of if block
beginning of if/else block

Note the colons at the end of **if** and **else**, lack of parentheses around **condition**, indentation to indicate the start of the **if** block, and the "dedent" to indicate the end of the entire **if/else** block. If/else was used in this example, but the same holds true for other control statements like **while** and **for**.

Lists and For in Loops in Python

Python has a **list** data type similar to arrays in scientific programming languages like IDL or MATLAB. However, there is a difference.

10. Here is how Python's **list** works.

    a. In the *input field*, type: **zero_to_ten = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]**
    Click **Evaluate**.

    b. Indexing is zero-based. In the *input field*, type: **print zero_to_ten[5]**

    This code results in:
    5

    c. "Slicing" syntax can be used, but be careful! The second part of the slice is the index *after* the last element selected by the slice. In the *input field*, type: **print zero_to_ten[3:6]**

    This code results in:
    [3, 4, 5]
    The 6 is not included.

    d. Python lists are not restricted to single data types as arrays are in other languages. Python lists can contain mixed data types. In the *input field*, type: **myList = [0, 1, 'foo', 2]**

    This property can be useful but makes the **list** data type a poor choice for representing large sets of numbers often needed in scientific computing.

    e. A common way to loop through lists in Python is with the **for..in** syntax. In the *input field*, type:

```
myList = [0, 1, 'foo', 2]
for thing in myList:
    print thing
```

This code results in:
0
1
foo
2

f. If indices are needed, like in a classical **for** loop, you can "enumerate" the list. In the *input field*, type:
for (i, thing) in enumerate(myList):
print 'The list at index %d is: %s' % (i, thing)

This code results in:
The list at index 0 is: 0
The list at index 1 is: 1
The list at index 2 is: foo
The list at index 3 is: 2

Lists are powerful and ubiquitous in Python, so get to know them. An example of a for loop will be used in the next section to list directory information from images in a dataset.

Creating a Simple Local ADDE Request

Up until now all of the functions have been customizing panel attributes. McIDAS-V scripting can also make ADDE requests to list and transfer image data. Once data has been transferred, it can be used to create data layers. The next part of this tutorial will access data from the Joplin tornado from 2011.

11. Create local datasets to access the Joplin tornado infrared imagery. In the *input field* of the **Jython Shell**, type:
dataDir = '<*local-path*>/Data/Scripting/tornado-areas/IR'
irDataSet = makeLocalADDEEntry(dataset='TORNADO', imageType='GOES-13 IR', mask=dataDir, format='McIDAS Area', save=True)

12. **listADDEImages** is a function that creates a list of dictionaries containing information about each available image. Dictionaries will be described in more detail later in this tutorial. Request a listing of all images from the dataset TORNADO. In the *input field* of the **Jython Shell**, type: **dirList = listADDEImages(server='localhost', position='ALL', localEntry=irDataSet)**

13. Earlier, we listed all the available fonts found on your machine. Using the same techniques, list out the directory information for each image. In the *input field* of the **Jython Shell**, type:

```
for imageDir in dirList:
   print '  '
   print 'New image directory %s %s' % (imageDir['sensor-type'], imageDir['nominal-time'])
   print ' ----------------------------------------------------------------------------------------------------'
   for key,value in imageDir.iteritems():
      print key,value
```

14. **loadADDEImage** is the function used to request imagery from an ADDE server. The inputs to **loadADDEImage** are in the form of keyword, value pairs. The dictionaries returned from **listADDEImages** are in this same format and can be used as inputs to **loadADDEImage**. Make an ADDE request to get the imagery data from the first keyword parameter pairing returned from **listADDEImages**. In the *input field*, type:
    imageData = loadADDEImage(size='ALL', **dirList[1])

15. **loadADDEImage** returns one object containing a list of metadata and an array of data. Build a new window using **buildWindow** and display the data using **createLayer**. In the *input field*, type:
    panel = buildWindow(height=600, width=900, panelTypes=MAP)
    dataLayer = panel[0].createLayer('Image Display', imageData)

16. Use the method **captureImage** to save the display to a file in the **imagePath** directory. In the *input field*, type:
    panel[0].captureImage('<*local-path*>**/Data/Scripting/Images/IR-Image.jpg')**

    Because McIDAS-V does a screen capture on some platforms, be sure that the entire window is showing and is not blocked by other windows, or your resulting image will not be complete. After viewing **IR-Image.jpg** in a browser, close the image window.

Creating a Simple Remote ADDE Request

The data from the Joplin tornado from 2011 can also be found on the remote server pappy.ssec.wisc.edu. If you do not have internet access to remote servers, continue with next section.

17. Request a listing of all images from the dataset TORNADO found on the server pappy.ssec.wisc.edu. In the *input field*, type:
    dirList = listADDEImages(server='pappy.ssec.wisc.edu', dataset='TORNADO', descriptor='GOES13-IR', position='ALL')

18. As with the local dataset, list the directory information for each image. In the *input field*, type (the 4 space indentations are necessary):

```
for imageDir in dirList:
   print '  '
   print 'New image directory %s %s' % (imageDir['sensor-type'], imageDir['nominal-time'])
```

```
print '-'*55
for key,value in imageDir.iteritems():
   print key,value
```

The directories returned from a remote **listADDEImages** request are identical to those of a local ADDE request and can be used as inputs to **loadADDEImage**.

Dictionaries in Python

One useful data type in Python is called the **dict**, which is short for **dictionary**. A **dictionary** is a set of associations between "keys" and "values". Comparing this to a real life dictionary (the book of words and meanings), the "key" is the word being looked up, and the "value" is the definition of the word.

19. Run commands in the Jython Shell to become accustomed to dictionaries. The dictionaries created here are only to illustrate Python syntax and not directly useful as inputs to McIDAS-V functions. In Python, the "keys" of a dictionary can be almost anything, as long as the value of that thing doesn't change over the lifetime of the program. Numbers and strings are the most common "keys"; the value of 3 of 'a' doesn't ever change. "Values", in contrast, can be just about anything: numbers, strings, lists, and even other dictionaries

    a. Define a dictionary. In the *input field*, type:

    ```
    resolution = {
       # Key:    Value,
       'Band1': '1km',
       'Band2': '4km',
       'Band3': '4km',
    }
    ```

    Click **Evaluate**.

    b. Print the list of keys included in the dictionary. In the *input field*, type:

    ```
    print resolution.keys( )
    ```

    c. Once the dictionary is defined, key/value pairs can be accessed with square brackets. In the *input field*, type:

    ```
    print resolution['Band1']
    print resolution['Band2']
    ```

    This code results in:
    1km
    4km

d. If 'Band1', 'Band2', etc. is too verbose, integer keys can be used instead. In the *input field*, type:

```
resolution = {
   1: '1km',
   2: '4km',
   3: '4km',
}
print resolution[1]
print resolution[2]
```

The code results are the same.

e. Remember, the dictionary values can be arbitrarily complex. This makes it possible to represent a lot of useful information in an accessible way. In the *input field*, type:

```
sensor_info = {
   'name': 'viirs',
   'bands':      ['SVI01', 'SVM02', 'SVM03'],
   'resolution': ['375m',  '750m',   '750m'],
}
```

In this example, the **'name'** key maps to a simple string, but the **'bands'** and **'resolution'** keys map to lists of band information.

Dictionaries are extremely flexible and are often used in McIDAS-V. The next section of this tutorial covers building a dictionary to represent all parameters for a single ADDE request. That dictionary can then be passed to a McIDAS-V function that will return the data.

Using Dictionaries and Metadata to Formulate an ADDE Request

Most ADDE requests need many more parameters than the previous example. Specifying long lists of keyword parameters can be cumbersome and create code that is difficult to read. To avoid these problems, you can take advantage of a Python dictionary. Using a Python dictionary, you can specify all of the key:value pairs, or include just a few, and add the extra ones directly to the **loadADDEImage** function call.

20. The next few steps require a lot of typing. If you'd like, you can cut and paste the lines from the <*local path*>**/Data/Scripting/ADDE-dictionary.txt** file into the **Jython Shell** and then skip to the next step. All of the files used in this tutorial are printed at the end of the document. Alternatively, use the **editFile** function via **editFile(**<*local-path*>**/Data/Scripting/ADDE-dictionary.txt')**

a. Earlier in the tutorial, you created a local ADDE dataset for GOES-13 IR dataset for the TORNADO case. Use getLocalADDEEntry to get the value for localEntry and use it to create a dictionary to be use local data with loadADDEImage. In the *input field*, type:
irLocalDataSet = getLocalADDEEntry(dataset='TORNADO', imageType='GOES-13 IR')

b. In the *input field*, type (the 4 space indentation is required):
```
ADDE_IR_loadRequest = dict(
    server = 'localhost',
    localEntry = irLocalDataSet,
    size = 'ALL',
    time = ('23:45:00', '23:45:00'),
    day = '2011142',
    unit = 'BRIT',
)
```

c. Make an ADDE request for infrared data using key:value pairs and a dictionary. The ** before the dictionary tells Python to evaluate the dictionary's contents and include the key:value pairs in **loadADDEImage**. The dictionary must be last in the list. In the *input field*, type:
irData = loadADDEImage(band=4, **ADDE_IR_loadRequest)

21. **loadADDEImage** returns one object containing a list of metadata and an array of data. Build a new window using **buildWindow** and display the data using **createLayer**. The above request was for all the lines and elements (**size='ALL'**). Creating a window to show the entire image would probably go beyond the extents of your desktop. To avoid this problem, use the metadata to create a window with dimensions of half the number of lines and elements. In the *input field*, type:
```
bwLines = irData['lines'] / 2
bwEles = irData['elements'] / 2
panel = buildWindow(height=bwLines, width=bwEles)
```

22. Now create layer objects for the infrared data. Use **createLayer** with the object **irData**. In the *input field*, type:
irLayer = panel[0].createLayer('Image Display', irData)

23. Apply the '**Longwave Infrared Deep Convection**' color table to the infrared layer. Since there is a unique name for each color table, the syntax is a little different than that used with **setProjection**, and the entire naming structure is not necessary here. In the *input field*, type:
irLayer.setEnhancement('Longwave Infrared Deep Convection')

24. Using the values from the keywords '**sensor-type**' and '**nominal-time**' from the **irData** object, create a string to use with **setLayerLabel** (remember that the 4 spaces of indentation are mandatory).

a. Print the list of keys included in the **irData** object. In the *input field*, type:
print irData.keys( )

b. In the *input field*, type:
```
irLabel = '%s %s' % (
   irData['sensor-type'],
   irData['nominal-time']
)
```

c. In the *input field*, type:
irLayer.setLayerLabel(label=irLabel, size=16, color='White', font='SansSerif')

c. After checking the new layer label in the **buildWindow Display**, close the window.

Functions in Python

Functions are a way to refer to a piece of code that takes arguments and returns results based on those arguments. Functions are the key to creating reusable code and avoiding repetition, and Python makes them easy to define and use. The next sections utilize functions in McIDAS-V. In Python, defining functions is straightforward.

25. Create a function named **add** and demonstrate its usage with numbers and letters. In the *input field*, type:
```
def add(a, b):
   return a+b
print add(2,2)
```
Click **Evaluate**.

This code results in:
4

As with loops and **if/else** blocks, indentation/dedent indicate the end of the function.

26. Similarly to IDL but unlike languages like C and Fortran, Python functions do not need to explicitly state the type of argument. Consequences of this can be observed when attempting to use something other than numbers with the **add** function. In the *input field*, type:

print add('a', 'b')
Click **Evaluate**.

This code results in:
ab

This still works. The + operator works just as well on 'a' and 'b' as it does on 1 and 2, so the function completes without error.

Jython Library

The above exercises used functions such as **setLayerLabel** and **loadADDEImage**. The code for these functions can be found in the **Jython Library**.

27. Open the **Jython Library** and search for code to set a layer label.

    a. In **Main Display**, select ***Tools -> Formulas -> Jython Library***.

    b. Open the ***System -> Background Processing Functions*** library.

    c. Using the search utility type **setLayerLabel**. This shows you the code that sets a layer label. Keep pressing Enter or use the up/down arrows to search for multiple instances of **setLayerLabel** in the library.

28. Users can also share code by adding functions to the **Local Jython Library**. Add a function named **importColorTable** to the ***Local Jython*** library. This function will be used in the next part of the tutorial.

    a. Create a new Jython library by selecting ***File->New Jython Library*** from the menu. Type **Training Library** and click **OK**.

    b. Open a text editor (e.g. gedit, vi, WordPad, Notepad++), and edit the file <*local-path*>**/Data/Scripting/importColorTable.txt**.

    c. Copy the entire contents of this file and paste it into the ***Local Jython -> Training Library*** library.

    d. Select **Save** and close the **Jython Library**.

Using Functions in a McIDAS-V Script

Building upon the previous examples, the next script uses the **importColorTable**, **mask** and **mul** functions. **mask** and **mull** are system functions, packaged with McIDAS-V.

The <*local-path*>**/Data/Scripting/function.py** file is an example script showing how to use these functions and ways to make the script platform independent. These are not be entered into the **Jython Shell** at this time. Read through the following portions of the script and the associated comments to learn what the script is doing at each step.

The first line of the code imports common functions in the os library. These functions are used to create platform-independent path names.

```
import os
#
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users
#

homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')

areaPath = os.path.join(scriptingPath, 'tornado-areas')
irPath = os.path.join(areaPath, 'IR')

#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')
```

The GOES-13 IR TORNADO dataset is used, this time with temperature **(unit='TEMP')** values requested:

```
#
#   This example gets the information from the dataset created previously in the tutorial
#
irLocalDataSet = getLocalADDEEntry('TORNADO','GOES-13 IR')
ADDE_IR_loadRequest = dict(
    debug=True,
    server='localhost',
```

```
    localEntry=irLocalDataSet,
    size='ALL',
    time=('23:45:00','23:45:00'),
    day='2011142',
    unit='TEMP',
)

irData = loadADDEImage(**ADDE_IR_loadRequest)
```

The **mask** function requires a temperature threshold:

```
#
#   assign a temperature threshold used with mask() function
#
temperatureThreshold = 250.0
```

Applying a **mask** requires a two part process. First, a threshold temperature is applied to the data object returned from **loadADDEImage**, creating a new data object of either values of 1's or missing. Second, the original data object is multiplied by the new data object. The data object created by the **mul** function creates a data object that contains either temperature or missing values.

```
#
#   Applying a mask is a two part process.
#   First we assign a value of 1 or missing value to a temporary data object
#   Second, multiply the first results to the temporary data object
#
maskedData = mask(irData, 'lt', temperatureThreshold, 1)
finalDataSet = mul(irData, maskedData)
```

The **importColorTable** function reads in a file exported using the color table editor. The name of the color table is extracted and used with **setEnhancement**:

```
#
#   Import enhancement table
#
IRColorTableFile = os.path.join(enhancementPath, 'Tornado-IR.xml')
IRTable = importColorTable(IRColorTableFile)
IRTableName = IRTable.getName()
```

As previously done, the last section of the script builds a window, creates the layer, applies the enhancement table and sets the projection.

```
#
```

```
#   Build a window
#
bwLines = irData['lines'] / 2
bwEles = irData['elements'] / 2
panel = buildWindow(height=bwLines, width=bwEles)
panel[0].setWireframe(False)
#
#   Add layers to the existing window set enhancement table and data ranges
#
irLayer = panel[0].createLayer('Image Display', finalDataSet)
irLayer.setLayerLabel('GOES-13 Temperatures less than ' + str(temperatureThreshold) + ' %timestamp%')
irLayer.setEnhancement(IRTableName, range=(temperatureThreshold,200))

#
#   Set the center latitude, longitude and scale
#
panel[0].setProjection('US>States>N-Z>Oklahoma')
panel[0].setCenter(33, -97, scale=.5)

fileName=os.path.join(imagePath, 'ir-image.gif')
panel[0].captureImage(fileName)
```

29. From the **Jython Shell**, run **function.py**. In the *input field*, type:
    editFile('<*local-path*>**/Data/Scripting/function.py**')

30. Evaluate the **function.py** file by clicking **Evaluate**.

31. Open a browser and view the file '<*local-path*>**/Data/Scripting/Images/ir-image.gif**'.

Creating Movies in a McIDAS-V Script

In the previous example, you created a single image. You can also create movies that contain loops of images. To do this, multiple data requests must be made. The <*local-path*>**/Data/Scripting/image-movie.py** file is an example script showing the creation of movie loops in McIDAS-V.

In this example, the loop is created by making a call to **listADDEImageTimes** and multiple calls to **loadADDEImage**. **listADDEImageTimes** is similar to **listADDEImages**, but returns a list of dictionaries containing only image days and times. Below is part of the script with some comments. These are not be entered into the **Jython Shell** at this time.

A Python list, as described previously, is used to store data objects and is initialized using the syntax below. As the script loops through **loadADDEImage** calls, the data objects returned are appended to the list. In this script, **myLoop** is the python list:

```
myLoop=[]
```

listADDEImageTimes uses the dictionary **parms** as its input parameters. The dictionary object **dateTimeList** is returned and contains keyword/value pair for each day and time.

```
dateTimeList = listADDEImageTimes(**parms)
```

The script then loops through all the dictionaries returned from the call to **listADDEImageTimes**. Using a for loop, individual directories, **dateTime**, are extracted from the list dictionaries, **dateTimeList**, which was returned from **listADDEImageTimes**. The loop takes the **time** value out of the **dateTime** dictionary which is used to create a new dictionary that is passed into **loadADDEImage**.

```
for dateTime in dateTimeList:

    imageTime = dateTime['time']

    ADDE_IR_loadRequest = dict(
        localEntry=irLocalDataSet,
        day=dateTime['day'],
        time=(imageTime,imageTime),
        band=4,
        unit='TEMP',
        size='ALL'
    )

    IRData = loadADDEImage(**ADDE_IR_loadRequest)
    maskedData = mask(irData, 'lt', temperatureThreshold, 1)
    finalDataSet = mul(irData, maskedData)
```

The data objects returned from **listADDEImageTimes** and passed through **mask** and **mul** are added to **myLoop** using the **append** method.

```
    myLoop.append(finalDataSet)
```

Once the loop is completed, a window is built and **myLoop** is used to create an **Image Sequence Layer** which is then saved as an animated gif.

```
bwLines = irData['lines'] / 2
bwEles = irData['elements'] / 2
panel = buildWindow(height=bwLines, width=bwEles)
panel[0].setWireframe(False)
```

```
irLayer = panel[0].createLayer('Image Sequence Display' ,myLoop)

writeMovie(imagePath+'ir-loop.gif')
```

32. From the **Jython Shell** run **image-movie.py**. In the *input field*, type:
editFile('<*local path*>**/Data/Scripting/image-movie.py**')

33. Evaluate the **image-movie.py** file by clicking **Evaluate**.

34. Open a browser and view the file '<*local-path*>**/Data/Scripting/Images/ir-loop.gif**'.

Creating Your Own McIDAS-V Script

35. You now have all the tools necessary to write a script that creates a movie of the infrared images placed over a basemap. For this exercise,
    a. import the enhancement table from <*local*-path>/**Data/Scripting/Color-Enhancements/Tornado-basemap.xml**
    b. write a script that does the tasks listed below:
        1. uses the local data files from the **TORNADO** '**GOES-13 IR**' dataset
        2. creates and uses the local McIDAS Area dataset for a base map
            a. name the dataset **TORNADO**
            b. assign the image type '**Land Sea Mask**'
            c. data is located <*local*-path>/**Data/Scripting/tornado-areas/BASE**
        3. uses the entire size of the image of both datasets
        4. loads a list IR temperature data that spans 14:45 on day 2011142 to and 02:45 on day 2011143
            a. the **mask** function removes temperature greater than 250 K
            b. uses the enhancement table <*local*-path>/**Data/Scripting/Color-Enhancements/Tornado-IR.xml**
            c. applies the color enhancement with a range of 250 to 200 K
        5. loads a single base map image and uses the enhancement table:
            <*local*-path>/**Data/Scripting/Color-Enhancements/Tornado-basemap.xml**
        6. builds a window 700 lines X 1000 elements
        7. creates an Image Display layer from the base map dataset (do not include the layer label)
        8. overlays an Image Sequence Display layer from the IR data list
        9. adds a layer label to the IR data set which includes
            a. the text 'Joplin Tornado'
            b. timestamp
            c. displayname
        10. sets the projection to Central U.S.
        11. changes the center point to 35N 97W with a scale factor of 1.5

12. turns off the wireframe box
13. adds the annotation 'Joplin, Missouri'; text is left and center justified at 37.15N and 94.5W
14. saves the movie with the file name of <loc*al-path*>**/Data/Scripting/Images/image-exercise.gif**

An example solution is available at <*local path*>**/Data/Scripting/image-exercise.py**. However, before checking the solution, it is recommended that you try to complete the tasks on your own.

Running Scripts from a Command Prompt

So far in this tutorial, you have been running commands and scripts using the **Jython Shell**. Scripts can also be run from the command line by adding the flag **-script** to the startup script.

36. Run the McIDAS-V script using the **–script** flag.

    a. Exit McIDAS-V.

    b. Open a terminal and change directory to the directory where McIDAS-V is installed (<*user-path*>**/McIDAS-V-System**)

    c. Run the <*local-path*>**/Data/Scripting/image-exercise.py** script.

       For Unix, type:
       ./runMcV –script <*local-path*>**/Data/Scripting/image-exercise.py**

       For Windows, type:
       runMcV.bat –script <*local-path*>**/Data/Scripting/image-exercise.py**

    d. The progress of the script can be monitored by watching the **mcidasv.log** file in your McIDAS-V directory with the **tail** command.

       Type: **tail -f** <*user-path*>**/McIDAS-V/mcidasv.log**

    e. From your browser, view the file <*local-path*>**/Data/Scripting/Images/image-exercise.gif** that was created from <*local-path*>**/Data/Scripting/image-exercise.py**.

Calculating Statistics in a McIDAS-V Script

Calculating statistics for data is also important. McIDAS-V uses the VisAD statistics package to calculate statistics. *<local-path>*/**Data/Scripting/stats.py** is an example script showing statistics calculations in McIDAS-V scripting. Below is part of the script with some comments. These are not be entered into the **Jython Shell** at this time.

To calculate statistics on your data, extra lines of code are needed to specify the output files and the data object passed into the package. These lines open the files for writing statistics:

```
textFileName = os.path.join(statisticsPath, "stats.txt")
textOutputFile = open(textFileName, "w")

csvFileName = os.path.join(statisticsPath, "stats.csv")
csvOutputFile = open(csvFileName, "wb")
```

This line defines how to delimit the data going to the csv file:

```
csvData = csv.writer(csvOutputFile, delimiter=",")
```

This line writes a header text file:

```
csvData.writeRow(["Time", "latitude", "longitude", "geometricMean", "min",
                  "median", "max", "kurtosis", "skewness", "stdDev", "variance"])
```

This line passes the data from loadADDEImage to the statistics package:

```
stats = Statistics(finalDataSet)
```

These line write the statistic to the output text file:

```
textOutputFile.write("   stat and value for: %s \n" % irData["nominal-time"])
textOutputFile.write("   geometric mean: %s \n" % stats.geometricMean())
textOutputFile.write("   kurtosis: %s \n" % stats.kurtosis())
textOutputFile.write("   num points: %s \n" % stats.numPoints())
textOutputFile.write("   skewness: %s \n" % stats.skewness())
textOutputFile.write("   std dev: %s \n" % stats.standardDeviation())
textOutputFile.write("   variance: %s \n" % stats.variance())
textOutputFile.write("\n")
```

This line writes the statistics to the csv file:

```
csvData.writerow([theTime, "37.0", "-94.5", stats.geometricMean(), stats.min(),
                  stats.median(), stats.max(), stats.kurtosis(), stats.numPoints(),
                  stats.skewness(), stats.standardDeviation(), stats.variance()])
```

37. From a terminal in the directory where McIDAS-V is installed, run the **stats.py** script using the –script flag.

    For Unix, type: **./runMcV –script** <*local-path*>**/Data/Scripting/stats.py**
    For Windows, type: **runMcV.bat –script** <*local-path*>**/Data/Scripting/stats.py**

38. You can use the statistics created by McIDAS-V in other software packages, and you can plot the statistics values on your McIDAS-V images.

    a. Using Excel, open the csv file <*local-path*>**/Data/Scripting/statistics/stats.csv**, and do something like create a line graph of your statistics.

    b. Using your text editor, open the text file <*local-path*>**/Data/Scripting/statistics/stats.txt**, and view the file.

    c. From your browser, view the file <*local-path*>**/Data/Scripting/Images/stats-image.jpg** that was created from **stats.py**.

Using glob to Find Files on Disk

Previous sections of this tutorial have concentrated on ADDE requests for imagery data. This section will use a standard Python function called **glob** to find files on disk, a task scientific programmers deal with every day. Restart McIDAS-V before proceeding.

39. List all of the grib2 files contained in the <*local-path*>**/Data/Scripting/tornado-model** directory.

    a. In Python, only a small number of functions are available at start up, and **glob** is not one of them, so the function must be imported. In the *input field*, type: **from glob import glob**

    b. You can read this as "import the function **glob** from the module named **glob**." This example is a little awkward due to the module and function having the same name, so here is another example that imports a function called **basename** that can be used to strip the directory prefix off a full path of a file. In the *input field*, type: **from os.path import basename**

c. You now have all the tools needed to do something useful like locate a set of grib2 RUC files on your system. If you have used IDL's FILE_SEARCH, this will look familiar. The asterisk * is used as a "wildcard" to indicate parts of the filename which are varying. In the *input field*, type:

```
from glob import glob
from os.path import basename
file_path = '<local-path>/Data/Scripting/tornado-model/'
ruc_files = glob(file_path + 'RUC*.grib2')
for full_path in ruc_files:
   print 'Full path to the file: %s' % (full_path)
   print 'Filename only:        %s' % (basename(full_path))
```

glob returns a list of strings (representing file paths) that can be operated on like any **list**. Depending on what is in that directory, the code above may return:

Full path to the file: /home/myuser/Data/Scripting/tornado-model\RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0000_000_NA_XXX_XXXX_XXX.grib2
Filename only: RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0000_000_NA_XXX_XXXX_XXX.grib2
Full path to the file: /home/myuser/Data/Scripting/tornado-model\RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0003_000_NA_XXX_XXXX_XXX.grib2
Filename only: RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0003_000_NA_XXX_XXXX_XXX.grib2
Full path to the file: C:/Users/rcarp/Data/Scripting/tornado-model\RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0006_000_NA_XXX_XXXX_XXX.grib2Filename only: RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0006_000_NA_XXX_XXXX_XXX.grib2
Full path to the file: /home/myuser/Data/Scripting/tornado-model\RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0009_000_NA_XXX_XXXX_XXX.grib2
Filename only: RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0009_000_NA_XXX_XXXX_XXX.grib2
Full path to the file: /home/myuser/Data/Scripting/tornado-model\RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0012_000_NA_XXX_XXXX_XXX.grib2
Filename only: RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0012_000_NA_XXX_XXXX_XXX.grib2

The next section of this tutorial will cover listing information from these RUC grids.

Using Gridded Data in a McIDAS-V Script – Part 1: Listing Grid information

The previous sections of the tutorial concentrated on ADDE requests for imagery data. This part of the tutorial concentrates on functions that load, list and display gridded data. An assumption is made that the previous portions of the tutorial have been completed. Basic concepts discussed above carry over for all McIDAS-V scripts. The files used for this part of the tutorial are found in *<local-path>*/**Data/Scripting/tornado-model**. The files are typical of those that come across the NOAAPORT feed. The *<local-path>*/**Data/Scripting/grid-list.py** file is an example script showing how to begin listing gridded data. The lines of script are shown below interleaved with comments. These comments are not be entered into the **Jython Shell**. Copy and paste the lines of code into the **Jython Shell**.

These lines initialize variables used throughout the script.

```
import os
#
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users
#
#     Some of the path definitions are not not used with in the script,
#     but rather in subsequent scripts
#
homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')

gridPath = os.path.join(scriptingPath, 'tornado-model')
areaPath = os.path.join(scriptingPath, 'tornado-areas')
irPath = os.path.join(areaPath, 'IR')

#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')
```

The next lines of code assign a value to the variable gridFileName. The parameter filename= is required for all grid functions and can be used in a dictionary. Dictionaries are useful as scripts become more complex.

```
#
#   Grid file names can be long, assigning the value to a variable
#   makes a script easier to read
#
gridFileName = os.path.join(gridPath,
  'RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0006_000_NA_XXX_XXXX_XXX.grib2')

#
#    Create a dictionary to define the file name to be used. The parameter
#    filename is used with many of the grid functions.
#
modelData = dict(
    filename=gridFileName
)
```

The contents of the grid file can be listed using the function **listGridFieldsInFile**. List all the fields found in our grid file.

```
#
#    List all the fields in the grib file
#
listGridFieldsInFile(**modelData)
```

The listing contains both a field name and a description. The description is seen when using the GUI. For subsequent grid functions, the field name is used. In most cases, each field contains several different levels. These levels are listed using **listGridLevelsInField**.

```
#
#    Using the v-component field, list all the levels for the v-component
#
listGridLevelsInField(field='v-component_of_wind_isobaric', **modelData)
```

As mentioned above, the files sent across the NOAAPORT feed frequently contain only one forecast hour, this is the case for the files used in this tutorial. Use the function **listGridTimesInField** to list the time in the grid file field **v-component_of_wind_isobaric**:

```
#
#    Again using the v-component field, list all the times available
#
listGridTimesInField(field='v-component_of_wind_isobaric', **modelData)
```

During the image portion of the tutorial, a for loop was used to step through a list of images to create a movie. A method to loop through a list grid is to use the Jython function **glob**. Import the **glob** function.

```
from glob import glob

#
#    Create a list of files
#
fileMatch = os.path.join(gridPath,'*.grib2')
fileList = glob(fileMatch)

print fileList

#
#    List each of the individual file names

for gridFile in fileList:
    print gridFile
```

By concatenating all the files in <*local-path*>**/Data/Scripting/tornado-model/**, we can create a file with multiple times. Concatenating the files works similar to the *Aggregate Grids By Time* **Data Type**. Use the function **listGridTimesInField** with the file <*local-path*>**/Data/Scripting/tornado-model/multiple-times** to create a list of times for the field **v-component_of_wind_isobaric**.

```
#
#    Here is an example of a grib file containing multiple times
#
timesFileName = os.path.join(gridPath, 'multiple-times')
timeList = listGridTimesInField(field='v-component_of_wind_isobaric', filename=timesFileName)

print timeList
```

During the image portion of the tutorial, a for loop was used to step through a list of images to create a movie. A similar set of code could be used to loop the times found in the variable **timeList**.

Using Gridded Data in a McIDAS-V Script – Part 2: Loading and Displaying Grid Data

The information obtained from **listGridFieldsInFile** and **listGridTimesInField** are used as inputs to the function that **loadGrid**. Similar to **loadADDEImage**, **loadGrid** returns a data object that can be manipulated mathematically or used with other functions such as **createLayer**.

As we have done with our previous scripts libraries are imported and variables are initialize to make the scripts platform and user independent. Cut and paste the lines of code onto the **Jython Shell**. This code can be found in the <*local-path*>**/Scripting/grid-display.py** script.

```
import os
import java.awt.Color as Color

#
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users
#

homePath = expandpath('~')
dataPath = os.path.join(homePath,'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')

gridPath = os.path.join(scriptingPath, 'tornado-model')
areaPath = os.path.join(scriptingPath, 'tornado-areas')
irPath = os.path.join(areaPath,'IR')

#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')


#
#   Grid file names can be long, assigning the value to a variable
#   makes a script easier to read
#
gridFileName = os.path.join(gridPath,
    'RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0006_000_NA_XXX_XXXX_XXX.grib2')
```

```
#
#    Create a dictionary to define the file name to be used. The parameter
#    filename is used with many of the grid functions.
#
modelData = dict(
    filename = gridFileName
)

#
#    Load the model data for the following grids:
#        v component of the at 25000 Pa for the verification time of 2011-05-22 @ 21:00:00Z
#        w component of the at 25000 Pa for the verification time of 2011-05-22 @ 21:00:00Z
#
```

The listing function showed that the grid file contains both the u and v components of the wind. Using the **loadGrid** function, data objects of each of these parameters can be created:

```
vWind = loadGrid(field='v-component_of_wind_isobaric',
                 level='25000 Pa', time='2011-05-22 21:00:00Z', **modelData)
uWind = loadGrid(field='u-component_of_wind_isobaric',
                 level='25000 Pa', time='2011-05-22 21:00:00Z', **modelData)
```

Using basic mathematical functions a new data object, windSpeed, can be computed from the u and v components of the wind. As is mentioned in the comment imbedded in the code, there are multiple mathematical methods to calculate the same quantity.

```
#
#    Create a data object of wind speed.
windSpeed = (uWind**2+vWind**2)**.5

#
#    The IDV function sqrt could also be used:
#        windSpeed = sqrt(pow(uWind,2)+pow(vWind,2))
#
```

Through the GUI, there is a grid function to make flow vectors from u and v components of the wind. Mousing over that formula reveals that the function **makeVector** is used. The data object returned from **makeVector** is used to create streamlines or wind fields.

```
#
#    Create a flow vector using the IDV function makeVector
#
flowVectors = makeVector(uWind,vWind)
```

The next parts of the script are similar to the previous image scripts, importing color tables, building a window and creating layers.

```
#
#    Import enhancement tables
#
windSpeedColorFile = os.path.join(enhancementPath, 'Tornado-Jet-Wind-Speed.xml')
windSpeedColorTable = importColorTable(windSpeedColorFile, overwrite=True)
windSpeedTableName = windSpeedColorTable.getName()

#
#    Build a window with a height of 500 pixels and width of 600 pixels
#
Panel = buildWindow(height=700,width=900)

#
#    Add individual layers to the existing window and set enhancement table and data ranges
#
windSpeedLayer = panel[0].createLayer('Image Display', windSpeed)
windSpeedLayer.setEnhancement(windSpeedTableName, range=(0,80))
windSpeedLayer.setLayerLabel(visible=False)
```

A new display type, '**Streamline Plan View**', is used to display streamlines. At the top of the script we imported the library import java.awt.Color as Color. Code from this library provides the capability of changing the color of the streamlines.

```
streamLineLayer = panel[0].createLayer('Streamline Plan View', flowVectors)
streamLineLayer.setColor(Color.cyan)
streamLineLayer.setLineWidth(2)
streamLineLayer.setLayerLabel(visible=False)

#
#   Set the center latitude, longitude and scale
#
panel[0].setProjection('US>Central U.S.')
panel[0].setCenter(35, -97, scale=1.5)

filename = os.path.join(imagePath, 'isotachs-streamlines.gif')
panel[0].captureImage(filename)
```

40. Use **listGridFieldsInFile** to determine the correct parameter name for CAPE (Convective Available Potential Energy).

41. Open a text editor (e.g. gedit, vi, WordPad), and start with the file <*local-path*>**/Data/Scripting/grid-display.py** to write a script that does the following:

    a. Uses the glob function to step through all NOAAPORT grid files and create a movie showing (in order from bottom to top):
        1. image sequence of CAPE
            - applying the enhancement table found in **Tornado-Cape.xml**
            - stretching the enhancement table across a range of 4000 to 7550
        2. image sequence of wind speed
            - applying the enhancement table found in **Tornado-Jet-Wind-Speed.xml**
            - stretching the enhancement table across a range of 0 to 80
        3. loop of streamlines
            - color cyan
            - line width of 2
    b. Imports the color table 'Tornado-Cape.xml
    c. Sets the projection to the Central U.S.
    d. Changes the center point to 35N 97W with a scale factor of 1.5
    e. Turns off the wireframe box
    f. Saves the movie as <loc*al path*>**/Data/Scripting/Images/winds-cape.gif**

An example solution is available at <*local-path*>**/Data/Scripting/grid-exercise.py**. However, before checking the solution, it is recommended that you try to complete the tasks on your own.

42. The last exercise requires you to write a script combining both grid and image data types. The script should do the following:

    a. Uses the glob function to step through all the NOAAPORT grid files
    b. Uses the position number keyword with **loadADDEImage** to specify the image matching the time of the gridded data
    c. Imports the enhancement table from <*local*-path>/**Data/Scripting/Color-Enhancements/Tornado-IR.xml**
    d. Imports the enhancement table from <*local*-path>/**Data/Scripting/Color-Enhancements/Tornado-basemap.xml**
    e. Imports the enhancement table from <*local*-path>/**Data/Scripting/Color-Enhancements/ Tornado-Jet-Wind-Speed.xml**
    f. Creates and uses the local McIDAS AREA dataset for a base map
        - name the dataset **TORNADO**
        - assign the image type '**Land Sea Mask**'
        - data is located <*local*-path>/**Data/Scripting/tornado-areas/BASE**
        - use the enhancement table from **Tornado-basemap.xml**
        - stretch the range from 0 to 255

g. Creates an image sequence of the local data files from the **TORNADO** '**GOES-13 IR**' dataset
    - a mask is applied to temperature values of data greater than 250 K
    - use the enhancement table from **Tornado-IR.xml**
    - stretch the range from the specified temperature threshold to 200
h. Creates an image sequence of wind speed
    - apply the enhancement table found in **Tornado-Jet-Wind-Speed.xml**
    - stretch the range of 0 to 80
    - apply the enhancement table found in Tornado-Jet-Wind-Speed.xml
i. Creates a loop of streamlines
    - use a line width of 2
    - use a color of Cyan
j. Adds a layer label to the IR data set which includes:
    - the text 'Joplin Tornado'
    - font size of 12
    - color White
    - display name
k. Adds a layer label to the wind speed dataset which includes:
    - the text 'Wind Speed and Streamlines'
    - style of NONE
    - font size of 12
    - color White
l. Adds a layer label to the stream line dataset which includes:
    - font size of 12
    - color White
    - timestamp
m. Adds a color scale for the IR data set
n. Adds the annotation 'Joplin, Missouri'
1. Places the text left and center justified located at 37.15N and 94.5W
o. Turns wireframe off
p. Sets the projection to Central U.S.
q. Changes the center point to 35N and 97W with a scale factor of 1.5
r. Saves the movie with the file name of <loc*al path*>**/Data/Scripting/Images/final-exercise.gif**

An example solution is available at <*local-path*>**/Data/Scripting/final-exercise.py**. However, before checking the solution, it is recommended that you try to complete the tasks on your own. Note, creating time matched loops of multiple data types can be challenging and is currently only available through the GUI. Development for this capability within the scripting environment may be done at a later date.

Files Used In This Tutorial

ADDE-dictionary.txt

```
#   This example assumes that the TORNADO dataset has been
#   defined on your workstation in the local ADDE Data Manager
#   <local path>/Data/Scripting/areas-files/IR
#
#   Create a dictionary to be used with loadADDEImage.
#   (remember the 4 space indentation is required)
#
irLocalDataSet = getLocalADDEEntry(dataset='TORNADO', imageType='GOES-13 IR')

ADDE_IR_loadRequest = dict(
    server = 'localhost',
    localEntry = irLocalDataSet,
    size = 'ALL',
    time = ('23:45:00','23:45:00'),
    day = '2011142',
    unit = 'BRIT',
)

#
#   Make an ADDE request for infrared data using keyword=parameter
#   pairs and the dictionary.
#

irData = loadADDEImage(band=4, **ADDE_IR_loadRequest)

#
#   The ** before the dictionary tells python to evaluate the contents of the
#   dictionary and include the keyword=parameter with the request to
#   loadADDEImage. Note, the dictionary must be the last parameter specified.
#
```

importColorTable.txt

```
def importColorTable(filename, name=None, category=None, overwrite=False):
    """Import color table using the given path.

    If the color table in question was exported from the Color Table Manager,
    the name and category parameters will be set to values within the file.

    If the name already exists within the category, a unique name will be
    generated. The format is <name>_<integer>.

    If the given parameters match a system color table and overwriting is
    enabled, the specified color table will merely take precedence over the
    system color table. Removing the local color table will result in the
    system color table being made available.
    Args:
        filename: Path to color table to import.

        name: Name of the color table. If not specified, name will be the
              "base" filename without an extension (e.g. foo.et becomes foo).

        category: Category of the color table. If not specified, the category
                  will default to Basic.

        overwrite: Optional value that controls whether or not an existing
                   color table that matches all the given parameters will be
                   overwritten. Default value is False.
    Returns:
        Either the imported color table or nothing if there was a problem.
    """
    from ucar.unidata.util import IOUtil
    from ucar.unidata.util import ResourceManager
    from ucar.unidata.xml import XmlEncoder

    mcv = getStaticMcv()
    if mcv:
        makeUnique = not overwrite
        ctm = mcv.getColorTableManager()
        tables = ctm.processSpecial(filename, name, category)
        if tables:
            return _ColorTable(ctm.doImport(tables, makeUnique))
        else:
            xml = IOUtil.readContents(filename, ResourceManager.__class__)
            if xml:
                obj = XmlEncoder().toObject(xml)
                return _ColorTable(ctm.doImport(obj, makeUnique))
```

function.py

```
import os
#
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users
#

homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath=os.path.join(scriptingPath, 'Color-Enhancements')

areaPath = os.path.join(scriptingPath, 'tornado-areas')
irPath = os.path.join(areaPath, 'IR')

#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')

#
#   This example gets the information from the dataset created previously in the tutorial
#
irLocalDataSet = getLocalADDEEntry('TORNADO', 'GOES-13 IR')
ADDE_IR_loadRequest = dict(
    debug=True,
    server='localhost',
    localEntry=irLocalDataSet,
    size='ALL',
    time=('23:45:00','23:45:00'),
    day='2011142',
    unit='TEMP',
)

irData = loadADDEImage(**ADDE_IR_loadRequest)

#
#   assign a temperature threshold used with mask() function
#
temperatureThreshold = 250.0
```

```
#
#   Applying a mask is a two part process.
#   First we assign a value of 1 or missing value to a temporary data object
#   Second, multiply the first results to the temporary data object
#
maskedData = mask(irData, 'lt', temperatureThreshold, 1)
finalDataSet = mul(irData, maskedData)

#
#   Import enhancement table
#
IRColorTableFile = os.path.join(enhancementPath, 'Tornado-IR.xml')
IRTable = importColorTable(IRColorTableFile)
IRTableName = IRTable.getName()

#
#   Build a window and turn off the wireframe box
#
bwLines = irData['lines'] / 2
bwEles = irData['elements'] / 2
panel = buildWindow(height=bwLines, width=bwEles)
panel[0].setWireframe(False)

#
#   Add layers to the existing window set enhancement table and data ranges
#
irLayer = panel[0].createLayer('Image Display', finalDataSet)
irLayer.setLayerLabel('GOES-13 Temperatures less than ' + str(temperatureThreshold) + ' %timestamp%')
irLayer.setEnhancement(IRTableName, range=(temperatureThreshold,200))

#
#   Set the center latitude, longitude and scale
#
panel[0].setProjection('US>States>N-Z>Oklahoma')
panel[0].setCenter(33, -97, scale=.5)

fileName=os.path.join(imagePath, 'ir-image.gif')
panel[0].captureImage(fileName)
```

image-movie.py

```
import os
#
#   The ** before the dictionary tells python to evaluate the contents of the
#   dictionary and include the keyword=parameter with the request to
#   loadADDEImage. Note, the dictionary must be the last parameter specified.
#
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users
#
homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')

areaPath = os.path.join(scriptingPath, 'tornado-areas')
irPath = os.path.join(areaPath, 'IR')

#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')

#
#   assign a temperature threshold used with mask() function
#
temperatureThreshold = 250.0

#
#    Initialize a python list
#
myLoop=[]

#
#   Create a dictionary for requesting images
#
irLocalDataSet = getLocalADDEEntry(dataset='TORNADO', imageType='GOES-13 IR')
parms = dict(
    server='localhost',
    localEntry=irLocalDataSet,
    position='ALL'
)
```

```
#
#   Create a list of all available Images using listADDEImageTimes
#
dateTimeList = listADDEImageTimes(**parms)

#
#   listADDEImages was successful, so now try loadADDEImage for each of the
#   directories returned.  There may be occasions when the loadADDEImage fails
#   but we want to continue
#
for dateTime in dateTimeList:

    imageTime = dateTime['time']
    print dateTime['time']

    ADDE_IR_loadRequest = dict(
        localEntry=irLocalDataSet,
        day=dateTime['day'],
        time=(imageTime,imageTime),
        band=4,
        unit='TEMP',
        size='ALL',
    )

    irData = loadADDEImage(**ADDE_IR_loadRequest)

#
#   Applying a mask is a two part process.
#   First we assign a value of 1 or missing value to a temporaary data object
#   Second, multiply the first results to the temporary data object
#
    maskedData = mask(irData, 'lt', temperatureThreshold, 1)
    finalDataSet = mul(irData, maskedData)

    myLoop.append(finalDataSet)

#
#   Import enhancement table
#
IRColorTableFile=os.path.join(enhancementPath,'Tornado-IR.xml')
IRTable=importColorTable(IRColorTableFile,overwrite=True)
IRTableName=IRTable.getName()
```

```
#
#   Build a window and turn off the wireframe box
#
bwLines = irData['lines'] / 2
bwEles = irData['elements'] / 2
panel = buildWindow(height=bwLines, width=bwEles)
panel[0].setWireframe(False)

#
#   Add layers to the existing window set enhancement table and data ranges
#
irLayer = panel[0].createLayer('Image Sequence Display', myLoop)
irLayer.setLayerLabel('GOES-13 Temperatures less than ' + str(temperatureThreshold) + ' %timestamp%')
irLayer.setEnhancement(IRTableName, range=(temperatureThreshold,200))


#
#   Set the center latitude, longitude and scale
#
panel[0].setProjection('US>States>N-Z>Oklahoma')
panel[0].setCenter(33,-97,scale=.5)

fileName = os.path.join(imagePath, 'ir-loop.gif')
writeMovie(fileName)
```

image-exercise.py

```
import os
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users
homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')

areaPath = os.path.join(scriptingPath, 'tornado-areas')
irPath = os.path.join(areaPath, 'IR')
basePath = os.path.join(areaPath, 'BASE')
#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')
#
#   assign a temperature threshold used with mask() function
#
temperatureThreshold = 250.0
#
#     Create a dictionary for a basemap image
#
baseMapDataSet = makeLocalADDEEntry(dataset='TORNADO', imageType='Land Sea Mask', mask=basePath, format='McIDAS Area',
save=True)
baseMapParms = dict(
    server='localhost',
    localEntry=baseMapDataSet,
    size='ALL'
)
baseMapData = loadADDEImage(**baseMapParms)

irLocalDataSet = getLocalADDEEntry('TORNADO', 'GOES-13 IR')
ADDE_IR_loadRequest = dict(
    debug=True,
    server='localhost',
    localEntry=irLocalDataSet,
    size='ALL',
    mag=(1,1),
    position='ALL',
    unit='TEMP',
)
```

```
irData = loadADDEImage(**ADDE_IR_loadRequest)

#
#    Initialize a python list
#
myLoop=[]

#
#   Create a list of all available Images using listADDEImageTimes
#
dateTimeList = listADDEImageTimes(**ADDE_IR_loadRequest)

#
#   listADDEImages was successful, so now try loadADDEImage for each of the
#   directories returned.  There may be occasions when the loadADDEImage fails
#   but we want to continue
#
for dateTime in dateTimeList:

    imageTime = dateTime['time']
    print dateTime['time']

    ADDE_IR_loadRequest = dict(
        localEntry=irLocalDataSet,
        day=dateTime['day'],
        time=(imageTime,imageTime),
        band=4,
        unit='TEMP',
        size='ALL',
    )

    irData = loadADDEImage(**ADDE_IR_loadRequest)
#
#   Applying a mask is a two part process.
#   First we assign a value of 1 or missing value to a temporary data object
#   Second, multiply the first results to the temporary data object
#
    maskedData = mask(irData, 'lt', temperatureThreshold, 1)
    finalDataSet = mul(irData, maskedData)

    myLoop.append(finalDataSet)
```

```
#
#   Import enhancement tables
#
basemapTableFile = os.path.join(enhancementPath, 'Tornado-Basemap.xml')
basemapTable = importColorTable(basemapTableFile, overwrite=True)
basemapTableName = basemapTable.getName()

IRColorTableFile=os.path.join(enhancementPath,'Tornado-IR.xml')
IRTable=importColorTable(IRColorTableFile,overwrite=True)
IRTableName=IRTable.getName()

#
#   Build a window and turn off the wireframe box
#
bwLines = 700
bwEles = 1000
panel = buildWindow(height=bwLines, width=bwEles)
panel[0].setWireframe(False)

#
#   Add individual layers to the existing window and set enhancement table and data ranges
#   Note that the layer order is important
#
baseMapLayer = panel[0].createLayer('Image Display', baseMapData)
baseMapLayer.setEnhancement(basemapTableName, range=(0,255))
baseMapLayer.setLayerLabel(' ', visible=False)

irLayer = panel[0].createLayer('Image Sequence Display', myLoop)
irLayer.setLayerLabel('%longname% Joplin Tornado Temperatures less than ' + str(temperatureThreshold) + ' %timestamp%')
irLayer.setEnhancement(IRTableName,range=(temperatureThreshold, 200))

irLayer.setColorScale(visible=True, placement='Top', showUnit=True)

#
#   Set the center latitude, longitude and scale
#
panel[0].setProjection('US>Central U.S.')
panel[0].setCenter(35, -97, scale=1.5)

panel[0].annotate('Joplin, Missouri - <b>&gt</b>',lat=37.15, lon=-94.5,size=18,
font='SansSerif',alignment=('left','center'),color='White')
fileName=os.path.join(imagePath,'image-exercise.gif')
writeMovie(fileName)
```

stats.py

```
import os
import csv

#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users
#

homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')
statisticsPath = os.path.join(scriptingPath, 'Statistics')

areaPath = os.path.join(scriptingPath, 'tornado-areas')
irPath = os.path.join(areaPath, 'IR')

#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')

#
#    The easiest way to make an ADDE request is to create a dictionary
#    That defines your parameters.  Here we have a generic request
#
irLocalDataSet = getLocalADDEEntry('TORNADO', 'GOES-13 IR')
ADDE_IR_loadRequest = dict(
    server='localhost',
    localEntry=irLocalDataSet,
    place=Places.CENTER,
    size=(100,200),
    coordinateSystem=LATLON,
    location=(37.0,-94.5),
    mag=(1, 1),
    unit='TEMP',
)

#
#    Assign a temperature threshold used with mask() function
#
temperatureThreshold = 250.0
```

```
#
#    Define the output file names
#
textFileName = os.path.join(statisticsPath, "stats.txt")
textOutputFile = open(textFileName, "w")

csvFileName = os.path.join(statisticsPath, "stats.csv")
csvOutputFile = open(csvFileName, "wb")

csvData = csv.writer(csvOutputFile, delimiter=",")
csvData.writerow(["Time", "latitude", "longitude", "geometricMean", "min", "median", "max", "kurtosis", "numPoints",
"skewness", "stdDev", "variance"])

#
#     Now make the request using the function loadADDEImage
#     This returns an object containing data and metadata
#
for pos in range(-4,1):
    irData = loadADDEImage(position=(pos),band=4, **ADDE_IR_loadRequest)

#
#   Applying a mask is a two part process.
#   First we assign a value of 1 or missing value to a temporaary data object
#   Second, multiply the first results to the temporary data object
#
    maskedData = mask(irData, 'lt', temperatureThreshold, 1)
    finalDataSet = mul(irData, maskedData)

#
#     pass the irData into the Statistics package
#
    stats = Statistics(finalDataSet)

#
#     open a file and write out the statistics data
#
    textOutputFile.write("   stat and value for: %s \n" % irData["nominal-time"])
    textOutputFile.write("   geometric mean: %s \n" % stats.geometricMean())
    textOutputFile.write("   kurtosis: %s \n" % stats.kurtosis())
    textOutputFile.write("   num points: %s \n" % stats.numPoints())
    textOutputFile.write("   skewness: %s \n" % stats.skewness())
    textOutputFile.write("   std dev: %s \n" % stats.standardDeviation())
    textOutputFile.write("   variance: %s \n" % stats.variance())
    textOutputFile.write("\n")
```

```
#
#     import the csv library for writing out the
#     statistics values
#
    theTime = str(irData["nominal-time"])[11:16]
    csvData.writerow([theTime, "37.0", "-94.5", stats.geometricMean(), stats.min(), stats.median(), stats.max(),
stats.kurtosis(),
        stats.numPoints(), stats.skewness(), stats.standardDeviation(), stats.variance()])

csvOutputFile.close()
textOutputFile.close()

#
#     The last section of the script will annotate an image
#     with the information from the statistics package.
#     Create some strings from the data object to be able
#     to annotate our window with the stats values.
#
min = 'min: %s' % (
    stats.min()
    )
max = 'max: %s' % (
    stats.max()
    )
stddev = 'std dev: %s' % (
    stats.standardDeviation()
    )
geomean = 'geometric mean: %s' % (
    stats.geometricMean()
    )
numpoints = 'num points: %s' % (
    stats.numPoints()
    )

#     Create a string from the data to make it
#     easier to label the image.
#
irLabel = '%s %s' % (
    irData['sensor-type'],
    irData['nominal-time']
    )
```

```
#
#     Build a window with a single panel
#
panel = buildWindow(height=600,width=900)

#
#   Import enhancement table
#

#
#     Create a layer from the infrared data object
#
IRColorTableFile = os.path.join(enhancementPath,  'Tornado-IR.xml')
IRTable = importColorTable(IRColorTableFile, overwrite=True)
IRTableName = IRTable.getName()

#
#     When changing attributes, some are panel based and
#     others are layer based.  In the following steps, they are:
#
#     Change the projection (panel)
#     Turn off the wire frame box (panel)
#     Change the center point (panel)
#     Add the statistics values (panel)
#     Add a layer label (layer)
#     Save the output file (panel)
#
irLayer = panel[0].createLayer('Image Display', finalDataSet)
irLayer.setEnhancement(IRTableName, range=(temperatureThreshold,200))
irLayer.setColorScale(visible=True, placement='Top', showUnit=True, size=20)
irLayer.setLayerLabel(label=irLabel, size=14)

panel[0].setProjection('US>States>N-Z>Oklahoma')
panel[0].setCenter(33,-97,scale=.5)
panel[0].setWireframe(False)
panel[0].annotate(min, line=430,element=30, size=18, color='White', alignment=('right','center'))
panel[0].annotate(max, line=460,element=30, size=18, color='White', alignment=('right','center'))
panel[0].annotate(stddev, line=490,element=30, size=18, color='White', alignment=('right','center'))
panel[0].annotate(geomean, line=520,element=30, size=18, color='White', alignment=('right','center'))
panel[0].annotate(numpoints, line=550,element=30, size=18, color='White', alignment=('right','center'))

fileName = os.path.join(imagePath, 'stats-image.jpg')
panel[0].captureImage(fileName)
```

grid-list.py

```
import os
from glob import glob
#
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users
#
#     Some of the path definitions are not not used with in the script,
#     but rather in subsequent scripts
#
homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')

gridPath = os.path.join(scriptingPath, 'tornado-model')
areaPath = os.path.join(scriptingPath, 'tornado-areas')
irPath = os.path.join(areaPath, 'IR')

#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')

#
#   Grid file names can be long, assigning the value to a variable
#   makes a script easier to read
#
gridFileName = os.path.join(gridPath, 'RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0006_000_NA_XXX_XXXX_XXX.grib2')

#
#    Create a dictionary to define the file name to be used. The parameter
#    filename is used with many of the grid functions.
#
modelData=dict(
    filename=gridFileName
)

#
#    List all the fields in the grib file
#
listGridFieldsInFile(**modelData)
```

```
#
#    Using the v-component field, list all the levels for the v-component
#
listGridLevelsInField(field='v-component_of_wind_isobaric', **modelData)

#
#    Again using the v-component field, list all the times available
#
listGridTimesInField(field='v-component_of_wind_isobaric', **modelData)


#
#    Create a list of files
#
fileMatch = os.path.join(gridPath, '*.grib2')
fileList = glob(fileMatch)

print fileList

#
#    List each of the individual file names

for gridFile in fileList:
    print gridFile

#
#    Here is an example of a grib file containing multile times
#
timesFileName = os.path.join(gridPath, 'multiple-times')
timeList = listGridTimesInField(field='v-component_of_wind_isobaric', filename=timesFileName)

print timeList
```

grid-display.py

```
import os
import java.awt.Color as Color
#
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users
#
homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')

gridPath = os.path.join(scriptingPath, 'tornado-model')
areaPath = os.path.join(scriptingPath, 'tornado-areas')
irPath = os.path.join(areaPath, 'IR')
#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')
#
#   Grid file names can be long, assigning the value to a variable
#   makes a script easier to read
#
gridFileName = os.path.join(gridPath, 'RUC-USLC13KM_105_2011_142_05_22_RH1500_FH0006_000_NA_XXX_XXXX_XXX.grib2')
#
#    Create a dictionary to define the file name to be used. The parameter
#    filename is used with many of the grid functions.
#
modelData = dict(
    filename = gridFileName
)
#
#    Load the model data for the following grids:
#        v component of the at 25000 Pa for the verification time of 2011-05-22 @ 21:00:00Z
#        w component of the at 25000 Pa for the verification time of 2011-05-22 @ 21:00:00Z
#
vWind = loadGrid(field='v-component_of_wind_isobaric', level='25000 Pa', time='2011-05-22 21:00:00Z', **modelData)
uWind = loadGrid(field='u-component_of_wind_isobaric', level='25000 Pa', time='2011-05-22 21:00:00Z', **modelData)
```

```
#
#    Create a data object of wind speed.
windSpeed = (uWind**2+vWind**2)**.5
#
#    The IDV function sqrt could also be used:
#        windSpeed=sqrt(pow(uWind,2)+pow(vWind,2))
#

#
#    Create a flow vector using the IDV function makeVector
#
flowVectors = makeVector(uWind,vWind)

#
#    Import enhancement tables
#
windSpeedColorFile = os.path.join(enhancementPath, 'Tornado-Jet-Wind-Speed.xml')
windSpeedColorTable = importColorTable(windSpeedColorFile, overwrite=True)
windSpeedTableName = windSpeedColorTable.getName()

#
#    Build a window with a height of 500 pixels and width of 600 pixels
#
panel = buildWindow(height=700,width=900)

#
#    Add individual layers to the existing window and set enhancement table and data ranges
#
windSpeedLayer = panel[0].createLayer('Image Display', windSpeed)
windSpeedLayer.setEnhancement(windSpeedTableName, range=(0,80))
windSpeedLayer.setLayerLabel(visible=False)

streamLineLayer = panel[0].createLayer('Streamline Plan View', flowVectors)
streamLineLayer.setColor(Color.cyan)
streamLineLayer.setLineWidth(2)
streamLineLayer.setLayerLabel(visible=False)

#
#   Set the center latitude, longitude and scale
#
panel[0].setProjection('US>Central U.S.')
panel[0].setCenter(35, -97, scale=1.5)

fileName = os.path.join(imagePath,'isotachs-streamlines.gif')
panel[0].captureImage(fileName)
```

grid-exercise.py

```
import os
from glob import glob
import java.awt.Color as Color

#
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users^
#

homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')

gridPath = os.path.join(scriptingPath, 'tornado-model')

#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath, 'Images')

#
#    Initialize a movie loops for wind speed and streamlines
#
windSpeedLoop = []
streamLineLoop = []
capeLoop = []

#
#    Loop through all the files to make a movie
#
fileMatch = os.path.join(gridPath, '*.grib2')
fileList = glob(fileMatch)

for gridFile in fileList:
```

```
#
#        Load the model data for the following grids:
#            v component of the at 25000 Pa
#            w component of the at 25000 Pa
#
    print gridFile
    vWind = loadGrid(field='v-component_of_wind_isobaric', level='25000 Pa', filename=gridFile)
    uWind = loadGrid(field='u-component_of_wind_isobaric', level='25000 Pa', filename=gridFile)
    cape = loadGrid(field='Convective_available_potential_energy_surface', level='25000 Pa', filename=gridFile)

#
#        Create a data object of wind speed.
#
    windSpeed = (uWind**2+vWind**2)**.5
    windSpeedLoop.append(windSpeed)

#
#    The IDV function sqrt could also be used:
#        windSpeed = sqrt(pow(uWind,2)+pow(vWind,2))
#

#
#       Create a flow vector using the IDV function makeVector
#
    flowVectors = makeVector(uWind, vWind)
    streamLineLoop.append(flowVectors)

#
#    Add new forecast hour for CAPE to loop
#
    capeLoop.append(cape)

#
#    Import enhancement tables
#
windSpeedColorFile = os.path.join(enhancementPath, 'Tornado-Jet-Wind-Speed.xml')
windSpeedColorTable = importColorTable(windSpeedColorFile, overwrite=True)
windSpeedTableName = windSpeedColorTable.getName()

capeColorFile = os.path.join(enhancementPath,'Tornado-Cape.xml')
capeColorTable = importColorTable(capeColorFile, overwrite=True)
capeTableName = capeColorTable.getName()
```

```
#
#    Build a window with a height of 500 pixels and width of 600 pixels
#
panel = buildWindow(height=700,width=900)

#
#    Add individual layers to the existing window and set enhancement table and data ranges
#
capeLayer = panel[0].createLayer('Image Sequence Display', capeLoop)
capeLayer.setEnhancement(capeTableName, range=(4000,7550))
capeLayer.setLayerLabel(visible=False)

windSpeedLayer = panel[0].createLayer('Image Sequence Display', windSpeedLoop)
windSpeedLayer.setEnhancement(windSpeedTableName, range=(0,80))
windSpeedLayer.setLayerLabel(visible=False)

streamLineLayer = panel[0].createLayer('Streamline Plan View', streamLineLoop)
streamLineLayer.setColor(Color.cyan)
streamLineLayer.setLineWidth(2)
streamLineLayer.setLayerLabel(visible=False)

#
#   Set the center latitude, longitude and scale
#
panel[0].setProjection('US>Central U.S.')
panel[0].setCenter(35, -97, scale=1.5)

fileName = os.path.join(imagePath, 'winds-cape.gif')
writeMovie(fileName)
```

final-exercise.py

```
import os
from glob import glob
import java.awt.Color as Color

#
#     Setting up a variable to specify the location of your final images
#     makes your script easier to read and more portable when you share it
#     with other users^
#

homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')

enhancementPath = os.path.join(scriptingPath, 'Color-Enhancements')

gridPath = os.path.join(scriptingPath, 'tornado-model')

#
#   imagePath is the directory to store final images
#   and/or animated gif files
#
imagePath = os.path.join(scriptingPath,'Images')

#
#   assign a temperature threshold used with mask() function
#
temperatureThreshold = 250.0

#
#     Create a dictionary for a basemap image
#
baseMapDataSet = makeLocalADDEEntry(dataset='TORNADO', imageType='Land Sea Mask', mask=basePath, format='McIDAS Area',
save=True)
baseMapParms = dict(
    server='localhost',
    localEntry=baseMapDataSet,
    size='ALL'
)

#
#    Request basemap data
#
baseMapData = loadADDEImage(**baseMapParms)
```

```
#
#    Get local data set information for IR and
#    create a dictionary for the IR data - this will be used in the loop
#
irLocalDataSet = getLocalADDEEntry('TORNADO', 'GOES-13 IR')
ADDE_IR_loadRequest = dict(
    debug=True,
    server='localhost',
    localEntry=irLocalDataSet,
    size='ALL',
    mag=(1,1),
    unit='TEMP',
)

#
#    Initialize a position number to be used for requestiong IR data
#
irPos = -4

#
#    Initialize a movie loops for wind speed and streamlines and ir data
#
windSpeedLoop = []
streamLineLoop = []
irLoop = []

#
#    Loop through all the files to make a movie
#
fileMatch = os.path.join(gridPath, '*.grib2')
fileList = glob(fileMatch)

for gridFile in fileList:
#
#        Load the model data for the following grids:
#            v component of the at 25000 Pa
#            w component of the at 25000 Pa
#
    vWind = loadGrid(field='v-component_of_wind_isobaric', level='25000 Pa', filename=gridFile)
    uWind = loadGrid(field='u-component_of_wind_isobaric', level='25000 Pa', filename=gridFile)
```

```
#
#        Create a data object of wind speed.
#
    windSpeed = (uWind**2+vWind**2)**.5
    windSpeedLoop.append(windSpeed)
#
#    The IDV function sqrt could also be used:
#        windSpeed = sqrt(pow(uWind,2)+pow(vWind,2))
#


#
#       Create a flow vector using the IDV function makeVector
#
    flowVectors = makeVector(uWind, vWind)
    streamLineLoop.append(flowVectors)

#
#   request IR data
#
    irData = loadADDEImage(position=irPos, **ADDE_IR_loadRequest)
    irPos = irPos+1

#
#   Applying a mask is a two part process.
#   First we assign a value of 1 or missing value to a temporary data object
#   Second, multiply the first results to the temporary data object
#
    maskedData = mask(irData, 'lt', temperatureThreshold, 1)
    finalIRData = mul(irData, maskedData)

#
#    Add IR data to loop
#
    irLoop.append(finalIRData)

#
#    Import enhancement tables
#
windSpeedColorFile = os.path.join(enhancementPath, 'Tornado-Jet-Wind-Speed.xml')
windSpeedColorTable = importColorTable(windSpeedColorFile, overwrite=True)
windSpeedTableName = windSpeedColorTable.getName()

basemapTableFile = os.path.join(enhancementPath, 'Tornado-Basemap.xml')
basemapTable = importColorTable(basemapTableFile, overwrite=True)
basemapTableName = basemapTable.getName()
```

```
IRColorTableFile = os.path.join(enhancementPath, 'Tornado-IR.xml')
IRTable = importColorTable(IRColorTableFile, overwrite=True)
IRTableName = IRTable.getName()

#
#    Build a window with a height of 500 pixels and width of 600 pixels
#
panel = buildWindow(height=700, width=900)

#
#    Add individual layers to the existing window and set enhancement table and data ranges
#
baseMapLayer = panel[0].createLayer('Image Display', baseMapData)
baseMapLayer.setEnhancement(basemapTableName, range=(0,255))

irLayer = panel[0].createLayer('Image Sequence Display', irLoop)
irLayer.setEnhancement(IRTableName, range=(temperatureThreshold,200))

irLayer.setColorScale(visible=True, placement='Top', showUnit=True)

windSpeedLayer = panel[0].createLayer('Image Sequence Display', windSpeedLoop)
windSpeedLayer.setEnhancement(windSpeedTableName, range=(0,80))

streamLineLayer = panel[0].createLayer('Streamline Plan View', streamLineLoop)
streamLineLayer.setColor(Color.cyan)
streamLineLayer.setLineWidth(2)
streamLineLayer.setLayerLabel(visible=True)

baseMapLayer.setLayerLabel(' ', visible=False)
irLayer.setLayerLabel(label='%longname% Joplin Tornado IR Brightness Temperatures less than ' +
str(temperatureThreshold) + 'K', style='NONE', size=12, color='White')
windSpeedLayer.setLayerLabel(label='Wind Speed and Streamlines', style='NONE', size=12, color='White')
streamLineLayer.setLayerLabel(label='%timestamp%')

panel[0].annotate('Joplin, Missouri - <b>&gt</b>', lat=37.15, lon=-94.5,size=18, font='SansSerif',
alignment=('left','center'), color='White')
#
#   Set the center latitude, longitude and scale
#
panel[0].setProjection('US>Central U.S.')
panel[0].setCenter(35, -97, scale=1.5)

fileName = os.path.join(imagePath, 'final-example.gif')
writeMovie(fileName)
```