McIDAS-V Tutorial

An Introduction to Jython Scripting and Data Analysis

updated April 2024 (software version 1.9)

McIDAS-V is a free, open source, visualization and data analysis software package that is the next generation in SSEC's 50-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 a java implementation of Python called Jython. The McIDAS-V scripting 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 - https://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 - https://mcidas.ssec.wisc.edu/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 have access to the functions we need 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

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.

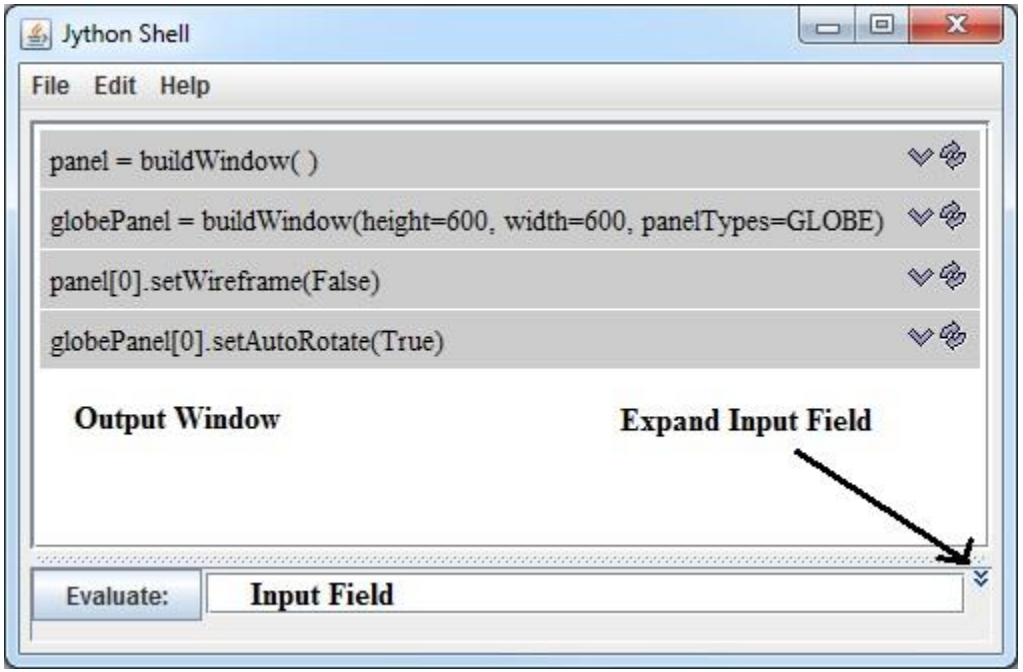
In summary, Jython's language and standard library are the same as CPython, but many non-standard Python libraries such as NumPy are not available in Jython. However, McIDAS-V has good alternatives for data access, plotting, and numerical computation.

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 of **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 single or 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 the **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 created a window, just as you would using the GUI with ***File -> New Display Window…***.

2. 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 the 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 was 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** operated 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*.

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 to edit the object.

4. Click the **Expand Input Field** icon to the right of the *input field* so multiple lines can be entered into the **Jython Shell**. Enter the following into the **Jython Shell**.

Create an object called myList. In the *input field*, type: `myList = [1, 2, 3]` `print(myList)` Click **Evaluate**.	This code results in: *[1, 2, 3]*
Append an item to the object. In the *input field*, type: `myList.append(4)` `print(myList)` Click **Evaluate**.	This code results in: *[1, 2, 3, 4]*
Remove an item from the object. In the *input field*, type: `myList.remove(2)` `print(myList)` Click **Evaluate**.	This code results in: *[1, 3, 4]*

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**.

5. Create an image window, and adjust the size of the window with **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)
- ***Python documentation*** (https://docs.python.org/2/), especially the tutorial (https://docs.python.org/2/tutorial/)
- ***Style Guide for Python Code*** (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 syntax for setting a projection is similar to the menu structure you see when you change the projection using the GUI in the Main Display. Note that Jython is a case sensitive language, and you must type things exactly as documented here.

   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**.

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 (all one line):
    ```
    here = panel[0].annotate('<b>You Are Here</b>', 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 (all one line):
    ```
    plus = panel[0].annotate('<b>+</b>', 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 step 8a, 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]
for i=0, n_elements(myList)-1 do begin
print, myList[i]
endfor
```

This code results in:

1
2
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

   The only thing wrong with this Python code 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]
   for thing in myList:
       print(thing)
   ```

   Click **Evaluate**.

   This code results in:

   1
   2
   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. "Loop is finished" from the previous example was only printed a single time after the **for** loop. This is because **print endingMessage** 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 bracket. 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. Follow these steps to see 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 and evaluate: `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 and evaluate: `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 and evaluate: `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 and evaluate:

```
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 and evaluate:

```
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

So far, 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.

11. Create local datasets to access the 2011 Joplin tornado infrared imagery. In the *input field* of the **Jython Shell**, type and evaluate (irDataSet line is all one line):
```
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*, type and evaluate:
```
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 the directory information for each image. In the *input field*, type and evaluate (note that "\n" is a new-line character, meaning an empty line will be printed at the beginning of each iteration of the for loop):
```
for imageDir in dirList:
    print '\nNew image directory %s %s' % (imageDir['sensor-type'], imageDir['nominal-time'])
    print '-'*55
    for key,value in imageDir.iteritems():
        print key,value
```

14. Make an ADDE request to get the imagery data from the first keyword parameter pairing returned from **listADDEImages**. **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**. In the *input field*, type and evaluate:
```
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 and evaluate:

```
panel = buildWindow(height=600, width=900, panelTypes=MAP)
dataLayer = panel[0].createLayer('Image Display', imageData)
```

Use the method **captureImage** to save the display to a file. Because McIDAS-V does a screen capture on some platforms, be sure the entire window is showing and is not blocked by other windows, or your resulting image may not be complete. After viewing **IR-Image.jpg** in a browser, close the image window. In the *input field*, type and evaluate:

```
panel[0].captureImage('<local-path>/Data/Scripting/Images/IR-Image.jpg')
```

Creating a Simple Remote ADDE Request

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

16. Request a listing of all images from the dataset TORNADO found on the server pappy.ssec.wisc.edu. Directories returned from a remote **listADDEImages** request are identical to those of a local ADDE request and can be used as inputs to **loadADDEImage**. In the *input field*, type and evaluate (all one line):

```
dirList = listADDEImages(server='pearl.ssec.wisc.edu', dataset='TORNADO', descriptor='GOES13',
position='ALL')
```

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

```
for imageDir in dirList:
    print '\nNew image directory %s %s' % (imageDir['sensor-type'], imageDir['nominal-time'])
    print '-'*55
    for key,value in imageDir.iteritems():
        print key,value
```

Dictionaries in Python

One useful data type in Python is called the **dict**, 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, and the “value” is the definition of the word.

18. 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 and evaluate:

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

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

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

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

```
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. The code results are the same. In the *input field*, type and evaluate:

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

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 and evaluate:

```
sensor_info = {
```

```
    'name': 'VIIRS',
    'bands':      ['SVI01', 'SVM02', 'SVM03'],
    'resolution': ['375m',  '750m',    '750m'],
}
```

Print the keys and values of the bands and resolution. In the *input field*, type and evaluate:

```
for x in range(0,3):
    print 'band:',sensor_info['bands'][x],'resolution:',sensor_info['resolution'][x]
```

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.

19. 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',encoding='UTF-8')**

    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 and evaluate:

    ```
    irLocalDataSet = getLocalADDEEntry(dataset='TORNADO', imageType='GOES-13 IR')
    ```

    b. In the *input field*, type and evaluate (the 4 space indentation is required):

    ```
    ADDE_IR_loadRequest = dict(
        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 and evaluate (Note that you can skip this step if the entire ADDE-dictionary.txt file was evaluated in the previous step): `irData = loadADDEImage(band=4, **ADDE_IR_loadRequest)`

20. **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 and evaluate:

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

21. Now create layer objects for the infrared data. Use **createLayer** with the object **irData**. In the *input field*, type and evaluate:

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

22. 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 and evaluate:

```
irLayer.setEnhancement('Longwave Infrared Deep Convection')
```

23. 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 and evaluate:

```
print irData.keys( )
```

b. In the *input field*, type and evaluate:

```
irLabel = '%s %s' % (
    irData['sensor-type'],
    irData['nominal-time']
)
```

c. In the *input field*, type and evaluate:
```
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.

24. Create a function named **add** and demonstrate its usage with numbers and letters. Hopefully, 2 plus 2 will equal 4. As with loops and **if/else** blocks, indentation/dedent indicate the end of the function. In the *input field*, type and evaluate:

```
def add(a, b):
    return a+b
print add(2,2)
```

25. Similar 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

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

Note that users can share code by adding functions to the **Local Jython Library**. An example of this will be covered later in this tutorial.

Using Functions in a McIDAS-V Script

Building upon the previous examples, the next script uses the **importEnhancement**, **mask** and **mul** functions. **mask** and **mul** 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 to 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 **importEnhancement** 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 = importEnhancement(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) + 'K
%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)
```

27. From the **Jython Shell**, run **function.py**. In the *input field*, type:
    editFile('<*local-path*>**/Data/Scripting/function.py', encoding='UTF-8'**)

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

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

30. Close the display window created by the **function.py** script.

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 of 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 to 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 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)

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

31. From the **Jython Shell** run **image-movie.py**. In the *input field*, type:
    editFile('<*local path*>**/Data/Scripting/image-movie.py', encoding='UTF-8**')

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

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

Creating Your Own McIDAS-V Script

34. 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.

35. 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**

       Note that on Windows "tail" may not be a recognized command in a Command Prompt window. An alternative to this would be to use a Windows PowerShell window and:

       Type: Get-Content <*user-path*>/**McIDAS-V/mcidasv.log** –Wait –Tail 30

       Type: Ctrl+c to escape the "tail" command.

    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**.

Bonus: Run the <*local-path*>/**Data/Scripting/sandwich_example.py** script to create a sandwich (IR overlaid on VIS) GOES-16 sandwich display. The output image will be <*local-path*>/**Data/Scripting/Images/Sandwich.jpg**.

Running Scripts from a Command Prompt – Continued

The previous section evaluated a script from the script from the background with all variables defined within the script. It's possible to evaluate a background script while allowing the user to specify variables at runtime. This is done with the –scriptargs flag, which uses uses standard Python optparse/argparse syntax within the script to specify the arguments. Anything after the scriptargs flag will be considered a scripting argument, so scriptargs should be specified last. The <*user-path*> /**Data/Scripting/user-band.py** script uses argparse at the beginning of the script to define the variable that the user specifies at runtime (band number):

```
from argparse import ArgumentParser
parser=ArgumentParser()
parser.add_argument('-b', '--band', action='store', dest='bnd', type=str, help='Band Number')
args = parser.parse_args()
```

The band number the user specifies at startup is then passed through loadADDEImage, setLayerLabel, and in the filename string for capture Image.

36. From the same Command Prompt/Terminal window used in the last example, run the **user-band.py** script.

    a. Run the **user-band.py** script with the –scriptargs flag to specify a band number of GOES-16 data to load. In this example, band 13 will be specified (note the two dashes before "band"):

       For Unix, type:
       ./runMcV –script <*local-path*>**/Data/Scripting/user-band.py –scriptargs --band 13**

       For Windows, type:
       runMcV.bat –script <*local-path*>**/Data/Scripting/user-band.py –scriptargs --band 13**

    b. From your browser, view the file <*local-path*>**/Data/Scripting/Images/band_*.jpg** (where * is the band number) that was created from <*local-path*>**/Data/Scripting/user-band.py**.

Introduction to Data Manipulation in McIDAS-V

This section of the tutorial will use the term "VisAD data object." A VisAD data object contains the data as well as detailed information about a data's structure and type. These objects can also contain units, temporal and geospatial information. In this section of the tutorial, an introduction to the VisAD data object is presented as well as a general introduction to accessing the information within the data object.

37. Restart McIDAS-V and load the '<*local-path*>**/Data/Scripting/load_grid.py**' script provided with this tutorial into the **Jython Shell** (***Tools -> Formulas -> Jython Shell***). In the *input field* of the **Jython Shell**, type and evaluate:

    ```
    editFile('<local-path>/Data/Scripting/load_grid.py', encoding='UTF-8')
    ```
    Click **Evaluate** (or *Shift+Enter*) to load the script into the **Jython Shell**.
    Click **Evaluate** again to execute the script.

38. Use the standard Jython function **type()** to display the class of the data contained in *h8b1*:

    ```
    print type(h8b1)
    ```
    Click **Evaluate**.

    The *h8b1* object belongs to a class called "*MappedGeoGridFlatField.*" This class is returned when data is loaded via the **loadGrid()** function.

39. The Jython built-in **type** function does not describe how the data is structured. The JPythonMethod library built into McIDAS-V contains a function called "**whatTypes()**". This function describes the structure of any VisAD data object. **whatTypes** can be used for debugging. In the *input field* of the **Jython Shell**, type and evaluate:

    ```
    print whatTypes(h8b1)
    ```
    Click **Evaluate**.

    The *h8b1* object domain contains latitude and longitude coordinates. The latitude and longitude coordinate unit is "degrees". The range of the *h8b1* data object contains the data. In this case, the data are values of albedo. Albedo has no unit (Unit: 1).

40. View the data mapping. Type and evaluate: `print getType(h8b1)`

    The result, ((Longitude, Latitude) -> albedo[unit:1]), displays a map of the albedo data (stored in the range) to the longitude (stored in domain[0]) and latitude coordinates (stored in domain[1]).

41. Determine the size of the data and range of the coordinates. In the **Jython Shell**, type:

```
h8ds = getDomainSet(h8b1); print h8ds
```

Click **Evaluate**.

The semicolon (;) links the commands together. In this way, multiple commands can be entered on one line, and executed in sequence. This output can be interpreted with the help of other commands run earlier. In the previous steps, evaluating the **whatTypes()** function showed that the domain index for the longitude coordinate is 0 and the latitude coordinate is 1. This can also be inferred from the **getType()** command. Therefore, in the output of **getDomainSet()**, Dimension 0 is longitude which has a range of 124.242645 to 125.749756, and Dimension 1 is latitude which has a range of 35.063034 to 36.36338. The **getDomainSet()** output also shows that the length, or number of data points in this data are 441. If a user should use **len(h8b1)**, the length of the *MappedGeoGridFlatField* metadata is returned.

42. Once the data object domain set of a data object is retrieved (remember, this is the set of latitude and longitude coordinates), the latitude and longitude values for each grid point can be accessed. Using the variable *h8ds* created in the previous step, type and evaluate:

```
h8latlons = getLatLons(h8ds)
h8lats = h8latlons[0]
h8lons = h8latlons[1]
```

Note: The JPythonMethod **getLatLons** resets the order of the longitude and latitude coordinates. This means that the latitude values are always returned in the zero index, and the longitudes are always returned as the first index. To get return the shape of the **h8latlons** array, type and evaluate:

```
import Numeric
print Numeric.shape(h8latlons)
```

Note: McIDAS-V's **see** function allows for printing out all of the different options with Numeric. To do this, type and evaluate:

```
print see(Numeric)
```

43. Print the first four latitude points. Note: In the **Jython Shell**, it is a good practice to limit the size of data printed. Type and evaluate:

```
print h8lats[0:4]
```

44. Additional methods for working with this data can be found in JPythonMethods (http://www.ssec.wisc.edu/visad-docs/javadoc/visad/python/JPythonMethods.html). For example, a method to return the min/max of the data is **getMinMax()**. In the **Jython Shell**, type and evaluate:

```
print getMinMax(h8b1)
```

This method returns that the range of albedo values in **h8b1** goes from a minimum value of ~0.234 to a maximum value of ~0.553.

45. An additional JPythonMethod, **getValues()**, can be used to return the data points as float values. The values returned can be saved to a variable named *myData*. In the **Jython Shell**, type and evaluate:

```
myData = getValues(h8b1)
```

a. Determine the type of *myData*. In the **Jython Shell**, type:

```
print type(myData)
```

The type returned is an array.

b. What is the length of the *myData* array? Does it match the length found in the domain set? In the **Jython Shell**, type:

```
print len(myData)
```

The length is 1. This does not match the length found in the domain set in the next command.

c. What is the length of *myData[0]*? In the **Jython Shell**, type:

```
print len(myData[0])
```

The length is 441, which should match the length reported in **getDomainSet(h8b1)** above. The actual shape of this array is a 1x441. Therefore, the first **len()** returns the length of the first dimension of the array *myData*. In this case, since there is only one timestamp in the file, that length is 1. This is similar to a Fortran array of REAL myData(1,441). An alternative way of returning the shape of the array would be to run:

```
print Numeric.shape(myData)
```

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.
#
```

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 = importEnhancement(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) + 'K %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=importEnhancement(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, globalPalette=False)
```

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 = importEnhancement(basemapTableFile, overwrite=True)
basemapTableName = basemapTable.getName()

IRColorTableFile=os.path.join(enhancementPath,'Tornado-IR.xml')
IRTable=importEnhancement(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) + 'K
%timestamp%', size=14)
irLayer.setEnhancement(IRTableName,range=(temperatureThreshold, 200))

irLayer.setColorScale(visible=True, placement='Top', size=28, 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, globalPalette=False)
```

sandwich_example.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')

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

# generic Python dictionary for loadADDEImage as bands 3 and 13
# will both use the same server, dataset, descriptor, and position
addeParms = dict(
    server = 'adde.ucar.edu',
    dataset = 'EAST',
    descriptor = 'CONUS',
    position = -1,
    )

# create a data source for the visible band 3 data
# size and mag are specified to make the resolution of
# this band (1km) match that of the band 13 infrared data (2km)
b3Data = loadADDEImage(band=3, unit='ALB', size=(1500,2500), mag=(-2,-2), **addeParms)

# create a data source for the infrared band 13 data
b13Data = loadADDEImage(band=13, unit='TEMP', size='ALL', **addeParms)

# use the sandwich function to create a rgb data soruce
rgbData = sandwich(b13Data,b3Data)

# build a window to display the data
panel = buildWindow(height=600, width=600)

# display visible data, then overlay with the rgb layer
visLayer = panel[0].createLayer('Image Display', b3Data)
rgbLayer = panel[0].createLayer('RGB Composite', rgbData)
```

```
# lower the gamma value of the RGB layer to make it brighter
rgbLayer.updateGamma(0.6)

# capture an image of the display
fileName=os.path.join(imagePath, 'Sandwich.jpg')
panel[0].captureImage(fileName)
```

load_grid.py

```
# import the jython library used
import os

homeDirectory=expandpath('~')
dataDirectory=os.path.join(homeDirectory,'Data','Scripting','H8')

# Add the filename to the data
band1File=os.path.join(dataDirectory,'HS_H08_20150715_0100_B01_FLDK_subset.nc')

# Initialize the loadGrid parameters.
parms = dict(
    time=0,
    field='albedo',
    xStride = 5,
    yStride = 5,
    xRange = (100,200),
    yRange = (100,200)
    )

# load the data from the file
h8b1=loadGrid(filename=band1File,**parms)
```

user-band.py

```
# define band as the argument allowed to be passed
# through the scriptargs flag
from argparse import ArgumentParser
parser=ArgumentParser()
parser.add_argument('-b', '--band', action='store', dest='bnd', type=str, help='Band Number')
args = parser.parse_args()

# define directories to capture image
homePath = expandpath('~')
dataPath = os.path.join(homePath, 'Data')
scriptingPath = os.path.join(dataPath, 'Scripting')
imagePath = os.path.join(scriptingPath, 'Images')
fileName = os.path.join(imagePath, 'scriptargs.jpg')

# dictionary for loadADDEImage
addeParms = dict(
    server = 'adde.ucar.edu',
      dataset = 'EAST',
      descriptor = 'CONUS',
      size = 'ALL',
      position = -1,
      )

# load data and include the user-specified band
data = loadADDEImage(band=args.bnd, **addeParms)

# build window and display data
panel = buildWindow(800, 600)
layer = panel[0].createLayer('Image Display',data)
layer.setLayerLabel('GOES-16 band %s - %s' % (str(args.bnd), data['nominal-time']), size=20)

# capture image
fileName = ('%s/band_%s.jpg' % (imagePath, str(args.bnd)))
panel[0].captureImage(fileName)
```