UCRL-SM-209589

VisIt Python Interface Manual

February 2005

Version 1.4.1

Lawrence
Livermore
National
Laboratory

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

Chapter 1 Introduction to VisIt

1.0 Overview

VisIt is a distributed, parallel, visualization tool for visualizing data defined on two and three-dimensional structured and unstructured meshes. VisIt's distributed architecture allows it to leverage both the compute power of a large parallel computer and the graphics acceleration hardware of a local workstation. Another benefit of the distributed architecture is that VisIt can visualize the data where it is generated, eliminating the need to move data. VisIt can be controlled by a Graphical User Interface (GUI) or through the Python scripting language. More information about VisIt's Graphical User Interface can be found in the *VisIt User's Manual*.

2.0 Manual chapters

This manual is broken down into the following chapters:

Chapter title	Chapter description
Introduction to VisIt	This chapter.
Python	Describes the basic features of the Python programming language.
Quick Recipes	Describes common patterns for scripting using the VisIt Python Interface.
Function Reference	Describes functions in the VisIt Python Interface.
Object Reference I	Describes extension types.

Plot Plugin Object Reference	Describes plot plugin extension types
Operator Plugin Object Reference	Describes operator plugin extension types.
Appendix A	Describes VisIt's command line options.

3.0 Understanding how VisIt works

VisIt visualizes data by creating one or more plots in a visualization window, also known as a vis window. Examples of plots include Mesh plots, Contour plots and Pseudocolor plots. Plots take as input one or more mesh, material, scalar, or tensor variables. It is possible to modify the variables by applying one or more operators to the variables before passing them to a plot. Examples of operators include arithmetic operations or taking slices through the mesh. It is also possible to restrict the visualization of the data to subsets of the mesh.

VisIt provides Python bindings to all of its plots and operators so they may be controlled through scripting. Each plot or operator plugin provides a function, which is added to the VisIt namespace, to create the right type of plot or operator attributes. The attribute object can then be modified by setting its fields and then it can be passed to a general-purpose function to set the plot or operator attributes. To display a complete list of functions in the VisIt Python Interface, you can type dir() at the Python prompt. Similarly, to inspect the contents of any object, you can type its name at the Python prompt.

VisIt supports up to 16 visualization windows, also called vis windows. Each vis window is independent of the other vis windows and VisIt Python functions generally apply only to the currently active vis window.

This manual explains how to use the VisIt Python Interface which is a Python extension module that controls VisIt's viewer. In that way, the VisIt Python Interface fulfills the same role as VisIt's GUI. The difference is that the viewer is totally controlled through Python scripting, which makes it easy to write scripts to create visualizations and even movies. Since the VisIt module controls VisIt's viewer, the Python interpreter currently has no direct mechanism for passing data to the compute engine (see Figure 1-1). If you want to write a script that generates simulation data and have that script pass data to the compute engine, you must pass the data through a file on disk.

The VisIt Python Interface comes packaged in two varieties: the extension module and the Command Line Interface (CLI). The extension module version of the VisIt Python Interface is imported into a standard Python 2.1.2 interpreter using the import directive. VisIt's command line interface (CLI) is essentially a Python interpreter where the VisIt

Python Interface is built-in. The CLI is provided to simplify the process of running VisIt Python scripts.

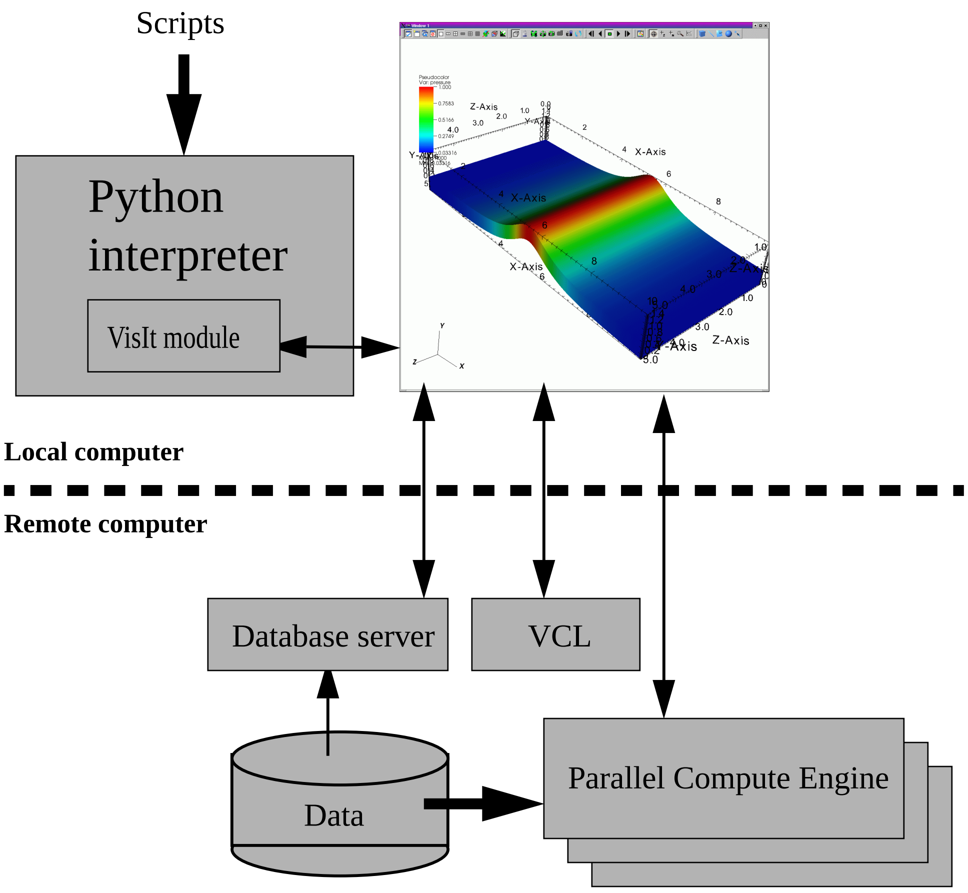

Figure 1-1: VisIt's architecture

4.0 Starting VisIt

You can invoke VisIt's command line interface from the command line by typing: **visit -cli**

For a complete list of the command line options, see **Appendix A** on page 259. It is best to have VisIt in your default search path instead of specifying the absolute path to VisIt when starting it. This isn't important when VisIt is run locally, but VisIt may not run properly in a distributed manner if it isn't in your default search path on all the machines on which you plan to run VisIt.

5.0 Getting started

VisIt is a tool for visualizing 2D and 3D scientific databases. The first thing to do when running VisIt is select databases to visualize. To select a database, you must first open the database using the OpenDatabase function. After a window has an open database, any number of plots and operators can be added. To create a plot, use the AddPlot function. After adding a plot, call the DrawPlots function to make sure that all of the new plots are drawn.

Example:

```
OpenDatabase(“/usr/gapps/visit/data/multi_curv3d.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()
```

To see a list of the available plots and operators when you use the VisIt Python Interface, use the OperatorPlugins and PlotPlugins functions. Each of those functions returns a tuple of strings that contain the names of the currently loaded plot or operator plugins. Each plot and operator plugin provides a function for creating an attributes object to set the plot or operator attributes. The name of the function is the name of the plugin in the tuple returned by the OperatorPlugins or PlotPlugins functions plus the word “Attributes”. For example, the “Pseudocolor” plot provides a function called PseudocolorAttributes.

To set the plot attributes or the operator attributes, first use the attributes creation function to create an attributes object. Assign the newly created object to a variable name and set the fields in the object. Each object has its own set of fields. To see the available fields in an object, print the name of the variable at the Python prompt and press the Enter key. This will print the contents of the object so you can see the fields contained by the object. After setting the appropriate fields, pass the object to either the SetPlotOptions function or the SetOperatorAttributes function.

Example:

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
AddOperator(“Slice”)
p = PseudocolorAttributes()
p.colorTableName = “rainbow”
p.opacity = 0.5
SetPlotOptions(p)
a = SliceAttributes()
a.originType = a.Point
a.normal, a.upAxis = (1,1,1), (-1,1,-1)
SetOperatorOptions(a)
DrawPlots()
```

That’s all there is to creating a plot using VisIt’s Python Interface. For more information on creating plots and performing specific actions in VisIt, refer to the documentation for each function later in this manual.

Chapter 2 Python

1.0 Overview

Python is a general purpose, interpreted, extensible, object-oriented scripting language that was chosen for VisIt's scripting language due to its ease of use and flexibility. VisIt's Python interface was implemented as Python module and it allows you to enhance your Python scripts with coding to control VisIt. This chapter explains some of Python's syntax so it will be more familiar when you examine the examples found in this document. For more information on programming in Python, there are a number of good references, including on the Internet at *http://www.python.org*.

2.0 Indentation

One of the most obvious features of Python is its use of indentation for new scopes. You must take special care to indent all program logic consistently or else the Python interpreter may halt with an error, or worse, not do what you intended. You must increase indentation levels when you define a function, use an if/elif/else statement, or use any loop construct.

Note the different levels of indentation:

```
def example_function(n):
   v = 0
   if n > 2:
      print “n greater than 2.”
   else:
      v = n * n
   return v
```

3.0 Comments

Like all good programming languages, Python supports the addition of comments in the code. Comments begin with a pound character (#) and continue to the end of the line.

```
# This is a comment
a = 5 * 5
```

4.0 Identifiers

The Python interpreter accepts any identifier that contains letters ‘A’-’Z’, ‘a’-’z’ and numbers ‘0’’9’ as long as the identifier does not begin with a number. The Python interpreter is case-sensitive so the identifier “case” would not be the same identifier as “CASE”. Be sure to case consistently throughout your Python code since the Python interpreter will instantiate any identifier that it has not seen before and mixing case would cause the interpreter to use multiple identifiers and cause problems that you might not expect. Identifiers can be used to refer to any type of object since Python is flexible in its treatment of types.

5.0 Data types

Python supports a wide variety of data types and allows you to define your own data types readily. Most types are created from a handful of building-block types such as integers, floats, strings, tuples, lists, and dictionaries.

5.1 Strings

Python has built-in support for strings and you can create them using single quotes or double quotes. You can even use both types of quotes so you can create strings that include quotes in case quotes are desired in the output. Strings are sequence objects and support operations that can break them down into characters.

```
s = ‘using single quotes’
s2 = “using double quotes”
s3 = ‘nesting the “spiffy” double quotes’
```

5.2 Tuples

Python supports tuples, which can be thought of as a read-only set of objects. The members of a tuple can be of different types. Tuples are commonly used to group multiple related items into a single object that can be passed around more easily. Tuples support a number of operations. You can subscript a tuple like an array to access its individual members. You can easily determine whether an object is a member of a tuple. You can iterate over a tuple. There are many more uses for tuples. You can create tuples by enclosing a comma-separated list of objects in parenthesis.

```
# Create a tuple
a = (1,2,3,4,5)
print “The first value in a is:”, a[0]
# See if 3 is in a using the “in” operator.
print “3 is in a:”, 3 in a
# Create another tuple and add it to the first one to create c.
b = (6,7,8,9)
c = a + b
# Iterate over the items in the tuple
for value in c:
   print “value is: “, value
```

5.3 Lists

Lists are just like tuples except they are not read-only and they use square brackets [] to enclose the items in the list instead of using parenthesis.

```
# Start with an empty list.
L = []
for i in range(10):
   # Add i to the list L
   L = L + [i]
print L
# Assign a value into element 6
L[5] = 1000
print L
```

5.4 Dictionaries

Dictionaries are Python containers that allow you to store a value that is associated with a key. Dictionaries are convenient for mapping 1 set to another set since they allow you to perform easy lookups of values. Dictionaries are declared using curly braces {} and each item in the dictionary consists of a key:value pair with the key and values being separated by a colon. To perform a lookup using a dictionary, provide the key whose value you want to look up to the subscript [] operator.

```
colors = {“red” : “rouge”, “orange” : “orange”, \
“yellow” : “jaune”, “green” : “vert”, “blue” : “bleu”}
# Perform lookups using the keys.
for c in colors.keys():
   print “%s in French is: %s“ % (c, colors[c])
```

6.0 Control flow

Python, like other general-purpose programming languages provides keywords that implement control flow. Control flow is an important feature to have in a programming language because it allows complex behavior to be created using a minimum amount of scripting.

6.1 if/elif/else

Python provides if/elif/else for conditional branching. The if statement takes any expression that evaluates to an integer and it takes the if branch if the integer value is 1 other wise it takes the else branch if it is present.

```
# Example 1
if condition:
   do_something()

# Example 2
if condition:
   do_something()
else:
   do_something_else()

# Example 3
if condition:
   do_domething()
elif conditionn:
   do_something_n()
...
else:
   do_something_else()
```

6.2 For loop

Python provides a for loop that allows you to iterate over all items stored in a sequence object (tuples, lists, strings). The body of the for loop executes once for each item in the sequence object and allows you to specify the name of an identifier to use in order to reference the current item.

```
# Iterating through the characters of a string
for c in “characters”:
   print c

# Iterating through a tuple
for value in (“VisIt”, “is”, “coolness”, “times”, 100):
   print value

# Iterating through a list
for value in [“VisIt”, “is”, “coolness”, “times”, 100):
   print value

# Iterating through a range of numbers [0,N) created with range(N).
N = 100
for i in range(N):
   print i, i*i
```

6.3 While loop

Python provides a while loop that allows you to execute a loop body indefinitely based on some condition. The while loop can be used for iteration but can also be used to execute more complex types of loops.

```
token = get_next_token()
while token != "":
   do_something(token)
   token = get_next_token()
```

7.0 Functions

Python comes with many built-in functions and modules that implement additional functions. Functions can be used to execute bodies of code that are meant to be re-used. Functions can optionally take arguments and can optionally return values. Python provides the def keyword, which allows you to define a function. The def keyword is followed by the name of the function and its arguments, which should appear as a tuple next to the name of the function.

```
# Define a function with no arguments and no return value.
def my_function():
   print "my function prints this..."

# Define a function with arguments and a return value.
def n_to_the_d_power(n, d):
   value = 1
   if d > 0:
      for i in range(d):
         value = value * n
   elif d < 0:
      value = 1. / float(n_to_the_d_power(n, -d))

   return value
```

Chapter 3 Quick Recipes

1.0 Overview

This manual contains documentation for over two hundred functions and several dozen extension object types. Learning to combine the right functions in order to accomplish a visualization task without guidance would involve hours of trial and error. To maximize productivity and start creating visualizations using Visit's Python Interface as fast as possible, this chapter provides some common patterns, or "quick recipes" that you can combine to quickly create complex scripts.

2.0 Patterns

This section contains short various code snippets that illustrate various concepts. Each bit of code is targetted at performing a specific visualization action and contains the VisIt scripting commands that are required to complete that action. You can combine one or more of these sections of code to build up more complex scripts.

2.1 Using session files

VisIt's session files contain all of the information required to recreate plots that have been set up in previous interactive VisIt sessions. Since session files contain all of the information about plots, etc., they are natural candidates to make scripting easier since they can be used to do the hard part of setting up the complex visualization, leaving the bulk of the script to animate through time or alter the plots in some way. To use session files within a script, use the RestoreSession function, documented on page 144.

```
# Import a session file from the current working directory.
RestoreSesssion("my_visualization.session", 0)
# Now that VisIt has restored the session, animate through time.
```

```
for states in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
   SaveWindow()
```

2.2 Getting something on the screen

If you don’t want to use a session file to begin the setup for your visualization then you will have to dive into opening databases, creating plots, and animating through time. This is where all of hand-crafted scripts begin. The first step in creating a visualization is opening a database. VisIt provides the OpenDatabase function (see page 118) to open a database. Once a database has been opened, you can create plots from its variables using the AddPlot function (see page 34). The AddPlot function takes a plot plugin name and the name of a variable from the open database. Once you’ve added a plot, it is in the new state, which means that it has not yet been submitted to the compute engine for processing. To make sure that the plot gets drawn, call the DrawPlots function.

```
# Step 1: Open a database
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)

# Step 2: Add plots
AddPlot(“Pseudocolor”, “pressure”)
AddPlot(“Mesh”, “quadmesh”)

# Step 3: Draw the plots
DrawPlots()

# Step 4: Animate through time and save images
for states in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
   SaveWindow()
```

2.3 Saving images

Much of the time, the entire purpose of using VisIt’s Python Interface is to create a script that can save out images of a time-varying database for the purpose of making movies. Saving images using VisIt’s Python Interface is a straight-forward process, involving just a few functions.

2.3.1 Setting the output image characteristics

VisIt provides a number of options for saving files, including: file format, filename, image size, to name a few. These attributes are grouped into the SaveWindowAttributes object. To set the options that VisIt uses to save files, you must create a SaveWindowAttributes object, change the necessary attributes, and call the SetSaveWindowAttributes function.

Note that if you want to create images using a specific image resolution, the best way is to use the *-geometry* command line argument with VisIt’s Command Line Interface and tell VisIt to use screen capture. If you instead require your script to be capable of saving several different image sizes then you can turn off screen capture and set the image resolution in the SaveWindowAttributes object.

```
# Save a BMP file at 1024x768 resolution
s = SaveWindowAttributes()
s.format = s.BMP
s.fileName = “mybmpfile”
s.width, s.height = 1024,768
s.screenCapture = 0
SetSaveWindowAttributes(s)
```

2.3.2 Saving an image

Once you have set the SaveWindowAttributes to your liking, you can call the SaveWindow function to save an image. The SaveWindow function returns the name of the image that is saved so you can use that for other purposes in your script.

```
# Save images of all time steps and add each image filename to a list.
names = []
for state in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
   # Save the image
   n = SaveWindow()
   names = names + [n]
print names
```

2.4 Working with databases

VisIt allows you to open a wide array of databases both in terms of supported file formats and in terms how databases treat time. Databases can have a single time state or can have multiple time states. Databases can natively support multiple time states or sets of single time states files can be grouped into time-varying databases using .visit files or using virtual databases. Working with databases gets even trickier if you are using VisIt to visualize a database that is still being generated by a simulation. This section describes how to interact with databases.

2.4.1 Opening a database

Opening a database is a relatively simple operation - most complexities arise in how the database treats time. If you only want to visualize a single time state or if your database format natively supports multiple time states per file then opening a database requires just a single call to the OpenDatabase function.

```
# Open a database at time state 0
OpenDatabase(“/usr/gapps/visit/data/allinone00.pdb”)
```

2.4.2 Opening a database at late time

Opening a database at a later time state is done just the same as opening a database at time state zero except that you must specify the time state at which you want to open the database. There are a number of reasons for opening a database at a later time state. The most common reason for doing so, as opposed to just changing time states later, is that VisIt uses the metadata from the first opened time state to describe the contents of the database for all time states (except for certain file formats that don't do this i.e. SAMRAI). This means that the list of variables found for the first time state that you open is used for all time states. If your database contains a variable at a later time state that does not exist at earlier time states, you must open the database at a later time state to gain access to the transient variable.

```
# Open a database at a later time state to pick up transient variables
OpenDatabase("/usr/gapps/visit/data/wave.visit", 17)
```

2.4.3 Opening a virtual database

VisIt provides two ways for accessing a set of single time-state files as a single time-varying database. The first method is a .visit file, which is a simple text file that contains the names of each file to be used as a time state in the time-varying database. The second method uses "virtual databases", which allow VisIt to exploit the file naming conventions that are often employed by simulation codes when they create their dumps. In many cases, VisIt can scan a specified directory and determine which filenames look related. Filenames with close matches are grouped as individual time states into a virtual database whose name is based on the more abstract pattern used to create the filenames.

```
# Opening first file in series wave0000.silo, wave0010.silo, ...
OpenDatabase("/usr/gapps/visit/data/wave0000.silo")

# Opening a virtual database representing all wave*.silo files.
OpenDatabase("/usr/gapps/visit/data/wave*.silo database")
```

2.4.4 Opening a remote database

VisIt supports running the client on a local computer while also allowing you to process data in parallel on a remote computer. If you want to access databases on a remote computer using VisIt's Python Interface, the only difference to accessing a database on a local computer is that you must specify a host name as part of the database name.

```
# Opening a file on a remote computer by giving a host name
OpenDatabase("thunder:/usr/gapps/visit/data/wave.visit", 17)
```

2.4.5 Opening a compute engine

Sometimes it is advantageous to open a compute engine before opening a database. When you tell VisIt to open a database using the OpenDatabase function, VisIt also launches a compute engine and tells the compute engine to open the specified database. When the VisIt Python Interface is run with a visible window, the **Engine Chooser Window** will present itself so you can select a host profile. If you want to design a script that must specify parallel options, etc in batch mode where there is no **Engine Chooser Window** then you have few options other than to open a compute engine before opening a database. To open a compute engine, use the OpenComputeEngine function. You can pass the name of the host on which to run the compute engine and any arguments that must be used to launch the engine such as the number of processors.

```
# Open a remote, parallel compute engine before opening a database
OpenComputeEngine(“mcr”, (“-np”, “4”, “-nn”, “2”))
OpenDatabase(“mcr:/usr/gapps/visit/data/multi_ucd3d.silo”)
```

2.5 Working with plots

Plots are viewable objects, created from a database, that can be displayed in a visualization window. VisIt provides several types of plots and each plot allows you to view data using different visualization techniques. For example, the Pseudocolor plot allows you to see the general shape of a simulated object while painting colors on it according to the values stored in a variable’s scalar field. The most important functions for interacting with plots are covered in this section.

2.5.1 Creating a plot

The function for adding a plot in VisIt is: AddPlot. The AddPlot function takes the name of a plot type and the name of a variable that is to be plotted and creates a new plot and adds it to the plot list. The name of the plot to be created corresponds to the name of one of VisIt’s plot plugins, which can be queried using the PlotPlugins function. The variable that you pass to the AddPlot function must be a valid variable for the open database. New plots are not realized, meaning that they have not been submitted to the compute engine for processing. If you want to force VisIt to process the new plot you must call the DrawPlots function.

```
# Names of all available plot plugins
print PlotPlugins()
# Create plots
AddPlot(“Pseudocolor”, “pressure”)
AddPlot(“Mesh”, “quadmesh”)
# Draw the plots
DrawPlots()
```

2.5.2 Plotting materials

Plotting materials is a common operation in VisIt. The Boundary and FilledBoundary plots enable you to plot material boundaries and materials, respectively.

```
# Plot material boundaries
AddPlot(“Boundary”, “mat1”)
# Plot materials
AddPlot(“FilledBoundary”, “mat1”)
```

2.5.3 Setting plot attributes

Each plot type has an attributes object that controls how the plot generates its data or how it looks in the visualization window. The attributes object for each plot contains different fields. You can view the individual object fields by printing the object to the console. Each plot type provides a function that creates a new instance of one of its attribute objects. The function name is always of the form: plotname + “Attributes”. For example, the attributes object creation function for the Pseudocolor plot would be: PseudocolorAttributes. To change the attributes for a plot, you create an attributes object using the appropriate function, set the properties in the returned object, and tell VisIt to use the new plot attributes by passing the object to the SetPlotOptions function (see page 166). Note that you should set a plot’s attributes before calling the DrawPlots method to realize the plot since setting a plot’s attributes can cause the compute engine to recalculate the plot.

```
# Creating a Pseudocolor plot and setting min/max values.
AddPlot(“Pseudocolor”, “pressure”)
p = PseudocolorAttributes()
# Look in the object
print p
# Set the min/max values
p.min, p.minFlag = 1, 0.0
p.max, p.maxFlag = 1, 10.0
SetPlotOptions(p)
```

2.5.4 Working with multiple plots

When you work with more than one plot, it is sometimes necessary to set the active plots because some of VisIt’s functions apply to all of the active plots. The active plot is usually the last plot that was created unless you’ve changed the list of active plots. Changing which plots are active is useful when you want to delete or hide certain plots or set their plot attributes independently. When you want to set which plots are active, use the SetActivePlots function. If you want to list the plots that you’ve created, call the ListPlots function.

```
# Create more than 1 plot of the same type
AddPlot(“Pseudocolor”, “pressure”)
AddPlot(“Pseudocolor”, “density”)

# List the plots. The second plot should be active.
ListPlots()

# Draw the plots
DrawPlots()

# Hide the first plot
SetActivePlots(0)
HideActivePlots()

# Set both plots’ color table to “hot”
p = PseudocolorAttributes()
p.colorTableName = “hot”
SetActivePlots((0,1))
SetPlotOptions(p)

# Show the first plot again.
SetActivePlots(0)
HideActivePlots()
```

```
# Delete the second plot
SetActivePlots(1)
DeleteActivePlots()
ListPlots()
```

2.5.5 Plots in the error state

When VisIt’s compute engine cannot process a plot, the plot is put into the error state. Once a plot is in the error state, it no longer is displayed in the visualization window. If you are generating a movie, plots entering the error state can be a serious problem because you most often want all of the plots that you have created to animate through time and not disappear in the middle of the animation. You can add extra code to your script to prevent plots from disappearing (most of the time) due to error conditions by adding a call to the DrawPlots function.

```
# Save an image and take care of plots that entered the error state.
drawThePlots = 0
for state in range(TimeSliderGetNStates()):
   if SetTimeSliderState(state) == 0:
      drawThePlots = 1
   if drawThePlots == 1:
      if DrawPlots() == 0:
         print “VisIt could not draw plots for state: %d” % state
      else:
         drawThePlots = 0
   SaveWindow()
```

2.6 Operators

Operators are filters that are applied to database variables before the compute engine uses them to create plots. Operators can be linked one after the other to form chains of operators that can drastically transform the data before plotting it.

2.6.1 Adding operators

Adding an operator is similar to adding a plot in that you call a function with the name of the operator to be added. The list of available operators is returned by the OperatorPlugins function. Any of the names returned in that plugin can be used to add an operator using the AddOperator function. Operators are added to the active plots by default but you can also force VisIt to add them to all plots in the plot list.

```
# Print available operators
print OperatorPlugins()

# Create a plot
AddPlot(“Pseudocolor”)

# Add an Isovolume operator and a Slice operator
AddOperator(“Isovolume”)
AddOperator("Slice")
DrawPlots()
```

2.6.2 Setting operator attributes

Each plot gets its own instance of an operator which means that you can set each plot's operator attributes independently. Like plots, operators use objects to set their attributes. These objects are returned by functions whose names are of the form: operatorname + "Attributes". Once you have created an operator attributes object, you can pass it to the SetOperatorOptions to set the options for an operator. Note that setting the attributes for an operator nearly always causes the compute engine to recalculate the operator. You can use the power of VisIt's Python Interface to create complex operator behavior such as in the following code example, which moves slice planes through a Pseudocolor plot.

```
# Moving slice planes
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
AddOperator("Slice")
s = SliceAttributes()
s.originType = s.Percent
s.project2d = 0
SetOperatorOptions(s)
DrawPlots()

nSteps = 20
for axis in (0,1,2):
   s.axisType = axis
   for step in range(nSteps):
      t = float(step) / float(nSteps - 1)
      s.originPercent = t * 100.
      SetOperatorOptions(s)
      SaveWindow()
```

2.7 Quantitative operations

This section focuses on some of the operations that allow you to examine your data more quantitatively.

2.7.1 Defining expressions

VisIt allows you to create derived variables using its powerful expressions language. You can plot or query variables created using expressions just as you would if they were read from a database. VisIt's Python Interface allows you to create new scalar, vector, tensor variables using the DefineScalarExpression, DefineVectorExpression, and DefineTensorExpression functions.

```
# Creating a new expression
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
DrawPlots()
```

```
DefineScalarExpression(“newvar”, “sin(hardyglobal) + \
cos(shepardglobal”)
ChangeActivePlotsVar(“newvar”)
```

2.7.2 Pick

VisIt allows you to pick on cells, nodes, and points within a database and reutrn information for the item of interest. To that end, VisIt provides several pick functions. Once a pick function has been called, you can call the GetPickOutput function to get a string that contains the pick information. The information in the string could be used for a multitude of uses such as building a test suite for a simulation code.

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hgslice”)
DrawPlots()
s = []
# Pick by a node id
PickbyNode(300)
s = s + [GetPickOutput()]
# Pick by a cell id
PickByZone(250)
s = s + [GetPickOutput()]
# Pick on a cell using a 3d point
Pick((-2., 2., 0.))
s = s + [GetPickOutput()]
# Pick on the node closest to (-2,2,0)
NodePick((-2,2,0))
s = s + [GetPickOutput()]
# Print all pick results
print s
```

2.7.3 Lineout

VisIt allows you to extract data along a line, called a lineout, and plot the data using a Curve plot.

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hgslice”)
DrawPlots()
Lineout((-5,-3), (5,8))
# Specify a number of sample points
Lineout((-5,-4), (5,7))
```

2.7.4 Query

VisIt can perform a number of different queries based on values calculated about plots or their originating database.

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hardyglobal”)
DrawPlots()
Query(“NumNodes)
print “The float value is: %g” % GetQueryOutputValue()
```

```
Query(“NumNodes”)
```

2.7.5 Finding the min and the max

A common operation in debugging a simulation code is examining the min and max values. Here is a pattern that allows you to print out the min and the max values and their locations in the database and also see them visually.

```
# Define a helper function to get the id’s of the MinMax query.
def GetMinMaxIds():
   Query(“MinMax”)
   import string
   s = string.split(GetQueryOutputString(), “ “)
   retval = []
   nextGood = 0
   idType = 0
   for token in s:
      if token == “(zone” or token == “(cell”:
         idType = 1
         nextGood = 1
         continue
      elif token == “(node”:
         idType = 0
         nextGood = 1
         continue
      if nextGood == 1:
         nextGood = 0
         retval = retval + [(idType, int(token))]
   return retval

# Set up a plot
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hgslice”)
DrawPlots()

# Do picks on the ids that were returned by MinMax.
for ids in GetMinMaxIds():
   idType = ids[0]
   id = ids[1]
   if idType == 0:
      PickByNode(id)
   else:
      PickByZone(id)
```

2.8 Subsetting

VisIt allows the user to turn off subsets of the visualization using a number of different methods. Databases can be divided up any number of ways: domains, materials, etc. This section provides some details on how to remove materials and domains from your visualization.

2.8.1 Turning off domains

VisIt’s Python Interface provides the TurnDomainsOn and TurnDomainsOff functions to make it easy to turn domains on and off.

```
OpenDatabase(“/usr/gapps/visit/data/multi_rect2d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()
# Turning off all but the last domain
d = GetDomains()
for dom in d[:-1]:
   TurnDomainsOff(dom)
# Turn all domains off
TurnDomainsOff()
# Turn on domains 3,5,7
TurnDomainsOn((d[3], d[5], d[7]))
```

2.8.2 Turning off materials

VisIt’s Python Interface provides the TurnMaterialsOn and TurnMaterialsOff functions to make it easy to turn materials on and off.

```
OpenDatabase(“/usr/gapps/visit/data/multi_rect2d.silo”)
AddPlot(“FilledBoundary”, “mat1”)
DrawPlots()
# Print the materials are:
GetMaterials()
# Turn off material 2
TurnMaterialsOff(“2”)
```

2.9 View

Setting up the view in your Python script is one of the most important things you can do to ensure the quality of your visualization because the view concentrates attention on an object or inferest. VisIt provides different methods for setting the view, depending on the dimensionality of the plots in the visualization window but despite differences in how the view is set, the general procedure is basically the same.

2.9.1 Setting the 2D view

The 2D view consists of a rectangular window in 2D space and a 2D viewport in the visualization window. The window in 2D space determines what parts of the visualization you will look at while the viewport determines where the images will appear in the visualization window. It is not necessary to change the viewport most of the time.

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hgslice”)
AddPlot(“Mesh”, “Mesh2D”)
AddPlot(“Label”, “hgslice”)
DrawPlots()
print “The current view is:”, GetView2D()
```

```
# Get an initialized 2D view object.
v = GetView2D()
v.windowCoords = (-7.67964, -3.21856, 2.66766, 7.87724)
SetView2D(v)
```

2.9.2 Setting the 3D view

The 3D view is much more complex than the 2D view. For information on the actual meaning of the fields in the View3DAttributes object, refer to page 210 or the *VisIt User's Manual*. VisIt automatically computes a suitable view for 3D objects and it is best to initialize new View3DAttributes objects using the GetView3D function so most of the fields will already be initialized. The best way to get new views to use in a script is to interactively create the plot and repeatedly call GetView3D() after you finish rotating the plots with the mouse. You can paste the printed view information into your script and modify it slightly to create sophisticated view transitions.

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hardyglobal”)
AddPlot(“Mesh”, “Mesh”)
DrawPlots()
v = GetView3D()
print “The view is: “, v
v.viewNormal = (-0.571619, 0.405393, 0.713378)
v.viewUp = (0.308049, 0.911853, -0.271346)
SetView3D(v)
```

2.9.3 Flying around plots

Flying around plots is a commonly requested feature when making movies. Fortunately, this is easy to script. The basic method used for flying around plots is interpolating the view. VisIt provides a number of functions that can interpolate View2DAttributes and View3DAttributes objects. The most useful of these functions is the EvalCubicSpline function. The EvalCubicSpline function uses piece-wise cubic polynomials to smoothly interpolate between a tuple of N like items. Scripting smooth view changes using EvalCubicSpline is rather like keyframing in that you have a set of views that are mapped to some distance along the parameterized space [0., 1.]. When the parameterized space is sampled with some number of samples, VisIt calculates the view for the specified parameter value and returns a smoothly interpolated view. One benefit over keyframing, in this case, is that you can use cubic interpolation whereas VisIt's keyframing mode currently uses linear interpolation.

```
# Do a pseudocolor plot of u.
OpenDatabase(’/usr/gapps/visit/data/globe.silo’)
AddPlot(’Pseudocolor’, ’u’)
DrawPlots()

# Create the control points for the views.
c0 = View3DAttributes()
c0.viewNormal = (0, 0, 1)
c0.focus = (0, 0, 0)
c0.viewUp = (0, 1, 0)
```

```
c0.viewAngle = 30
c0.parallelScale = 17.3205
c0.nearPlane = 17.3205
c0.farPlane = 81.9615
c0.perspective = 1

c1 = View3DAttributes()
c1.viewNormal = (-0.499159, 0.475135, 0.724629)
c1.focus = (0, 0, 0)
c1.viewUp = (0.196284, 0.876524, -0.439521)
c1.viewAngle = 30
c1.parallelScale = 14.0932
c1.nearPlane = 15.276
c1.farPlane = 69.917
c1.perspective = 1

c2 = View3DAttributes()
c2.viewNormal = (-0.522881, 0.831168, -0.189092)
c2.focus = (0, 0, 0)
c2.viewUp = (0.783763, 0.556011, 0.27671)
c2.viewAngle = 30
c2.parallelScale = 11.3107
c2.nearPlane = 14.8914
c2.farPlane = 59.5324
c2.perspective = 1

c3 = View3DAttributes()
c3.viewNormal = (-0.438771, 0.523661, -0.730246)
c3.focus = (0, 0, 0)
c3.viewUp = (-0.0199911, 0.80676, 0.590541)
c3.viewAngle = 30
c3.parallelScale = 8.28257
c3.nearPlane = 3.5905
c3.farPlane = 48.2315
c3.perspective = 1

c4 = View3DAttributes()
c4.viewNormal = (0.286142, -0.342802, -0.894768)
c4.focus = (0, 0, 0)
c4.viewUp = (-0.0382056, 0.928989, -0.36813)
c4.viewAngle = 30
c4.parallelScale = 10.4152
c4.nearPlane = 1.5495
c4.farPlane = 56.1905
c4.perspective = 1

c5 = View3DAttributes()
c5.viewNormal = (0.974296, -0.223599, -0.0274086)
c5.focus = (0, 0, 0)
c5.viewUp = (0.222245, 0.97394, -0.0452541)
c5.viewAngle = 30
c5.parallelScale = 1.1052
c5.nearPlane = 24.1248
c5.farPlane = 58.7658
```

```
c5.perspective = 1

c6 = c0

# Create a tuple of camera values and x values. The x values
# determine where in [0,1] the control points occur.
cpts = (c0, c1, c2, c3, c4, c5, c6)
x=[]
for i in range(7):
   x = x + [float(i) / float(6.)]

# Animate the view using EvalCubicSpline.
nsteps = 100
for i in range(nsteps):
   t = float(i) / float(nsteps - 1)
   c = EvalCubicSpline(t, x, cpts)
   c.nearPlane = -34.461
   c.farPlane = 34.461
   SetView3D(c)
```

Chapter 4 Function Reference

3.0 Overview

This chapter describes all of the non-plugin functions that are built into the VisIt Python interface. Functions that are added by plugins and functions that exist only to create extension objects are covered in the later object references; not this chapter.

There are many functions in the VisIt Python Interface because VisIt provides a lot of capabilities for visualizing data. Each function is described in detail so you will know what each function does and also what arguments to pass. Most functions also include a brief Python script that demonstrates how to use the function in context.

VisIt's Python Interface is implemented as a Python extension module, which is a library that can be loaded by the Python interpreter at runtime. All of the functions in this chapter are part of the "visit" module namespace since modules all have their own namespace. If you use the Python "import" directive to import the VisIt module into a Python interpreter, you must preface each call to a VisIt function with the module name. For example, if you wanted to call the OpenDatabase function, you would type: visit.OpenDatabase("mydatabase"). If you use the VisIt Command Line Interface (CLI), which can be invoked by typing *visit -cli* on the command line, all of the functions are added to the global namespace and prefacing functions with the name of the visit module is not necessary.

The first functions descibed in this chapter are most directly related to importing the VisIt module into a Python interpreter. The functions allow you to specify the debug level, add arguments, launch, and close the module's connection to VisIt's viewer. If you intend to always use VisIt functions from within VisIt's CLI, then you can skip to page 32 to see the first function that is not associated with setting properties for the VisIt module.

`AddArgument`—Add an argument to the viewer’s command line argument list.

Synopsis:

```
AddArgument(argument)
```

Arguments:

`argument`	A string object that is added to the viewer’s command line argument list.

Returns:

AddArgument does not return a value.

Description:

The AddArgument function is used to add extra command line arguments to VisIt’s viewer. This is only useful when VisIt’s Python interface is imported into a stand-alone Python interpreter because the AddArgument function must be called before the viewer is launched. The AddArgument function has no effect when used in VisIt’s cli program because the viewer is automatically launched before any commands are processed.

Example:

```
import visit
visit.AddArgument(“-nowin”) # Add the -nowin argument to the viewer.
```

`Close`—Closes the viewer.

Synopsis:

```
Close()
```

Arguments:

```
none
```

Returns:

The Close function does not return a value.

Description:

The Close function terminates VisIt’s viewer. This is useful for Python scripts that only need access to VisIt’s capabilties for a short time before closing VisIt.

Example:

```
import visit
visit.Launch()
visit.Close() # Close the viewer
```

Launch—Launches VisIt’s viewer

Synopsis:

```
Launch()
LaunchNowin()
```

Arguments:

```
none
```

Returns:

The Launch functions do not return a value

Description:

The Launch function is used to launch VisIt’s viewer when the VisIt module is imported into a stand-alone Python interpreter. The Launch function has no effect when a viewer already exists. The difference between Launch and LaunchNowin is that LaunchNowin prevents the viewer from ever creating onscreen visualization windows. The LaunchNowin function is primarily used in Python scripts that want to generate visualizations using VisIt without the use of a display such as when generating movies.

Example 1:

```
import visit
visit.AddArgument(“-geometry”)
visit.AddArgument(“1024x1024”)
visit.LaunchNowin()
```

Example 2:

```
import visit
visit.AddArgument(“-nowin”)
visit.Launch()
```

`LocalNamespace`—Tells the VisIt module to import plugins into the global namespace.

Synopsis:

```
LocalNamespace()
```

Arguments:

```
none
```

Returns:

The LocalNamespace function does not return a value.

Description:

The LocalNamespace function tells the VisIt module to add plugin functions to the global namespace when the VisIt module is imported into a stand-alone Python interpreter. This is the default behavior when using VisIt’s cli program.

Example:

```
import visit
visit.LocalNamespace()
visit.Launch()
```

`DebugLevel`—Set or Get the VisIt module’s debug level.

Synopsis:

```
GetDebugLevel() -> integer
SetDebugLevel(level)
```

Arguments:

`level`	An integer in the range [1,5]. A value of 1 is a low debug level, which should be used to produce little output while a value of 5 should produce a lot of debug output.

Returns:

The GetDebugLevel function returns the debug level of the VisIt module.

Description:

The GetDebugLevel and SetDebugLevel functions are used when debugging VisIt Python scripts. The SetDebugLevel function sets the debug level for VisIt’s viewer thus it must be called before a Launch method. The debug level determines how much detail is written to VisIt’s execution logs when it executes. The GetDebugLevel function can be used in Python scripts to alter the behavior of the script. For instance, the debug level can be used to selectively print values to the console.

Example:

% visit -cli -debug 2

```
print “VisIt’s debug level is: %d” % GetDebugLevel()
```

`Version`—Returns VisIt's version string.

Synopsis:

```
Version() -> string
```

Returns:

The Version function return a string that represents VisIt's version.

Description:

The Version function returns a string that represents VisIt's version. The version string can be used in Python scripts to make sure that the VisIt module is a certain version before processing the rest of the Python script.

Example:

% visit -cli

```
print “We are running VisIt version %s” % Version()
```

`ActivateDatabase`—Activates a database that has been previously opened.

Synopsis:

```
ActivateDatabase(argument) -> integer
```

Arguments:

`argument`	A string object containing the name of the database to be activated.

Returns:

ActivateDatabase returns 1 on success and 0 on failure.

Description:

The ActivateDatabase function is used to set the active database to a database that has been previously opened. The ActivateDatabase function only works when you are using it to activate a database that you have previously opened. You do not need to use this function unless you frequently toggle between more than one database when making plots or changing time states. While the OpenDatabase function on page 117 can also be used to set the active database, the ActivateDatabase function does not have any side effects that would cause the time state for the new active database to be changed.

Example:

% visit -cli

```
dbs = (“/usr/gapps/visit/data/wave.visit”, \
“/usr/gapps/visit/data/curv3d.silo”)
OpenDatabase(dbs[0], 17)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

OpenDatabase(dbs[1])
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

# Let’s add another plot from the first database.
ActivateDatabase(dbs[0])
AddPlot(“Mesh”, “quadmesh”)
DrawPlots()
```

`AddOperator`—Adds the named operator to the selected plots.

Synopsis:

```
AddOperator(operator) -> integer
AddOperator(operator, all) -> integer
```

Arguments:

`operator`	This is a string containing the name of the operator to be applied.
`all`	This is an optional integer argument that applies the operator to all plots if the value of the argument is not zero.

Returns:

The AddOperator function returns an integer value of 1 for success and 0 for failure.

Description:

The AddOperator function adds a VisIt operator to the selected plots. The operator argument is a string containing the name of the operator to be added to the selected plots. The operatore name must be a valid operator plugin name that is a member of the tuple returned by the OperatorPlugins function (see page 120). The all argument is an integer that determines whether or not the operator is applied to all plots. If the all argument is not provided, the operator is only added to selected plots. Once the AddOperator function is called, the desired operator is added to all selected plots unless the all argument is a non-zero value. When the all argument is a non-zero value, the operator is applied to all plots regardless of whether or not they are selected. Operator attributes are set through the SetOperatorOptions function.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
AddPlot(“Mesh”, “mesh1”)
AddOperator(“Slice”, 1) # Slice both plots
DrawPlots()
```

`AddPlot`—Creates a new plot.

Synopsis:

```
AddPlot(plotType, variableName) -> integer
```

Arguments:

`plotType`	This is a string containing the name of a valid plot plugin type.
`variableName`	This is a string containing a valid variable name for the open database.

Returns:

The AddPlot function returns an integer value of 1 for success and 0 for failure.

Description:

The AddPlot function creates a new plot of the specified type using a variable from the open database. The plotType argument is a string that contains the name of a valid plot plugin type which must be a member of the string tuple that is returned by the PlotPlugins function (see page 127). The variableName argument is a string that contains the name of a variable in the open database. After the AddPlot function is called, a new plot is created and it is made the sole active plot.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Subset”, “mat1”) # Create a subset plot
DrawPlots()
```

`AddWindow`—Creates a new visualization window.

Synopsis:

```
AddWindow()
```

Returns:

The AddWindow function does not a return value.

Description:

The AddWindow function creates a new visualization window and makes it the active window. This function can be used to create up to 16 visualization windows. After that, the AddWindow function has no effect.

Example:

```
import visit
visit.Launch()
visit.AddWindow() # Create window #2
visit.AddWindow() # Create window #3
```

AlterDatabaseCorrelation—Alters a specific database correlation.

Synopsis:

```
AlterDatabaseCorrelation(name, databases, method) -> integer
```

Arguments:

`name`	The name argument must be a string object containing the name of the database correlation to be altered.
`databases`	The databases argument must be a tuple or list of strings containing the fully qualified database names to be used in the database correlation.
`method`	The method argument must be an integer in the range [0,3].

Returns:

The AlterDatabaseCorrelation function returns 1 on success and 0 on failure.

Description:

The AlterDatabaseCorrelation method alters an existing database correlation. A database correlation is a VisIt construct that relates the time states for two or more databases in some way. You would use the AlterDatabaseCorrelation function if you wanted to change the list of databases used in a database correlation or if you wanted to change how the databases are related - the correlation method. The name argument is a string that is the name of the database correlation to be altered. If the name that you pass is not a valid database correlation then the AlterDatabaseCorrelation function fails. The databases argument is a list or tuple of string objects containing the fully-qualified (host:/path/filename) names of the databases to be involved in the database query. The method argument allows you to specify a database correlation method.

Correlation method	Value
IndexForIndexCorrelation	0
StretchedIndexCorrelation	1
TimeCorrelation	2
CycleCorrelation	3

```
dbs = (“/usr/gapps/visit/data/wave.visit”, \
“/usr/gapps/visit/data/wave*.silo database”)
OpenDatabase(dbs[0])
AddPlot(“Pseudocolor”, “pressure”)
OpenDatabase(dbs[1])
AddPlot(“Pseudocolor”, “d”)
# VisIt created an index for index database correlation but we
# want a cycle correlation.
AlterDatabaseCorrelation(“Correlation01”, dbs, 3)
```

`ChangeActivePlotsVar`—Changes the variable for the active plots

Synopsis:

```
ChangeActivePlotsVar(variableName) -> integer
```

Arguments:

`variableName`	The name of the new plot variable.

Returns:

The ChangeActivePlotsVar function returns an integer value of 1 for success and 0 for failure.

Description:

The ChangeActivePlotsVar function changes the plotted variable for the selected plots. This is a useful way to change what is being visualized without having to delete and recreate the current plots. The variableName argument is a string that contains the name of a variable in the open database.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()
SaveWindow()
ChangeActivePlotsVar("v")
```

CheckForNewStates —Checks the specified database for new time states.

Synopsis:

```
CheckForNewStates(name) -> integer
```

Arguments:

`name`	The name argument must be a string that contains the name of a database that has been opened previously.

Returns:

The CheckForNewStates function returns 1 for success and 0 for failure.

Description:

Calculations are often run at the same time as some of the preliminary visualization work is being performed. That said, you might be visualizing the leading time states of a database that is still being created. If you want to force VisIt to add any new time states that were added since you opened the database, you can use the CheckForNewStates function. The name argument must contain the name of a database that has been opened before.

Example:

% visit -cli

```
db = “/usr/gapps/visit/data/wave*.silo database”
OpenDatabase(db)
AddPlot(“Pseudocolor”, “pressure”)
DrawPlots()

SetTimeSliderState(TimeSliderGetNStates() - 1)
# More files appear on disk
CheckForNewStates(db)
SetTimeSliderState(TimeSliderGetNStates() - 1)
```

`ChooseCenterOfRotation`—Allows you to interactively pick a new center of rotation.

Synopsis:

```
ChooseCenterOfRotation() -> integer
ChooseCenterOfRotation(screenX, screenY) -> integer
```

Arguments:

`screenX`	The X coordinate of the pick point in normalized [0,1] screen space.
`screenY`	The Y cooridinate of the pick point in normalized [0,1] screen space.

Returns:

The ChooseCenterOfRotation function returns 1 if successful and 0 if it fails.

Description:

The ChooseCenterOfRotation function allows you to pick a new center of rotation, which is the point about which plots are rotated when you interactively rotate plots. The function can either take zero arguments, in which case you must interactively pick on plots, or it can take two arguments that correspond to the X and Y coordinates of the desired pick point in normalized screen space. When using the two argument version of the ChooseCenterOfRotation function, the X and Y values are floating point values in the range [0,1]. If the ChooseCenterOfRotation function is able to actually pick on plots, yes there must be plots in the vis window, then the center of rotation is updated and the new value is printed to the console.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlots("Pseudocolor", "u")
DrawPlots()

# Interactively choose the center of rotation
ChooseCenterOfRotation()

# Choose a center of rotation using normalized screen
# coordinates and print the value.
ResetView()
ChooseCenterOfRotation(0.5, 0.3)
print "The new center of rotation is:", GetView3D().centerOfRotation
```

ClearCache—Clears the compute engine’s network cache.

Synopsis:

```
ClearCache(host) -> integer
ClearCache(host, simulation) -> integer
ClearCacheForAllEngines() -> integer
```

Arguments:

`host`	The name of the computer where the compute engine is running.
`simulation`	The name of the simulation being processed by the compute engine.

Returns:

The ClearCache and ClearCacheForAllEngines functions return 1 on success and 0 on failure.

Description:

Sometimes during extended VisIt runs, you might want to periodically clear the compute engine’s network cache to reduce the amount of memory being used by the compute engine. Clearing the network cache is also useful when you want to change what the compute engine is working on. For example, you might process a large database and then decide to process another large database. Clearing the network cache beforehand will free up more resources for the compute engine so it can more efficiently process the new database. The host argument is a string object containing the name of the computer on which the compute engine is running. The simulation argument is optional and only applies to when you want to instruct a simulation that is acting as a VisIt compute engine to clear its network cache. If you want to tell more than one compute engine to clear its cache without having to call ClearCache multiple times, you can use the ClearCacheForAllEngines function.

Example:

%visit -cli

```
OpenDatabase(“localhost:very_large_database”)
# Do a lot of work
ClearCache(“localhost”)
OpenDatabase(localhost:another_large_database”)
# Do more work
OpenDatabase(“remotehost:yet_another_database”)
# Do more work
ClearCacheForAllEngines()
```

`Clear`—Clears visualization windows of plots.

Synopsis:

```
ClearAllWindows()
ClearWindow()
```

Returns:

The Clear functions do not return values.

Description:

The ClearWindow function is used to clear out the plots from the active visualization window. The plots are removed from the visualization window but are left in the plot list so that subsequent calls to the DrawPlots function regenerate the plots in the plot list. The ClearAllWindows function preforms the same work as the ClearWindow function except that all windows are cleared of their plots.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

AddWindow()
SetActiveWindow(2) # Make window 2 active
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Subset”, “mat1”)
DrawPlots()

ClearWindow() # Clear the plots in window 2.
DrawPlots() # Redraw the plots in window 2.

ClearAllWindows() # Clear the plots from all windows.
```

`ClearPickPoints`—Clears pick points from the visualization window

Synopsis:

```
ClearPickPoints()
```

Returns:

The ClearPickPoints function does not return a value.

Description:

The ClearPickPoints function removes pick points from the active visualization window. Pick points are the letters that are added to the visualization window where the mouse is clicked when the visualization window is in pick mode.

Example:

% visit -cli

```
# Put the visualization window into pick mode using the popup
# menu and add some pick points.

# Clear the pick points.
ClearPickPoints()
```

ClearReferenceLines—Clears reference lines from the visualization window.

Synopsis:

```
ClearReferenceLines()
```

Returns:

The ClearReferenceLines function does not return a value.

Description:

The ClearReferenceLines function removes reference lines from the active visualization window. Reference lines are the lines that are drawn on a plot to show where you have performed lineouts.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
Lineout((-3.0, 2.0), (2.0, 4.0), ("default", "u", "v"))
ClearReferenceLines()
```

ClearViewKeyframes—Clears any view keyframes that have been set.

Synopsis:

```
ClearViewKeyframes() -> integer
```

Returns:

The ClearViewKeyframes function returns 1 on success and 0 on failure.

Description:

The ClearViewKeyframes function clears any view keyframes that may have been set. View keyframes are used to create complex view behavior such as fly-throughs when VisIt is in keyframing mode.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
k = KeyframeAttributes()
k.enabled, k.nFrames, k.nFramesWasUserSet = 1,10,1
SetKeyframeAttributes(k)
DrawPlots()

SetViewKeyframe()
v1 = GetView3D()
v1.viewNormal = (-0.66609, 0.337227, 0.665283)
v1.viewUp = (0.157431, 0.935425, -0.316537)
SetView3D(v1)
SetTimeSliderState(9)
SetViewKeyframe()
ToggleCameraViewMode()

for i in range(10):
   SetTimeSliderState(i)
ClearViewKeyframes()
```

CloneWindow—Creates a new window that has the same plots, annotations, lights, and view as the active window.

Synopsis:

```
CloneWindow() -> integer
```

Returns:

The CloneWindow function returns an integer value of 1 for success and 0 for failure.

Description:

The CloneWindow function tells the viewer to create a new window, based on the active window, that contains the same plots, annotations, lights, and view as the active window. This function is useful for when you have a window set up like you want and then want to do the same thing in another window using a different database. You can first clone the window and then replace the database.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()

v = ViewAttributes()
v.camera = (-0.505893, 0.32034, 0.800909)
v.viewUp = (0.1314, 0.946269, -0.295482)
v.parallelScale = 14.5472
v.nearPlane = -34.641
v.farPlane = 34.641
v.perspective = 1
SetView3D() # Set the view

a = AnnotationAttributes()
a.backgroundColor = (0, 0, 255, 255)
SetAnnotationAttributes(a) # Set the annotation properties
CloneWindow() # Create a clone of the active window
DrawPlots() # Make the new window draw its plots
```

`CloseComputeEngine`—Closes the compute engine running on a specified host.

Synopsis:

```
CloseComputeEngine() -> integer
CloseComputeEngine(hostName) -> integer
CloseComputeEngine(hostName, simulation) -> integer
```

Arguments:

`hostName`	Optional name of the computer on which the compute engine is running.
`simulation`	Optional name of a simulation.

Returns:

The CloseComputeEngine function returns an integer value of 1 for success and 0 for failure.

Description:

The CloseComputeEngine function tells the viewer to close the compute engine running a specified host. The hostName argument is a string that contains the name of the computer where the compute engine is running. The hostName argument can also be the name “localhost” if you want to close the compute engine on the local machine without having to specify its name. It is not necessary to provide the hostName argument. If the argument is omitted, the first compute engine in the engine list will be closed. The simulation argument can be provided if you want to close a connection to a simulation that is acting as a VisIt compute engine. A compute engine can be launched again by creating a plot or by calling the OpenComputeEngine function.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”) # Launches an engine
AddPlot(“Pseudocolor”, “u”)
DrawPlots()
CloseComputeEngine() # Close the compute engine
```

`CloseDatabase`—Closes the specified database and frees up resources associated with it.

Synopsis:

```
CloseDatabase(name) -> integer
```

Arguments:

`name`	A string object containing the name of the database to close.

Returns:

The CloseDatabase function returns 1 on success and 0 on failure.

Description:

The CloseDatabase function is used to close a specified database and free all resources that were devoted to keeping the database open. This function has an effect similar to ClearCache (see page 40) but it does more in that in addition to clearing the compute engine’s cache, which it only does for the specified database, it also removes all references to the specified database from tables of cached metadata, etc. Note that the CloseDatabase function will fail and the database will not be closed if any plots reference the specified database.

Example:

% visit -cli

```
db = “/usr/gapps/visit/data/globe.silo”
OpenDatabase(db)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

print “This won’t work: retval = %d” % CloseDatabase(db)
DeleteAllPlots()
print “Now it works: retval = %d” % CloseDatabase(db)
```

`ColorTableNames`—Returns a tuple of color table names.

Synopsis:

```
ColorTableNames() -> tuple
```

Returns:

The ColorTableNames function returns a tuple of strings containing the names of the color tables that have been defined.

Description:

The ColorTableNames function returns a tuple of strings containing the names of the color tables that have been defined. This method can be used in case you want to iterate over several color tables.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv2d.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()
for ct in ColorTableNames():
   p = PseudocolorAttributes()
   p.colorTableName = ct
   SetPlotOptions(p)
```

Copy—Copies attributes from one visualization window to another visualization window.

Synopsis:

```
CopyAnnotationsToWindow(source, dest) -> integer
CopyLightingToWindow(source, dest) -> integer
CopyViewTowindow(source, dest) -> integer
CopyPlotsToWindow(source, dest) -> integer
```

Arguments:

`source`	The index (an integer from 1 to 16) of the source window.
`dest`	The index (an integer from 1 to 16) of the destination window.

Returns:

The Copy functions return an integer value of 1 for success and 0 for failure.

Description:

The Copy functions copy attributes from one visualization window to another visualization window. The CopyAnnotationsToWindow function copies the annotations from a source visualization window to a destination visualization window while the CopyLightingAttributes function copies lighting and the CopyViewToWindow function copies the view. The CopyPlotsToWindow function copies the plots from one visualization window to another visualization window but does not also force plots to generate so after copying plots with the CopyPlotsToWindow function, you should also call the DrawPlots function.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

AddWindow()
SetActiveWindow(2)
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Mesh”, “mesh1”)

# Copy window 1’s Pseudocolor plot to window 2.
CopyPlotsToWindow(1, 2)
DrawPlots() # Window 2 will have 2 plots

# Spin the plots around in window 2 using the mouse.
CopyViewToWindow(2, 1) # Copy window 2’s view to window 1.
```

`CreateAnnotationObject`—Creates an annotation object that can directly manipulate annotations in the vis window.

Synopsis:

```
CreateAnnotationObject(annotType) -> annotation object
```

Arguments:

`annotType`	A string containing the name of the type of annotation object to create.

Returns:

CreateAnnotationObject is a factory function that creates annotation objects of different types. The return value, if a valid annotation type is provided, is an annotation object. If the function fails, VisItException is raised.

Description:

CreateAnnotationObject is a factory function that creates different kinds of annotation objects. The annotType argument is a string containing the name of the type of annotation object to create. Each type of annotation object has different properties that can be set. Setting the different properties of an Annotation objects directly modifes annotations in the vis window. Currently there are 2 types of annotation objects:

Annotation type	String
2D text annotation	“Text2D”
Time slider annotation	“TimeSlider”

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/wave.visit”, 17)
AddPlot(“Pseudocolor”, “pressure”)
DrawPlots()

slider = CreateAnnotationObject(“TimeSlider”)
print slider
slider.startColor = (255,0,0,255)
slider.endColor = (255,255,0,255)
```

`CreateDatabaseCorrelation`—Creates a database correlation.

Synopsis:

```
CreateDatabaseCorrelation(name, databases, method) -> integer
```

Arguments:

`name`	String object containing the name of the database correlation to be created.
`databases`	Tuple or list of string objects containing the names of the databases to involve in the database correlation.
`method`	An integer in the range [0,3] that determines the correlation method.

Returns:

The CreateDatabaseCorrelation function returns 1 on success and 0 on failure.

Description:

The CreateDatabaseCorrelation function creates a database correlation, which is a VisIt construct that relates the time states for two or more databases in some way. You would use the CreateDatabaseCorrelation function if you wanted to put plots from more than one time-varying database in the same vis window and then move them both through time in some synchronized way. The name argument is a string that is the name of the database correlation to be created. You will use the name of the database correlation to set the active time slider later so that you can change time states. The databases argument is a list or tuple of string objects containing the fully-qualified (host:/path/filename) names of the databases to be involved in the database query. The method argument allows you to specify a database correlation method.

Correlation method	Value
IndexForIndexCorrelation	0
StretchedIndexCorrelation	1
TimeCorrelation	2
CycleCorrelation	3

Each database correlation has its own time slider that can be used to set the time state of databases that are part of a database correlation. Individual time-varying databases have their own trivial database correlation, consisting of only 1 database. When you call the CreateDatabaseCorrelation function, VisIt creates a new time slider with the same name as the database correlation and makes it be the active time slider.

Example:

% visit -cli

```
dbs = (“/usr/gapps/visit/data/dbA00.pdb”,
“/usr/gapps/visit/data/dbB00.pdb”)
OpenDatabase(dbs[0])
```

```
AddPlot(“FilledBoundary”, “material(mesh)”)
OpenDatabase(dbs[1])
AddPlot(“FilledBoundary”, “material(mesh)”)
DrawPlots()

CreateDatabaseCorrelation(“common”, dbs, 1)

# Creating a new database correlation also creates a new time
# slider and makes it be active.
w = GetWindowInformation()
print “Active time slider: %s” % w.timeSliders[w.activeTimeSlider]

# Animate through time using the “common” database correlation’s
# time slider.
for i in range(TimeSliderGetNStates()):
   SetTimeSliderState(i)
```

`DefineExpression`—Creates a expression variable.

Synopsis:

```
DefineMaterialExpression(variableName, expression)
DefineMeshExpression(variableName, expression)
DefineScalarExpression(variableName, expression)
DefineSpeciesExpression(variableName, expression)
DefineVectorExpression(variableName, expression)
```

Arguments:

`variableName`	The name of the variable to be created.
`expression`	The expression definition.

Returns:

The DefineExpression functions do not return a value.

Description:

The DefineScalarExpression function creates a new scalar variable based on other variables from the open database. Likewise, the DefineMaterialExpression function creates new material variables, DefineMeshExpression creates new mesh variables, DefineSpeciesExpression creates new species variables, and DefineVectorExpression creates new vector variables. Expression variables can be plotted like any other variable. The variableName argument is a string that contains the name of the new variable. The expression argument is a string that contains the definition of the new variable in terms of math operators and pre-existing variable names. Reference the *VisIt User's Manual* if you want more information on creating expressions, such as expression syntax, or a list of built-in expression functions.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
DefineScalarExpression("myvar", "sin(u) + cos(w)")

# Plot the scalar expression variable.
AddPlot("Pseudocolor", "myvar")
DrawPlots()

# Plot a vector expression variable.
DefineVectorExpression("myvec", "{u,v,w}")
AddPlot("Vector", "myvec")
DrawPlots()
```

`DeIconifyAllWindows`—Unhides all of the hidden visualization windows.

Synopsis:

```
DeIconifyAllWindows()
```

Returns:

The DeIconifyAllWindows function does not return a value.

Description:

The DeIconifyAllWindows function unhides all of the hidden visualization windows. This function is usually called after IconifyAllWindows as a way of making all of the hidden visualization windows visible.

Example:

% visit -cli

```
SetWindowLayout(4) # Have 4 windows
IconifyAllWindows()
DeIconifyAllWindows()
```

`DeletePlots`—Deletes plots from the active window’s plot list.

Synopsis:

```
DeleteActivePlots() -> integer
DeleteAllPlots() -> integer
```

Returns:

The Delete functions return an integer value of 1 for success and 0 for failure.

Description:

The Delete functions delete plots from the active window’s plot list. The DeleteActivePlots function deletes all of the active plots from the plot list. There is no way to retrieve a plot once it has been deleted from the plot list. The active plots are set using the SetActivePlots function. The DeleteAllPlots function deletes all plots from the active window’s plot list regardless of whether or not they are active.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
AddPlot(“Contour”, “u”)
AddPlot(“Mesh”, “curvmesh2d”)
DrawPlots()
DeleteActivePlots() # Delete the mesh plot
DeleteAllPlots() # Delete the pseudocolor and contour plots.
```

`DeleteDatabaseCorrelation`—Deletes a database correlation.

Synopsis:

```
DeleteDatabaseCorrelation(name) -> integer
```

Arguments:

`name`	A string object containing the name of the database correlation to delete.

Returns:

The DeleteDatabaseCorrelation function returns 1 on success and 0 on failure.

Description:

The DeleteDatabaseCorrelation function deletes a specific database correlation and its associated time slider. If you delete a database correlation whose time slider is being used for the current time slider, the time slider will be reset to the time slider of the next best suited database correlation. You can use the DeleteDatabaseCorrelation function to remove database correlations that you no longer need such as when you choose to examine databases that have nothing to do with your current databases.

Example:

% visit -cli

```
dbs = (“dbA00.pdb”, “dbB00.pdb”)
OpenDatabase(dbs[0])
AddPlot(“FilledBoundary”, “material(mesh)”)
OpenDatabase(dbs[1])
AddPlot(“FilledBoundary”, “material(mesh)”)
DrawPlots()

CreateDatabaseCorrelation(“common”, dbs, 1)
SetTimeSliderState(10)
DeleteAllPlots()

DeleteDatabaseCorrelation(“common”)
CloseDatabase(dbs[0])
CloseDatabase(dbs[1])
```

`DeleteExpression`—Deletes an expression variable from the expression list.

Synopsis:

```
DeleteExpression(variableName)
```

Arguments:

`variableName` The name of the expression variable to be deleted.

Returns:

The DeleteExpression function returns 1 on success and 0 on failure.

Description:

The DeleteExpression function deletes the definition of an expression. The variableName argument is a string containing the name of the variable expression to be deleted. Any plot that uses an expression that has been deleted will fail to regenerate if its attributes are changed.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
DefineScalarExpression(“myvar”, “sin(u) + cos(w)”)
AddPlot(“Pseudocolor”, “myvar”) # Plot the expression variable.
DrawPlots()

DeleteExpression(“myvar”) # Delete the expression variable myvar.
```

`DeletePlotDatabaseKeyframe`—Deletes a database keyframe for a plot.

Synopsis:

```
DeletePlotDatabaseKeyframe(plotIndex, frame)
```

Arguments:

`plotIndex`	A zero-based integer value corresponding to a plot's index in the plot list.
`frame`	A zero-based integer value corresponding to a database keyframe at a particular animation frame.

Returns:

DeletePlotDatabaseKeyframe does not return a value.

Description:

The DeletePlotDatabaseKeyframe function removes a database keyframe from a specific plot. A database keyframe represents the database time state that will be used at a given animation frame when VisIt's keyframing mode is enabled. The plotIndex argument is a zero-based integer that is used to identify a plot in the plot list. The frame argument is a zero-based integer that is used to identify the frame at which a database keyframe is to be removed for the specified plot.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/wave.visit")
k = GetKeyframeAttributes()
k.enabled,k.nFrames,k.nFramesWasUserSet = 1,20,1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "pressure")
SetPlotDatabaseState(0, 0, 60)
# Repeat time state 60 for the first few animation frames by adding a
keyframe at frame 3.
SetPlotDatabaseState(0, 3, 60)
SetPlotDatabaseState(0, 19, 0)
DrawPlots()

ListPlots()

# Delete the database keyframe at frame 3.
DeletePlotDatabaseKeyframe(0, 3)
ListPlots()
```

`DeletePlotKeyframe`—Deletes a plot keyframe at a specified frame.

Synopsis:

```
DeletePlotKeyframe(plotIndex, frame)
```

Arguments:

`plotIndex`	A zero-based integer value corresponding to a plot's index in the plot list.
`frame`	A zero-based integer value corresponding to a plot keyframe at a particular animation frame.

Returns:

DeletePlotKeyframe does not return a value.

Description:

The DeletePlotKeyframe function removes a plot keyframe from a specific plot. A plot keyframe is the set of plot attributes at a specified frame. Plot keyframes are used to determine what plot attributes will be used at a given animation frame when VisIt's keyframing mode is enabled. The plotIndex argument is a zero-based integer that is used to identify a plot in the plot list. The frame argument is a zero-based integer that is used to identify the frame at which a keyframe is to be removed.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/wave.visit")
k = GetKeyframeAttributes()
k.enabled,k.nFrames,k.nFramesWasUserSet = 1,20,1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "pressure")
# Set up plot keyframes so the Pseudocolor plot's min will change
# over time.
p0 = PseudocolorAttributes()
p0.minFlag,p0.min = 1,0.0
p1 = PseudocolorAttributes()
p1.minFlag,p1.min = 1, 0.5
SetPlotOptions(p0)
SetTimeSliderState(19)
SetPlotOptions(p1)
SetTimeSliderState(0)
DrawPlots()

ListPlots()

# Iterate over all animation frames and wrap around to the first one.
for i in list(range(TimeSliderGetNStates())) + [0]:
   SetTimeSliderState(i)

# Delete the plot keyframe at frame 19 so the min won't
# change anymore.
DeletePlotKeyframe(19)
ListPlots()
SetTimeSliderState(10)
```

`DeleteViewKeyframe`—Deletes a view keyframe at a specified frame.

Synopsis:

```
DeleteViewKeyframe(frame)
```

Arguments:

`frame`	A zero-based integer value corresponding to a view keyframe at a particular animation frame.

Returns:

DeleteViewKeyframe returns 1 on success and 0 on failure.

Description:

The DeleteViewKeyframe function removes a view keyframe at a specified frame. View keyframes are used to determine what view will be used at a given animation frame when VisIt's keyframing mode is enabled. The frame argument is a zero-based integer that is used to identify the frame at which a keyframe is to be removed.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
k = KeyframeAttributes()
k.enabled, k.nFrames, k.nFramesWasUserSet = 1,10,1
SetKeyframeAttributes(k)
AddPlot("Pseudocolor", "u")
DrawPlots()

# Set some view keyframes
SetViewKeyframe()
v1 = GetView3D()
v1.viewNormal = (-0.66609, 0.337227, 0.665283)
v1.viewUp = (0.157431, 0.935425, -0.316537)
SetView3D(v1)
SetTimeSliderState(9)
SetViewKeyframe()
ToggleCameraViewMode()

# Iterate over the animation frames to watch the view change.
for i in list(range(10)) + [0]:
   SetTimeSliderState(i)
# Delete the last view keyframe, which is on frame 9.
DeleteViewKeyframe(9)
# Iterate over the animation frames again. The view should stay
# the same.
for i in range(10):
   SetTimeSliderState(i)
```

DeleteWindow—Deletes the active visualization window.

Synopsis:

```
DeleteWindow() -> integer
```

Returns:

The DeleteWindow function returns an integer value of 1 for success and 0 for failure.

Description:

The DeleteWindow function deletes the active visualization window and makes the visualization window with the smallest window index the new active window. This function has no effect when there is only one remaining visualization window.

Example:

% visit -cli

```
DeleteWindow() # Does nothing since there is only one window
AddWindow()
DeleteWindow() # Deletes the new window.
```

`DemoteOperator`—Moves an operator closer to the database in the visualization pipeline.

Synopsis:

```
DemoteOperator(opIndex) -> integer
DemoteOperator(opIndex, applyToAllPlots) -> integer
```

Arguments:

`opIndex`	A zero-based integer corresponding to the operator that should be demoted.
`applyAll`	An integer flag that causes all plots in the plot list to be affected when it is non-zero.

Returns:

DemoteOperator returns 1 on success and 0 on failure.

Description:

The DemoteOperator function moves an operator closer to the database in the visualization pipeline. This allows you to change the order of operators that have been applied to a plot without having to remove them from the plot. For example, consider moving a Slice to before a Reflect operator when it had been the other way around. Changing the order of operators can result in vastly different results for a plot. The opposite function is PromoteOperator (see page 129).

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
AddOperator(“Slice”)
s = SliceAttributes()
s.project2d = 0
s.originPoint = (0,5,0)
s.originType=s.Point
s.normal = (0,1,0)
s.upAxis = (-1,0,0)
SetOperatorOptions(s)
AddOperator("Reflect")
DrawPlots()

# Now reflect before slicing. We’ll only get 1 slice plane
# instead of 2.
DemoteOperator(1)
DrawPlots()
```

`DisableRedraw`—Prevents the active visualization window from redrawing itself.

Synopsis:

```
DisableRedraw()
```

Returns:

The DisableRedraw function does not return a value.

Description:

The DisableRedraw function prevents the active visualization window from ever redrawing itself. This is a useful function to call when performing many operations that would cause unnecessary redraws in the visualization window. The effects of this function are undone by calling the RedrawWindow function.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Contour”, “u”)
AddPlot(“Pseudocolor”, “w”)
DrawPlots()

DisableRedraw()
AddOperator(“Slice”)

# Set the slice operator attributes

# Redraw now that th operator attributes are set. This will
# prevent 1 redraw.
RedrawWindow()
```

`DrawPlots`—Draws any new plots

Synopsis:

```
DrawPlots() -> integer
```

Returns:

The DrawPlots function returns an integer value of 1 for success and 0 for failure.

Description:

The DrawPlots function forces all new plots in the plot list to be drawn. Plots are added and then their attributes are modified. Finally, the DrawPlots function is called to make sure all of the new plots draw themselves in the visualization window. This function has no effect if all of the plots in the plot list are already drawn.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots() # Draw the new pseudocolor plot.
```

`EnableTool`—Sets the enabled state of an interactive tool in the active visualization window.

Synopsis:

```
EnabledTool(toolIndex, activeFlag)
```

Arguments:

`toolIndex`	This is an integer that corresponds to an interactive tool. (Line tool = 0, Plane tool = 1, Sphere tool = 2)
`activeFlag`	A value of 1 enables the tool while a value of 0 disables the tool.

Returns:

The EnableTool function returns 1 on success and 0 on failure.

Description:

The EnableTool function is used to set the enabled state of an interactive tool in the active visualization window. The toolIndex argument is an integer index that corresponds to a certain tool. The activeFlag argument is an integer value (0 or 1) that indicates whether to turn the tool on or off.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

EnableTool(0, 1) # Turn on the line tool.
EnableTool(1,1) # Turn on the plane tool.
EnableTool(2,1) # Turn on the sphere tool.
EnableTool(2,0) # Turn off the sphere tool.
```

`EvalCubic`—Interpolates between four values using a cubic polynomial.

Synopsis:

```
EvalCubic(t, c0, c1, c2, c3) -> f(t)
```

Arguments:

`t`	A floating point number in the range [0., 1.] that represents the distance from c0 to c3.
`c0`	The first control point. f(0) = c0. Any object that can be used in an arithmetic expression can be passed for c0.
`c1`	The second control point. Any object that can be used in an arithmetic expression can be passed for c1.
`c2`	The third control point. Any object that can be used in an arithmetic expression can be passed for c2.
`c3`	The last control point. f(1) = c3. Any object that can be used in an arithmetic expression can be passed for c3.

Returns:

The EvalCubic function returns the interpolated value for t taking into account the control points that were passed in.

Description:

The EvalCubic function takes in four objects and blends them using a cubic polynomial and returns the blended value.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

v0 = GetView3D()
# rotate the plots
v1 = GetView3D()
# rotate the plots again.
v2 = GetView3D()
# rotate the plots one last time.
v3 = GetView3D()
# Fly around the plots using the views that have been specified.
nSteps = 100

for i in range(nSteps):
   t = float(i) / float(nSteps - 1)
   newView = EvalCubic(t, v0, v1, v2, v3)
   SetView3D(newView)
```

`EvalCubicSpline`—Interpolates between N values using piece-wise cubic splines.

Synopsis:

```
EvalCubicSpline(t, weights, values) -> f(t)
```

Arguments:

`t`	A floating point value in the range [0., 1.] that represents the distance from the first control point to the last control point.
`weights`	A tuple of N floating point values in the range [0., 1.] that represent how far along in parameterized space, the values will be located.
`values`	A tuple of N objects to be blended. Any objects that can be used in arithmetic expressions can be passed.

Returns:

The EvalCubicSpline function returns the interpolated value for t considering the objects that were passed in.

Description:

The EvalCubicSpline function takes in N objects to be blended and blends them using piece-wise cubic polynomials and returns the blended value. For an example, see page 22.

`EvalLinear`—Interpolates linearly between two values.

Synopsis:

```
EvalLinear(t, value1, value2) -> f(t)
```

Arguments:

`t`	A floating point value in the range [0., 1.] that represents the distance between the first and last control point in parameterized space.
`value1`	Any object that can be used in an arithmetic expression. f(0) = value1.
`value2`	Any object that can be used in an arithmetic expression. f(1) = value2.

Returns:

The EvalLinear function returns an interpolated value for t based on the objects that were passed in.

Description:

The EvalLinear function linearly interpolates between two values and returns the result.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

c0 = GetView3D()
c1 = GetView3D()
c1.viewNormal = (-0.499159, 0.475135, 0.724629)
c1.viewUp = (0.196284, 0.876524, -0.439521)

nSteps = 100
for i in range(nSteps):
   t = float(i) / float(nSteps - 1)
   v = EvalLinear(t, c0, c1)
   SetView3D(v)
```

`EvalQuadratic`—Interpolates between three values using a quadratic polynomial.

Synopsis:

EvalQuadratic(t, c0, c1, c2,) -> f(t)

Arguments:

`t`	A floating point number in the range [0., 1.] that represents the distance from c0 to c3.
`c0`	The first control point. f(0) = c0. Any object that can be used in an arithmetic expression can be passed for c0.
`c1`	The second control point. Any object that can be used in an arithmetic expression can be passed for c1.
`c2`	The last control point. f(1) = c2. Any object that can be used in an arithmetic expression can be passed for c2.

Returns:

The EvalQuadratic function returns the interpolated value for t taking into account the control points that were passed in.

Description:

The EvalQuadratic function takes in four objects and blends them using a cubic polynomial and returns the blended value.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

v0 = GetView3D()
# rotate the plots
v1 = GetView3D()
# rotate the plots one last time.
v2 = GetView3D()

# Fly around the plots using the views that have been specified.
nSteps = 100
for i in range(nSteps):
   t = float(i) / float(nSteps - 1)
   newView = EvalQuadratic(t, v0, v1, v2)
   SetView3D(newView)
```

`Expressions`—Returns a tuple of expression names and definitions.

Synopsis:

```
Expressions() -> tuple of expression tuples
```

Returns:

The Expressions function returns a tuple of tuples that contain two strings that give the expression name and definition.

Description:

The Expressions function returns a tuple of tuples that contain two strings that give the expression name and definition. This function is useful for listing the available expressions or for iterating through a list of expressions in order to create plots.

Example:

% visit -cli

```
SetWindowLayout(4)
DefineScalarExpression(“sin_u”, “sin(u)”)
DefineScalarExpression(“cos_u”, “cos(u)”)
DefineScalarExpression(“neg_u”, “-u”)
DefineScalarExpression(“bob”, “sin_u + cos_u”)
for i in range(1,5):
   SetActiveWindow(i)
   OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
   exprName = Expressions()[i-1][0]
   AddPlot(“Pseudocolor”, exprName)
   DrawPlots()
```

GetActiveColorTable—Returns the name of the active color table.

Synopsis:

```
GetActiveContinuousColorTable() -> string
GetActiveDiscreteColorTable() -> string
```

Returns:

Both functions return a string object containing the name of a color table.

Description:

A color table is a set of color values that are used as the colors for plots. VisIt supports two flavors of color table: continuous and discrete. A continuous color table is defined by a small set of color control points and the colors specified by the color control points are interpolated smoothly to fill in any gaps. Continuous color tables are used for plots that need to be colored smoothly by a variable (e.g. Pseudocolor plot). A discrete color table is a set of color control points that are used to color distinct regions of a plot (e.g. Subset plot). VisIt supports the notion of default continuous and default discrete color tables so plots can just use the “default” color table. This lets you change the color table used by many plots by just changing the “default” color table. The GetActiveContinuousColorTable function returns the name of the default continuous color table. The GetActiveDiscreteColorTable function returns the name of the default discrete color table.

Example:

% visit -cli

```
print “Default continuous color table: %s” % \
GetActiveContinuousColorTable()

print “Default discrete color table: %s” % \
GetActiveDiscreteColorTable()
```

`GetActiveTimeSlider`—Returns the name of the active time slider.

Synopsis:

```
GetActiveTimeSlider() -> string
```

Returns:

The GetActiveTimeSlider function returns a string containing the name of the active time slider.

Description:

VisIt can support having multiple time sliders when you have opened more than one time-varying database. You can then use each time slider to independently change time states for each database or you can use a database correlation to change time states for all databases simultaneously. Every time-varying database has a database correlation and every database correlation has its own time slider. If you want to query to determine which time slider is currently the active time slider, you can use the GetActiveTimeSlider function.

Example:

% visit -cli

```
OpenDatabase(“dbA00.pdb”)
AddPlot(“FilledBoundary”, “material(mesh)”)
OpenDatabase(“dbB00.pdb”)
AddPlot(“FilledBoundary”, “materials(mesh)”)
print “Active time slider: %s” % GetActiveTimeSlider()
CreateDatabaseCorrelation(“common”, (“dbA00.pdb”, “dbB00.pdb”), 2)
print “Active time slider: %s” % GetActiveTimeSlider()
```

`GetAnimationTimeout`—Returns the animation timeout in milliseconds.

Synopsis:

```
GetAnimationTimeout() -> integer
```

Returns:

The GetAnimationTimeout function returns an integer that contains the time interval, measured in milliseconds, between the rendering of animation frames.

Description:

The GetAnimationTimeout returns an integer that contains the time interval, measured in milliseconds, between the rendering of animation frames.

Example:

% visit -cli

```
print “Animation timeout = %d” % GetAnimationTimeout()
```

`GetAnimationNumStates`—Returns the number of time steps in an animation.

Synopsis:

```
GetAnimationNumStates() -> integer
```

Returns:

GetAnimationNumStates returns an integer value containing the number of time steps in the open database.

Description:

The GetAnimationNumStates function returns the number of time steps in the active window’s open database. This function will always return 1 unless a “.visit” file was opened as the database. When a “.visit” file is opened, the GetAnimationNumStates function returns the number of time steps in the time-varying database.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
print “Number of time steps = %d” % GetAnimationNumStates()
```

GetAnnotationAttributes—Returns an object containing the active visualization window's annotation attributes.

Synopsis:

```
GetAnnotationAttributes() -> AnnotationAttributes object
```

Returns:

The GetAnnotationAttributes function returns an AnnotationAttributes object that contains the annotation settings for the active visualization window.

Description:

The GetAnnotationAttributes function returns an AnnotationAttributes object that contains the annotation settings for the active visualization window. It is often useful to retrieve the annotation settings and modify them to suit the visualization.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()

a = GetAnnotationAttributes()
print a
a.backgroundMode = a.BACKGROUNDMODE_GRADIENT
a.gradientColor1 = (0, 0, 255)
SetAnnotationAttributes(a)
```

`GetAnnotationObject`—Returns a reference to the annotation object at the specified index in the annotation object list.

Synopsis:

```
GetAnnotationObject(index) -> Annotation object
```

Arguments:

`index`	A zero-based integer index into the annotation object list.

Returns:

GetAnnotationObject returns a reference to an annotation object that was created using the Create-AnnotationObject function.

Description:

GetAnnotationObject returns a reference to an annotation object that was created using the Create-AnnotationObject function (see page 50). The index argument is a zero-based integer that specifies the index of the object for which we want to return another reference. This function is not currently necessary unless the annotation object that you used to create an annotation has gone out of scope and you need to create another reference to the object to set its properties.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
AddPlot(“Pseudocolor”, “pressure”)
DrawPlots()

a = CreateAnnotationObject(“TimeSlider”)
ref = GetAnnotationObject(0)
print ref
```

`GetDatabaseNStates`—Returns the numberof time states in the active database.

Synopsis:

```
GetDatabaseNStates() -> integer
```

Returns:

Returns the number of time states in the active database or 0 if there is no active database.

Description:

GetDatabaseNStates returns the number of time states in the active database, which is not the same as the number of states in the active time slider. Time sliders can have different lengths due to database correlations and keyframing. Use this function when you need the actual number of time states in the active database.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/wave*.silo database”)
print “Number of time states: %d” % GetDatabaseNStates()
```

GetDomains—Returns a tuple containing the names of all of the domain subsets for the active plot.

Synopsis:

```
GetDomains() -> tuple of strings
```

Returns:

GetDomains returns a tuple of strings.

Description:

GetDomains returns a tuple containing the names of all of the domain subsets for a plot that was created using a database with multiple domains. This function can be used in specialized logic that iterates over domains to turn them on or off in some programmed way.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/multi_ucd3d.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

doms = GetDomains()
print doms

# Turn off all but the last domain, one after the other.
for d in doms[:-1]:
   TurnDomainsOff(d)
```

`GetEngineList`—Returns a tuple containing the names of the compute engines.

Synopsis:

```
GetEngineList() -> tuple of strings
```

Returns:

GetEngineList returns a tuple of strings that contain the names of the computers on which compute engines are running.

Description:

The GetEngineList function returns a tuple of strings containing the names of the computers on which compute engines are running. This function can be useful if engines are going to be closed and opened explicitly in the Python script. The contents of the tuple can be used to help determine which compute engines should be closed or they can be used to determine if a compute engine was successfully launched.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
OpenDatabase(“mcr:/usr/gapps/visit/data/globe.silo”)
AddPlot(“Mesh”, “mesh1”)
DrawPlots()

for name in GetEngineList():
   print “VisIt has a compute engine running on %s” % name

CloseComputeEngine(GetEngineList()[1])
```

GetGlobalAttributes—Returns a GlobalAttributes object.

Synopsis:

```
GetGlobalAttributes() -> GlobalAttributes object
```

Returns:

Returns a GlobalAttributes object that has been initialized.

Description:

The GetGlobalAttributes function returns a GlobalAttributes object that has been initialized with the current state of the viewer proxy’s GlobalAttributes object. The GlobalAttributes object contains read-only information about the list of sources, the list of windows, and various flags that can be queried.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()
g = GetGlobalAttributes()
print g
```

`GetGlobalLineoutAttributes`—Returns a GlobalLineoutAttributes object.

Synopsis:

```
GetGlobalLineoutAttributes() -> GlobalLineoutAttributes object
```

Returns:

Returns an initialized GlobalLineoutAttributes object.

Description:

The GetGlobalLineoutAttributes function returns an initialized GlobalLineoutAttributes object. The GlobalLineoutAttributes, as suggested by its name, contains global properties that apply to all lineouts. You can use the GlobalLineoutAttributes object to turn on lineout sampling, specify the destination window, etc. for curve plots created as a result of performing lineouts. Once you make changes to the object by setting its properties, use the SetGlobalLineoutAttributes function to make VisIt use the modified global lineout attributes.

Example:

% visit -cli

```
SetWindowLayout(4)
OpenDatabase(“/usr/gapps/visit/data/curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

g = GetGlobalLineoutAttributes()
print g
g.samplingOn = 1
g.windowId = 4
g.createWindow = 0
g.numSamples = 100
SetGlobalLineoutAttributes(g)

Lineout((-3,2),(3,3),("default"))
```

`GetInteractorAttributes`—Returns an InteractorAttributes object.

Synopsis:

```
GetInteractorAttributes() -> InteractorAttributes object
```

Returns:

Returns an initialized InteractorAttributes object.

Description:

The GetInteractorAttributes function returns an initialized InteractorAttributes object. The InteractorAttributes object can be used to set certain interactor properties. Interactors, can be thought of as how mouse clicks and movements are translated into actions in the vis window. To set the interactor attributes, first get the interactor attributes using the GetInteractorAttributes function. Once you've set the object's properties, call the SetInteractorAttributes function to make VisIt use the new interactor attributes.

Example:

% visit -cli

```
ia = GetInteractorAttributes()
print ia
ia.showGuidelines = 0
SetInteractorAttributes(ia)
```

`GetKeyframeAttributes`—Returns an initialized KeyframeAttributes object.

Synopsis:

```
GetKeyframeAttributes() -> KeyframeAttributes object
```

Returns:

GetKeyframeAttributes returns an initialized KeyframeAttributes object.

Description:

Use the GetKeyframeAttributes function when you want to examine a KeyframeAttributes object so you can determine VisIt's state when it is in keyframing mode. The KeyframeAttributes object allows you to see whether VisIt is in keyframing mode and, if so, how many animation frames are in the current keyframe animation.

Example:

% visit -cli

```
k = GetKeyframeAttributes()
print k
k.enabled,k.nFrames,k.nFramesWasUserSet = 1, 100, 1
SetKeyframeAttributes(k)
```

`GetLastError`—Returns a string containing the last error message that VisIt issued.

Synopsis:

```
GetLastError() -> string
```

Returns:

GetLastError returns a string containing the last error message that VisIt issued.

Description:

The GetLastError function returns a string containing the last error message that VisIt issued.

Example:

% visit -cli

```
OpenDatabase(“/this/database/does/not/exist”)
print “VisIt Error: %s” % GetLastError()
```

`GetLight`—Returns a light object containing the attributes for a specified light.

Synopsis:

```
GetLight(index) -> LightAttributes object
```

Arguments:

`index`	A zero-based integer index into the light list. Index can be in the range [0,7].

Returns:

GetLight returns a LightAttributes object.

Description:

The GetLight function returns a LightAttributes object containing the attributes for a specific light. You can use the LightAttributes object that GetLight returns to set light properties and then you can pass the object to SetLight to make VisIt use the light properties that you’ve set.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot("Pseudocolor", "w")
p = PseudocolorAttributes()
p.colorTableName = “xray”
SetPlotOptions(p)
DrawPlots()

InvertBackgroundColor()

light = GetLight(0)
print light
light.enabledFlag = 1
light.direction = (0,-1,0)
light.color = (255,0,0,255)
SetLight(0, light)

light.color,light.direction = (0,255,0,255), (-1,0,0)
SetLight(1, light)
```

`GetLocalName`—Gets the local user or host name.

Synopsis:

```
GetLocalHostName() -> string
GetLocalUserName() -> string
```

Returns:

Both functions return a string.

Description:

These functions are useful for determining the name of the local computer or the account name of the user running VisIt. The GetLocalHostName function returns a string that contains the name of the local computer. The GetLocalUserName function returns a string containing the name of the user running VisIt.

Example:

% visit -cli

```
print “Local machine name is: %s” % GetLocalHostName()
print “My username: %s” % GetLocalUserName()
```

GetMaterialAttributes—Returns a MaterialAttributes object containing VisIt’s current material interface reconstruction settings.

Synopsis:

```
GetMaterialAttributes() -> MaterialAttributes object
```

Returns:

Returns a MaterialAttributes object.

Description:

The GetMaterialAttributes function returns a MaterialAttributes object that contains VisIt’s current material interface reconstruction settings. You can set properties on the MaterialAttributes object and then pass it to SetMaterialAttributes to make VisIt use the new material attributes that you’ve specified:

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/allinone00.pdb”)
AddPlot(“Pseudocolor”, “mesh/mixvar”)
p = PseudocolorAttributes()
p.min,p.minFlag = 4.0, 1
p.max,p.maxFlag = 13.0, 1
SetPlotOptions(p)
DrawPlots()

# Tell VisIt to always do material interface reconstruction.
m = GetMaterialAttributes()
m.forceMIR = 1
SetMaterialAttributes(m)
ClearWindow()

# Redraw the plot forcing VisIt to use the mixed variable information.
DrawPlots()
```

GetMaterials—Returns a string tuple of material names for the current plot’s database.

Synopsis:

```
GetMaterials() -> tuple of strings
```

Returns:

The GetMaterials function returns a tuple of strings.

Description:

The GetMaterials function returns a tuple of strings containing the names of the available materials for the current plot’s database. Note that the active plot’s database must have materials for this function to return a tuple that has any string objects in it. Also, you must have at least one plot. You can use the materials returned by the GetMaterials function for a variety of purposes including turning materials on or off.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/allinone00.pdb”)
AddPlot(“Pseudocolor”, “mesh/mixvar”)
DrawPlots()

mats = GetMaterials()
for m in mats[:-1]:
   TurnMaterialOff(m)
```

GetNumPlots—Returns the number of plots in the active window.

Synopsis:

```
GetNumPlots() -> integer
```

Returns:

Returns the number of plots in the active window.

Description:

The GetNumPlots function returns the number of plots in the active window.

Example:

% visit -cli

```
print “Number of plots”, GetNumPlots()
OpenDatabase(“/usr/gapps/visit/data/curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
print “Number of plots”, GetNumPlots()
AddPlot(“Mesh”, “curvmesh2d”)
DrawPlots()

print “Number of plots”, GetNumPlots()
```

`GetPickAttributes`—Returns the current pick attributes.

Synopsis:

```
GetPickAttributes() -> PickAttributes object
```

Returns:

GetPickAttributes returns a PickAttributes object.

Description:

The GetPickAttributes object returns the pick settings that VisIt is currently using when it performs picks. These settings mainly determine which pick information is displayed when pick results are printed out but they can also be used to select auxiliary variables and generate time curves. You can examing the settings and you can set properties on the returned object. Once you've changed pick settings by setting properties on the object, you can pass the altered object to the SetPickAttributes function to force VisIt to use the new pick settings.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/allinone00.pdb")
AddPlot("Pseudocolor", "mesh/ireg")
DrawPlots()

p = GetPickAttributes()
print p
p.variables = ("default", "mesh/a", "mesh/mixvar")
SetPickAttributes(p)

# Now do some interactive picks and you'll see pick information
# for more than 1 variable.
p.doTimeCurve = 1
SetPickAttributes(p)

# Now do some interactive picks and you'll get time-curves in
# a new window.
```

GetPickOutput—Returns a string containing the output from the last pick.

Synopsis:

```
GetPickOutput() -> string
```

Returns:

Returns a string containing the output from the last pick.

Description:

The GetPickOutput returns a string object that contains the output from the last pick.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/rect2d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

Pick((0.4, 0.6), ("default", "u", "v"))
s = GetPickOutput()
print s
```

`GetPipelineCachingMode`—Returns if pipelines are cached in the viewer.

Synopsis:

```
GetPipelineCachingMode() -> integer
```

Returns:

The GetPipelineCachingMode function returns 1 if pipelines are being cached and 0 otherwise.

Description:

The GetPipelineCachingMode function returns whether or not pipelines are being cached in the viewer. For animations of long time sequences, it is often useful to turn off pipeline caching so the viewer does not run out of memory.

Example:

%visit -cli

```
offon = (“off”, “on”)
print “Pipeline caching is %s” % offon[GetPipelineCachingMode()]
```

`GetQueryOutput`—Returns information about the last query.

Synopsis:

```
GetQueryOutputString() -> string
GetQueryOutputValue() -> double, tuple of doubles
```

Returns:

GetQueryOutputString returns a string.

GetQueryOutputValue returns a single double precision number or a tuple of double precision numbers.

Description:

Both the GetQueryOutputString and GetQueryOutputValue functions return information about the last query to be executed but the type of information returns differs. GetQueryOutputString returns a string containing the output of the last query. GetQueryOutputValue returns a single number or tuple of numbers, depending on the nature of the last query to be executed.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/rect2d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

Query(“MinMax”)
print GetQueryOutputString()
print “The min is: %g and the max is: %g” % GetQueryOutputValue()
```

GetQueryOverTimeAttributes—Returns a QueryOverTimeAttributes object containing the settings that VisIt currently uses for queries over time.

Synopsis:

```
GetQueryOverTimeAttributes() -> QueryOverTimeAttributes object
```

Returns:

GetQueryOverTimeAttributes returns a QueryOverTimeAttributes object.

Description:

The GetQueryOverTimeAttributes function returns a QueryOverTimeAttributes object containing the settings that VisIt currently uses for query over time. You can use the returned object to change those settings by first setting object properties and then by passing the modified object to the SetQueryOverTimeAttributes function (see page 169).

Example:

% visit -cli

```
SetWindowLayout(4)
OpenDatabase(“/usr/gapps/visit/data/allinone00.pdb”)
AddPlot(“Pseudocolor”, “mesh/mixvar”)
DrawPlots()

qot = GetQueryOverTimeAttributes()
print qot

# Make queries over time go to window 4.
qot.createWindow,q.windowId = 0, 4
SetQueryOverTimeAttributes(qot)
QueryOverTime(“Min”)

# Make queries over time only use half of the number of time states.
qot.endTimeFlag,qot.endTime = 1, GetDatabaseNStates() / 2
SetQueryOverTimeAttributes(qot)
QueryOverTime(“Min”)
ResetView()
```

GetRenderingAttributes—Returns a RenderingAttributes object containing VisIt’s current rendering settings.

Synopsis:

```
GetRenderingAttributes() -> RenderingAttributes object
```

Returns:

Returns a RenderingAttributes object.

Description:

The GetRenderingAttributes function returns a RenderingAttributes object that contains the rendering settings that VisIt currently uses. The RenderingAttributes object contains information related to rendering such as whether or not specular highlights or shadows are enabled. The RenderingAttributes object also contains information scalable rendering such as whether or not it is currently in use and the scalable rendering threshold.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Surface”, “hgslice”)
DrawPlots()

v = GetView3D()
v.viewNormal = (-0.215934, -0.454611, 0.864119)
v.viewUp = (0.973938, -0.163188, 0.157523)
v.imageZoom = 1.64765
SetView3D(v)

light = GetLight(0)
light.direction = (0,1,-1)
SetLight(0, light)

r = GetRenderingAttributes()
r.scalableActivationMode = r.Always
r.doShadowing = 1
SetRenderingAttributes(r)
```

`GetSaveWindowAttributes`—Returns an object that contains the attributes used to save windows.

Synopsis:

```
GetSaveWindowAttributes() -> SaveWindowAttributes object
```

Returns:

This function returns a VisIt SaveWindowAttributes object that contains the attributes used in saving windows.

Description:

The GetSaveWindowAttributes function returns a SaveWindowAttributes object that is a structure containing several fields which determine how windows are saved to files. The object that us returned can be modified and used to set the save window attributes.

Example:

% visit -cli

```
s = GetSaveWindowAttributes()
print s
s.width = 600
s.height = 600
s.format = s.FILEFORMAT_RGB
print s
```

GetTimeSliders—Returns a tuple containing the names of all of the available time sliders.

Synopsis:

```
GetTimeSliders() -> tuple of strings
```

Returns:

GetTimeSliders returns a tuple of strings.

Description:

The GetTimeSliders function returns a tuple of strings containing the names of each of the available time sliders. The list of time sliders contains the names of any open time-varying database, all database correlations, and the keyframing time slider if VisIt is in keyframing mode.

Example:

% visit -cli

```
path = “/usr/gapps/visit/data/”
dbs = (path + “/dbA00.pdb”, path + “dbB00.pdb”, path + “dbC00.pdb”)
for db in dbs:
   OpenDatabase(db)
   AddPlot(“FilledBoundary”, “material(mesh)”)
DrawPlots()

CreateDatabaseCorrelation(“common”, dbs, 1)
print “The list of time sliders is: “, GetTimeSliders()
```

`GetView`—Return an object containing the current view.

Synopsis:

```
GetView2D() -> View2DAttributes object
GetView3D() -> View3DAttributes object
GetViewCurve() -> ViewCurveAttributes object
```

Returns:

Both functions return objects that represent the curve, 2D, or 3D view information.

Description:

The GetView functions return ViewAttributes objects which describe the current camera location. The GetView2D function should be called if the active visualization window contains 2D plots. The GetView3D function should be called to get the view if the active visualization window contains 3D plots. The GetViewCurve function should be called if the active visualization window contains 1D curve plots.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

# Change the view interactively using the mouse.
v0 = GetView3D()
# Change the view again using the mouse
v1 = GetView3D()
print v0

for i in range(0,20):
   t = float(i) / 19.
   v2 = (1. - t) * v1 + t * v0
   SetView3D(v2) # Animate the view back to the first view.
```

GetWindowInformation—Returns a WindowInformation object that contains information about the active visualization window.

Synopsis:

```
GetWindowInformation() -> WindowInformation object
```

Returns:

The GetWindowInformation object returns a WindowInformation object.

Description:

The GetWindowInformation object returns a WindowInformation object that contains information about the active visualization window. The WindowInformation object contains the name of the active source, the active time slider index, the list of available time sliders and their current states, as well as certain window flags that determine whether a window's view is locked, etc. Use the WindowInformation object if you need to query any of these types of information in your script to influence how it behaves.

Example:

```
path = “/usr/gapps/visit/data/”
dbs = (path + “dbA00.pdb”, path + “dbB00.pdb”, path + “dbC00.pdb”)
for db in dbs:
   OpenDatabase(db)
   AddPlot(“FilledBoundary”, “material(mesh)”)
DrawPlots()

CreateDatabaseCorrelation(“common”, dbs, 1)

# Get the list of available time sliders.
tsList = GetWindowInformation().timeSliders

# Iterate through “time” on each time slider.
for ts in tsList:
   SetActiveTimeSlider(ts)
   for state in range(TimeSliderGetNStates()):
      SetTimeSliderState(state)

# Print the window information to examine the other attributes
# that are available.
GetWindowInformation()
```

`HideActivePlots`—Hides the active plots in the active visualization window.

Synopsis:

```
HideActivePlots() -> integer
```

Returns:

The HideActivePlots function returns an integer value of 1 for success and 0 for failure.

Description:

The HideActivePlots function tells the viewer to hide the active plots in the active visualization window.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddPlot("Mesh", "mesh1")
DrawPlots()

SetActivePlots(0)
HideActivePlots()
AddPlot("FilledBoundary", "mat1")
DrawPlots()
```

`HideToolbars`—Hides the visualization window’s toolbars.

Synopsis:

```
HideToolbars() -> integer
HideToolbars(allWindows) ->integer
```

Arguments:

`allWindows`	An integer value that tells VisIt to hide the toolbars for all windows when it is non-zero.

Returns:

The HideToolbars function returns 1 on success and 0 on failure.

Description:

The HideToolbars function tells VisIt to hide the toolbars for the active visualization window or for all visualization windows when the optional allWindows argument is provided and is set to a non-zero value.

Example:

% visit -cli

```
SetWindowLayout(4)
HideToolbars()
ShowToolbars()
# Hide the toolbars for all windows.
HideToolbars(1)
```

`IconifyAllWindows`—Minimizes all of the visualization windows.

Synopsis:

```
IconifyAllWindows()
```

Returns:

The IconifyAllWindows function does not return a value.

Description:

The IconifyAllWindows function minimizes all of the hidden visualization windows to get them out of the way.

Example:

% visit -cli

```
SetWindowLayout(4) # Have 4 windows
IconifyAllWindows()
DeIconifyAllWindows()
```

`InvertBackgroundColor`—Swaps the background and foreground colors in the active visualization window.

Synopsis:

```
InvertBackgroundColor()
```

Returns:

The InvertBackgroundColor function does not return a value.

Description:

The InvertBackgroundColor function swaps the background and foreground colors in the active visualization window. This function is a cheap alternative to setting the foreground and background colors though the AnnotationAttributes in that it is a simple no-argument function call. It is not adequate to set new colors for the background and foreground, but in the event where the two colors can be exchanged favorably, it is a good function to use. An example of when this function is used is after the creation of a Volume plot.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Volume", "u")
DrawPlots()

InvertBackgroundColor()
```

`Lineout`—Performs a lineout.

Synopsis:

```
Lineout(start, end) -> integer
Lineout(start, end, variables) -> integer
Lineout(start, end, samples) -> integer
Lineout(start, end, variables, samples) -> integer
```

Arguments:

`start`	A 2 or 3 item tuple containing the coordinates of the starting point.
`end`	A 2 or 3 item tuple containing the coordinates of the end point.
`variables`	A tuple of strings containing the names of the variables for which lineouts should be created.
`samples`	An integer value containing the number of sample points along the lineout.

Returns:

The Lineout function returns 1 on success and 0 on failure.

Description:

The Lineout function extracts data along a given line segment and creates curves from it in a new visualization window. The start argument is a tuple of numbers that make up the coordinate of the lineout's starting location. The end argument is a tuple of numbers that make up the coordinate of the lineout's ending location. The optional variables argument is a tuple of strings that contain the variables that should be sampled to create lineouts. The optional samples argument is used to determine the number of sample points that should be taken along the specified line. If the samples argument is not provided then VisIt will sample the mesh where it intersects the specified line instead of using the number of samples to compute a list of points to sample.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/rect2d.silo")
AddPlot("Pseudocolor", "ascii")
DrawPlots()

Lineout((0.2,0.2), (0.8,1.2))
Lineout((0.2,1.2), (0.8,0.2), ("default", "d", "u"))
Lineout((0.6, 0.1), (0.6, 1.2), 100)
```

`List`—Lists the members of a SIL restriction category.

Synopsis:

```
ListDomains()
ListMaterials()
```

Returns:

The List functions do not return a value.

Description:

The List functions: ListDomains, and List Materials prints a list of the domains and the materials for the selected plots, which indicates which domains or materials are on and off. The list functions are used mostly to print the results of restricting the SIL.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()
TurnMaterialsOff("4") # Turn off material 4
ListMaterials() # List the materials in the SIL restriction
```

`ListPlots`—Lists the plots in the active visualization window’s plot list.

Synopsis:

```
ListPlots()
```

Returns:

The ListPlots function does not return a value.

Description:

Sometimes it is difficult to remember the order of the plots in the active visualization window’s plot list. The ListPlots function prints the contents of the plot list to the output console.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv2d.silo”)
AddPlot(“Pseudocolor”, “u”)
AddPlot(“Contour”, “d”)
DrawPlots()
ListPlots()
```

`LongFileName`—Returns the long filename for a short WIN32 filename.

Synopsis:

```
LongFileName(filename) -> string
```

Arguments:

`filename`	A string object containing the short filename to expand.

Returns:

The LongFileName function returns a string.

Notes:

This function returns the input argument unless you are on the Windows platform.

Description:

On Windows, filenames can have two different sizes: traditional 8.3 format, and long format. The long format, which lets you name files whatever you want, is implemented using the traditional 8.3 format under the covers. Sometimes filenames are given to VisIt in the traditional 8.3 format and must be expanded to long format before it is possible to open them. If you ever find that you need to do this conversion, such as when you process command line arguments, then you can use the LongFileName function to return the longer filename.

`MovePlotDatabaseKeyframe`—Moves a database keyframe for a plot.

Synopsis:

`MovePlotDatabaseKeyframe(index, oldFrame, newFrame)`

Arguments:

`index`	An integer representing the index of the plof in the plot list.
`oldFrame`	The old animation frame where the keyframe is located.
`newFrame`	The new animation frame where the keyframe will be moved.

Returns:

MovePlotDatabaseKeyframe does not return a value.

Description:

MovePlotDatabaseKeyframe moves a database keyframe for a specified plot to a new animation frame, which changes the list of database time states that are used for each animation frame when VisIt is in keyframing mode.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
AddPlot(“Pseudocolor”, “pressure”)
SetPlotFrameRange(0, 0, nFrames-1)
SetPlotDatabaseKeyframe(0, 0, 70)
SetPlotDatabaseKeyframe(0, nFrames/2, 35)
SetPlotDatabaseKeyframe(0, nFrames-1, 0)
DrawPlots()

for state in list(range(TimeSliderGetNStates())) + [0]:
   SetTimeSliderState(state)

MovePlotDatabaseKeyframe(0, nFrames/2, nFrames/4)

for state in list(range(TimeSliderGetNStates())) + [0]:
   SetTimeSliderState(state)
```

`MovePlotKeyframe`—Moves a keyframe for a plot.

Synopsis:

```
MovePlotKeyframe(index, oldFrame, newFrame)
```

Arguments:

`index`	An integer representing the index of the plof in the plot list.
`oldFrame`	The old animation frame where the keyframe is located.
`newFrame`	The new animation frame where the keyframe will be moved.

Returns:

MovePlotKeyframe does not return a value.

Description:

MovePlotKeyframe moves a keyframe for a specified plot to a new animation frame, which changes the plot attributes that are used for each animation frame when VisIt is in keyframing mode.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Contour”, “hgslice”)
DrawPlots()

k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
SetPlotFrameRange(0, 0, nFrames-1)

c = ContourAttributes()
c.contourNLevels = 5
SetPlotOptions(c)

SetTimeSliderState(nFrames/2)
c.contourNLevels = 10
SetPlotOptions(c)

c.contourLevels = 25
SetTimeSliderState(nFrames-1)
SetPlotOptions(c)

for state in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
   SaveWindow()

temp = nFrames-2
```

```
MovePlotKeyframe(0, nFrames/2, temp)
MovePlotKeyframe(0, nFrames-1, nFrames/2)
MovePlotKeyframe(0, temp, nFrames-1)

for state in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
   SaveWindow()
```

`MoveViewKeyframe`—Moves a view keyframe.

Synopsis:

`MoveViewKeyframe(oldFrame, newFrame)`

Arguments:

`oldFrame`	The old animation frame where the keyframe is located.
`newFrame`	The new animation frame where the keyframe will be moved.

Returns:

MoveViewKeyframe returns 1 on success and 0 on failure.

Description:

MoveViewKeyframe moves a view keyframe to a new animation frame, which changes the view that is used for each animation frame when VisIt is in keyframing mode.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Contour”, “hardyglobal”)
DrawPlots()

k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
SetViewKeyframe()

SetTimeSliderState(nFrames/2)
v = GetView3d()
v.viewNormal = (-0.616518, 0.676972, 0.402014)
v.viewUp = (0.49808, 0.730785, -0.466764)
SetViewKeyframe()

SetTimeSliderState(0)
# Move the view keyframe to the last animation frame.
MoveViewKeyframe(nFrames/2, nFrames-1)
```

`NodePick`—Performs a nodal pick on a plot.

Synopsis:

```
NodePick(point) -> integer
NodePick(point, variables) -> integer
NodePick(sx, sy) -> integer
NodePick(sx, sy, variables) -> integer
```

Arguments:

`point`	A tuple of values that describe the coordinate of where we want to perform the nodal pick.
`variables`	An optional tuple of strings containing the names of the variables for which we want information. The tuple can contain the name “default” if you want information for the plotted variable.
`sx`	A screen X location (in pixels) offset from the left side of the visualization window.
`sy`	A screen Y location (in pixels) offset from the bottom of the visualization window.

Returns:

The NodePick function prints pick information for the node closest to the specified point. The point can be specified as a 2D or 3D point in world space or it can be specified as a pixel location in screen space. If the point is specified as a pixel location then VisIt finds the node closest to a ray that is projected into the mesh. Once the nodal pick has been calculated, you can use the GetPick-Output function to retrieve the printed pick output as a string which can be used for other purposes.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hgslice”)
DrawPlots()

# Perform node pick in screen space
NodePick(300,300)

# Perform node pick in world space.
NodePick((-5.0, 5.0))
```

NumColorTableNames—Returns the number of color tables that have been defined.

Synopsis:

```
NumColorTableNames() -> integer
```

Returns:

The NumColorTableNames function return an integer.

Description:

The NumColorTableNames function returns the number of color tables that have been defined.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
p = PseudocolorAttributes()
p.colorTableName = “default”
SetPlotOptions(p)
DrawPlots()

print “There are %d color tables.” % NumColorTableNames()
for ct in ColorTableNames():
   SetActiveContinuousColorTable(ct)
   SaveWindow()
```

`NumOperatorPlugins`—Returns the number of available operator plugins.

Synopsis:

```
NumOperatorPlugins() -> integer
```

Returns:

The NumOperatorPlugins function returns an integer.

Description:

The NumOperatorPlugins function returns the number of available operator plugins.

Example:

% visit -cli

```
print “The number of operator plugins is: “, NumOperatorPlugins()
print “The names of the plugins are: “, OperatorPlugins()
```

`NumPlotPlugins`—Returns the number of available plot plugins.

Synopsis:

```
NumPlotPlugins() -> integer
```

Returns:

The NumPlotPlugins function returns an integer.

Description:

The NumPlotPlugins function returns the number of available plot plugins.

Example:

% visit -cli

```
print “The number of plot plugins is: “, NumPlotPlugins()
print “The names of the plugins are: “, PlotPlugins()
```

`OpenComputeEngine`—Opens a compute engine on the specified computer

Synopsis:

```
OpenComputeEngine() -> integer
OpenComputeEngine(hostName) -> integer
OpenComputeEngine(hostName, simulation) -> integer
OpenComputeEngine(hostName, args) -> integer
```

Arguments:

`hostName`	The name of the computer on which to start the engine.
`args`	Optional tuple of command line arguments for the engine.

Returns:

The OpenComputeEngine function returns an integer value of 1 for success and 0 for failure.

Description:

The OpenComputeEngine function is used to explicitly open a compute engine with certain properties. When a compute engine is opened implicitly, the viewer relies on sets of attributes called host profiles. Host profiles determine how compute engines are launched. This allows compute engines to be easily launched in parallel. Since the VisIt Python Interface does not expose VisIt's host profiles, it provides the OpenComputeEngine function to allow users to launch compute engines. The OpenComputeEngine function must be called before opening a database in order to prevent any latent host profiles from taking precedence. A complete list of VisIt's command line arguments is listed in **Appendix A** on page 259.

Example:

% visit -cli

```
# Launch parallel compute engine remotely.
args = (“-np”, “16”, “-nn”, “4”)
OpenComputeEngine("thunder", args)

OpenDatabase("thunder:/usr/gapps/visit/data/multi_ucd3d.silo")
AddPlot("Pseudocolor", "d")
DrawPlots()
```

`OpenDatabase`—Opens a database for plotting.

Synopsis:

```
OpenDatabase(databaseName) -> integer
OpenDatabase(databaseName, timeIndex) -> integer
```

Arguments:

`databaseName`	A string containing the name of the database to open.
`timeIndex`	This is an optional integer argument indicating the time index at which to open the database. If it is not specified, a time index of zero is assumed.

Returns:

The OpenDatabase function returns an integer value of 1 for success and 0 for failure.

Description:

The OpenDatabase function is one of the most important functions in the VisIt Python Interface because it opens a database so it can be plotted. The databaseName argument is a string containing the full name of the database to be opened. The database name is of the form: computer:/path/filename. The computer part of the filename can be omitted if the database to be opened resides on the local computer.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
OpenDatabase("mcr:/usr/gapps/visit/data/multi_ucd3d.silo")
OpenDatabase("file.visit")
OpenDatabase("file.visit", 4)
```

`OpenMDServer`—Explicitly opens a metadata server.

Synopsis:

```
OpenMDServer() -> integer
OpenMDServer(host) -> integer
OpenMDServer(host, args) -> integer
```

Arguments:

`host`	The optional host argument determines the host on which the metadata server is to be launched. If this argument is not provided, “localhost” is assumed.
`args`	A tuple of strings containing command line flags for the metadata server.

Returns:

The OpenMDServer function returns 1 on success and 0 on failure.

Description:

The OpenMDServer explicitly launches a metadata server on a specified host. This allows you to provide command line options that influence how the metadata server will run.

Argument	Description
-debug #	The -debug argument allows you to specify a debug level in the range [1,5] that VisIt uses to write debug logs to disk.
-dir visitdir	The -dir argument allows you to specify where VisIt is located on a remote computer. This allows you to successfully connect to a remote computer in the absence of host profiles. It also allows you to debug VisIt in distributed mode.
-default_format format	The -default_format argument allows you to specify the default database plugin that will be used to open files. This is useful when the files that you want to open do not have file extensions. Example: *-default_format PDB*

Example:

% visit -cli

```
args = (“-dir”, “/my/private/visit/version/”, “-default_format”, \
“PDB”, “-debug”, “4”)
# Open a metadata server before the call to OpenDatabase so we
# can launch it how we want.
OpenMDServer(“thunder”, args)
OpenDatabase(“thunder:/usr/gapps/visit/data/allinone00.pdb”)
# Open a metadata server on localhost too.
OpenMDServer()
```

`OperatorPlugins`—Returns a tuple of operator plugin names.

Synopsis:

```
OperatorPlugins() -> tuple of strings
```

Returns:

The OperatorPlugins function returns a tuple of strings.

Description:

The OperatorPlugins function returns a tuple of strings that contain the names of the loaded operator plugins. This can be useful for the creation of scripts that alter their behavior based on the available operator plugins.

Example:

% visit -cli

```
for plugin in OperatorPlugins():
   print “The %s operator plugin is loaded.” % plugin
```

`OverlayDatabase`—Creates new plots based on current plots but with a new database.

Synopsis:

```
OverlayDatabase(databaseName) -> integer
```

Arguments:

`databaseName`	A string containing the name of the new plot database.

Returns:

The OverlayDatabase function returns an integer value of 1 for success and 0 for failure.

Description:

VisIt has the concept of overlaying plots which, in the nutshell, means that the entire plot list is copied and a new set of plots with exactly the same attributes but a different database is appended to the plot list of the active window. The OverlayDatabase function allows the VisIt Python Interface to overlay plots. OverlayDatabase takes a single string argument which contains the name of the database. After calling the OverlayDatabase function, the plot list is larger and contains plots of the specified overlay database.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()

OverlayDatabase("riptide:/usr/gapps/visit/data/curv3d.silo")
```

Pick—Performs a zonal pick on a plot.

Synopsis:

```
Pick(point) -> integer
Pick(point, variables) -> integer
Pick(sx, sy) -> integer
Pick(sx, sy, variables) -> integer

ZonePick(point) -> integer
ZonePick(point, variables) -> integer
ZonePick(sx, sy) -> integer
ZonePick(sx, sy, variables) -> integer
```

Arguments:

`point`	A tuple of values that describe the coordinate of where we want to perform the zonal pick.
`variables`	An optional tuple of strings containing the names of the variables for which we want information. The tuple can contain the name “default” if you want information for the plotted variable.
`sx`	A screen X location (in pixels) offset from the left side of the visualization window.
`sy`	A screen Y location (in pixels) offset from the bottom of the visualization window.

Returns:

The Pick function prints pick information for the cell (a.k.a zone) that contains the specified point. The point can be specified as a 2D or 3D point in world space or it can be specified as a pixel location in screen space. If the point is specified as a pixel location then VisIt finds the zone that contains the intersection of a cell and a ray that is projected into the mesh. Once the zonal pick has been calculated, you can use the GetPickOutput function to retrieve the printed pick output as a string which can be used for other purposes.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hgslice”)
DrawPlots()

# Perform node pick in screen space
Pick(300,300)
# Perform node pick in world space.
Pick((-5.0, 5.0))
```

`PickByGlobalNode`—Performs pick operation for a specific node using a global node id.

Synopsis:

```
PickByGlobalNode(node) -> integer
PickByGlobalNode(node, variables) -> integer
```

Arguments:

`node`	Integer index of the global node for which we want pick information.
`variables`	A tuple of strings containing the names of the variables for which we want pick information.

Returns:

PickByGlobalNode returns 1 on success and 0 on failure.

Description:

The PickByGlobalNode function tells VisIt to perform pick using a specific global node index for the entire problem. Some meshes are broken up into smaller "domains" and then these smaller domains can employ a global indexing scheme to make it appear as though the mesh was still one large mesh. Not all meshes that have been decomposed into domains provide sufficient information to allow global node indexing. You can use the GetPickOutput function to retrieve a string containing the pick information once you've called PickByGlobalNode.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/multi_curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

# Pick on node 200 in the first domain.
PickByGlobalNode(200)
print “Last pick = “, GetPickOutput()
```

`PickByGlobalZone`—Performs pick operation for a specific cell using a global cell id.

Synopsis:

```
PickByGlobalZone(id) -> integer
PickByGlobalZone(id, variables) -> integer
```

Arguments:

`id`	Integer index of the global cell for which we want pick information.
`variables`	A tuple of strings containing the names of the variables for which we want pick information.

Returns:

PickByGlobalZone returns 1 on success and 0 on failure.

Description:

The PickByGlobalZone function tells VisIt to perform pick using a specific global cell index for the entire problem. Some meshes are broken up into smaller “domains” and then these smaller domains can employ a global indexing scheme to make it appear as though the mesh was still one large mesh. Not all meshes that have been decomposed into domains provide sufficient information to allow global cell indexing. You can use the GetPickOutput function to retrieve a string containing the pick information once you’ve called PickByGlobalZone.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/multi_curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

# Pick on cell 200 in the first domain.
PickByGlobalZone(200)
print “Last pick = “, GetPickOutput()
```

`PickByNode`—Performs pick operation for a specific node in a given domain.

Synopsis:

```
PickByNode(node) -> integer
PickByNode(node, variables) -> integer
PickByNode(node, domain) -> integer
PickByNode(node, domain, variables) -> integer
```

Arguments:

`node`	Integer index of the node for which we want pick information.
`domain`	An integer representing the index of the domain that contains the node for which we want pick information. Note that if the first domain is “domain1” then the first valid index is 1.
`variables`	A tuple of strings containing the names of the variables for which we want pick information.

Returns:

PickByNode returns 1 on success and 0 on failure.

Description:

The PickByNode function tells VisIt to perform pick using a specific node index in a given domain. Other pick by node variants first determine the node that is closest to some user-specified 3D point but the PickByNode functions cuts out this step and allows you to directly pick on the node of your choice. You can use the GetPickOutput function to retrieve a string containing the pick information once you’ve called PickByNode.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/multi_curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

# Pick on node 200 in the first domain.
PickByNode(200)
# Pick on node 200 in the second domain.
PickByNode(200, 2)
# Pick on node 100 in domain 5 and return information for two
additional variables.
PickByNode(100, 5, (“default”, “u”, “v”))
print “Last pick = “, GetPickOutput()
```

PickByZone—Performs pick operation for a specific cell in a given domain.

Synopsis:

```
PickByZone(cell) -> integer
PickByZone(cell, variables) -> integer
PickByZone(cell, domain) -> integer
PickByZone(cell, domain, variables) -> integer
```

Arguments:

`cell`	Integer index of the cell for which we want pick information.
`domain`	An integer representing the index of the domain that contains the cell for which we want pick information. Note that if the first domain is “domain1” then the first valid index is 1.
`variables`	A tuple of strings containing the names of the variables for which we want pick information.

Returns:

PickByZone returns 1 on success and 0 on failure.

Description:

The PickByZone function tells VisIt to perform pick using a specific cell index in a given domain. Other pick by zone variants first determine the cell that contains some user-specified 3D point but the PickByZone functions cuts out this step and allows you to directly pick on the cell of your choice. You can use the GetPickOutput function to retrieve a string containing the pick information once you’ve called PickByZone.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/multi_curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

# Pick on cell 200 in the first domain.
PickByZone(200)
# Pick on cell 200 in the second domain.
PickByZone(200, 2)
# Pick on cell 100 in domain 5 and return information for two
additional variables.
PickByZone(100, 5, (“default”, “u”, “v”))
print “Last pick = “, GetPickOutput()
```

`PlotPlugins`—Returns a tuple of plot plugin names.

Synopsis:

```
PlotPlugins() -> tuple of strings
```

Returns:

The PlotPlugins function returns a tuple of strings.

Description:

The PlotPlugins function returns a tuple of strings that contain the names of the loaded plot plugins. This can be useful for the creation of scripts that alter their behavior based on the available plot plugins.

Example:

% visit -cli

```
for plugin in PluginPlugins():
   print “The %s plot plugin is loaded.” % plugin
```

`PrintWindow`—Prints the active visualization window.

Synopsis:

```
PrintWindow() -> integer
```

Returns:

The PrintWindow function returns an integer value of 1 for success and 0 for failure.

Description:

The PrintWindow function tells the viewer to print the image in the active visualization window using the current printer settings.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
AddPlot(“Contour”, “u”)
DrawPlots()
PrintWindow()
```

`PromoteOperator`—Moves an operator closer to the end of the visualization pipeline.

Synopsis:

```
PromoteOperator(opIndex) -> integer
PromoteOperator(opIndex, applyToAllPlots) -> integer
```

Arguments:

`opIndex`	A zero-based integer corresponding to the operator that should be promoted.
`applyAll`	An integer flag that causes all plots in the plot list to be affected when it is non-zero.

Returns:

PromoteOperator returns 1 on success and 0 on failure.

Description:

The PromoteOperator function moves an operator closer to the end of the visualization pipeline. This allows you to change the order of operators that have been applied to a plot without having to remove them from the plot. For example, consider moving a Slice to after a Reflect operator when it had been the other way around. Changing the order of operators can result in vastly different results for a plot. The opposite function is DemoteOperator (see page 63).

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hardyglobal")
AddOperator(“Slice”)
s = SliceAttributes()
s.project2d = 0
s.originPoint = (0,5,0)
s.originType=s.Point
s.normal = (0,1,0)
s.upAxis = (-1,0,0)
SetOperatorOptions(s)
AddOperator("Reflect")
DrawPlots()

# Now slice after reflect. We’ll only get 1 slice plane instead of 2.
PromoteOperator(0)
DrawPlots()
```

`Queries`—Returns a tuple containing the names of all supported queries.

Synopsis:

```
Queries() -> tuple of strings
```

Returns:

The Queries function returns a tuple of strings.

Description:

The Queries function returns a tuple of strings that contain the names of all of VisIt’s supported queries.

Example:

% visit -cli

```
print “supported queries: “, Queries()
```

`QueriesOverTime`—Returns a tuple of containing the names of all supported queries that can execute over time.

Synopsis:

```
QueriesOverTime() -> tuple of strings
```

Returns:

Returns a tuple of strings.

Description:

The QueriesOverTime function returns a tuple of strings that contains the names of all of the VisIt queries that can be executed over time.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/allineone00.pdb”)
AddPlot(“Pseudocolor”, “mesh/mixvar”)
DrawPlots()

# Execute each of the queries over time on the plots.
for q in QueriesOverTime():
   QueryOverTime(q)
```

`Query`—Executes one of VisIt’s queries.

Synopsis:

```
Query(name) -> integer
Query(name, variables) -> integer
Query(name, arg1) -> integer
Query(name, arg1, variables) -> integer
Query(name, arg1, arg2) -> integer
Query(name, arg1, arg2, variables) -> integer
```

Arguments:

`name`	A string containing the name of the query to execute.
`variables`	An optional tuple of strings containing the names of additional query variables.
`arg1`	An optional general purpose integer argument.
`arg2`	An optional general purpose integer argument.

Returns:

The Query function returns 1 on success and 0 on failure.

Description:

The Query function is used to execute any of VisIt’s predefined queries. Since queries can take a wide array of arguments, the Query function has a wide variety of possible arguments. The list of queries can be found in the *VisIt User’s Manual* in the `Quantitative Analysis` chapter.You can get also get a list of queries that can be executed over time using the Queries function.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
AddPlot(“Pseudocolor”, “pressure”)
DrawPlots()

Query(“Volume”)
```

`QueryOverTime`—Executes one of VisIt’s queries over time to produce a curve.

Synopsis:

```
QueryOverTime(name) -> integer
QueryOverTime(name, variables) -> integer
QueryOverTime(name, arg1) -> integer
QueryOverTime(name, arg1, variables) -> integer
QueryOverTime(name, arg1, arg2) -> integer
QueryOverTime(name, arg1, arg2, variables) -> integer
```

Arguments:

`name`	A string containing the name of the query to execute.
`variables`	An optional tuple of strings containing the names of additional query variables.
`arg1`	An optional general purpose integer argument.
`arg2`	An optional general purpose integer argument.

Returns:

The QueryOverTime function returns 1 on success and 0 on failure.

Description:

The QueryOverTime function is used to execute any of VisIt’s predefined queries. Since queries can take a wide array of arguments, the QueryOverTime function has a wide variety of possible arguments. The list of queries can be found in the *VisIt User’s Manual* in the `Quantitative Analysis` chapter. You can get also get a list of queries that can be executed over time using the QueriesOverTime function.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
AddPlot(“Pseudocolor”, “pressure”)
DrawPlots()

for q in QueriesOverTime():
   QueryOverTime(q)

ResetView()
```

RecenterView—Recalculates the view for the active visualization window so that its plots are centered in the window.

Synopsis:

```
RecenterView()
```

Returns:

The RecenterView function does not return a value.

Description:

After adding plots to a visualization window or applying operators to those plots, it is sometimes necessary to recenter the view. When the view is recentered, the orientation does not change but the view is shifted to make better use of the screen.

Example:

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()

OpenDatabase("/usr/gapps/visit/data/curv3d.silo")
AddPlot(“Pseudocolor”, “d”)
DrawPlots()
RecenterView()
```

`RedrawWindow`—Enables redraws and forces the active visualization window to redraw.

Synopsis:

```
RedrawWindow()
```

Returns:

The RedrawWindow function returns 1 on success and 0 on failure.

Description:

The RedrawWindow function allows a visualization window to redraw itself and then forces the window to redraw. This function does the opposite of the DisableRedraw function and is used to recover from it.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Contour”, “u”)
AddPlot(“Pseudocolor”, “w”)
DrawPlots()
DisableRedraw()
AddOperator(“Slice”)
# Set the slice operator attributes

# Redraw now that the operator attributes are set. This will
# prevent 1 redraw.
RedrawWindow()
```

`RemoveOperator`—Removes operators from plots.

Synopsis:

```
RemoveAllOperators() -> integer
RemoveAllOperators(all) -> integer
RemoveLastOperator() -> integer
RemoveLastOperator(all) -> integer
RemoveOperator(index) -> integer
RemoveOperator(index, all) -> integer
```

Arguments:

`all`	An optional integer argument that tells the function to ignore the selected plots and use all plots in the plot list if the value of the argument is non-zero.
`index`	The zero-based integer index into a plot's operator list that specifies which operator is to be deleted.

Returns:

All functions return an integer value of 1 for success and 0 for failure.

Description:

The RemoveOperator functions allow operators to be removed from plots. The RemoveLastOperator function removes the operator that was last applied to the selected plots. The RemoveAllOperators function removes all operators from the selected plots in the active visualization window. If the all argument is provided and contains a non-zero value, all plots in the active visualization window are affected. If the value is zero or if the argument is not provided, only the selected plots are affected.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddOperator("Threshold")
AddOperator("Slice")
AddOperator("SphereSlice")
DrawPlots()

RemoveLastOperator() # Remove SphereSlice
RemoveOperator(0) # Remove Threshold
RemoveAllOperators() # Remove the rest of the operators
```

`ReOpenDatabase`—Reopens a database for plotting.

Synopsis:

```
ReOpenDatabase(databaseName) -> integer
```

Arguments:

`databaseName` A string containing the name of the database to open.

Returns:

The ReOpenDatabase function returns an integer value of 1 for success and 0 for failure.

Description:

The ReOpenDatabase function reopens a database that has been opened previously with the OpenDatabase function. The ReOpenDatabase function is primarily used for regenerating plots whose database has been rewritten on disk. ReOpenDatabase allows VisIt to access new variables and new time states that have been added since the database was opened using the OpenDatabase function. Note that ReOpenDatabase is expensive since it causes all plots that use the specified database to be regenerated. If you want to ensure that a time-varying database has all of its time states as they are being created by a simulation, try the CheckForNewStates function on page 38 instead.

The databaseName argument is a string containing the full name of the database to be opened. The database name is of the form: host:/path/filename. The host part of the filename can be omitted if the database to be reopened resides on the local computer.

Example:

% visit -cli

```
OpenDatabase("frost:/usr/gapps/visit/data/wave*.silo database")
AddPlot("Pseudocolor", "pressure")
DrawPlots()

last = TimeSliderGetNStates()
for state in range(last):
   SetTimeSliderState(state)
   SaveWindow()

ReOpenDatabase("frost:/usr/gapps/visit/data/wave*.silo database")
for state in range(last, TimeSliderGetNStates()):
   SetTimeSliderState(state)
   SaveWindow()
```

`ReplaceDatabase`—Replaces the database in the current plots with a new database.

Synopsis:

```
ReplaceDatabase(databaseName) -> integer
ReplaceDatabase(databaseName, timeState) -> integer
```

Arguments:

`databaseName`	A string containing the name of the new database.
`timeState`	A zero-based integer containing the time state that should be made active once the database has been replaced.

Returns:

The ReplaceDatabase function returns an integer value of 1 for success and 0 for failure.

Description:

The ReplaceDatabase function replaces the database in the current plots with a new database. This is one way of switching timesteps if no ".visit" file was ever created. If two databases have the same variable name then replace is usually a success. In the case where the new database does not have the desired variable, the plot with the variable not contained in the new database does not get regenerated with the new database.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo)
AddPlot("Pseudocolor", "u")
DrawPlots()

ReplaceDatabase("/usr/gapps/visit/data/curv3d.silo")
SaveWindow()

# Replace with a time-varying database and change the time
# state to 17.
ReplaceDatabase("/usr/gapps/visit/data/wave.visit", 17)
```

`ResetLineoutColor`—Resets the color used by lineout to the first color.

Synopsis:

```
ResetLineoutColor() -> integer
```

Returns:

ResetLineoutColor returns 1 on success and 0 on failure.

Description:

Lineouts on VisIt cause reference lines to be drawn over the plot where the lineout was being extracted. Each reference line uses a different color in a discrete color table. Once the colors in the discrete color table are used up, the reference lines start using the color from the start of the discrete color table and so on. ResetLineoutColor forces reference lines to start using the color at the start of the discrete color table again thus resetting the lineout color.

ResetOperatorOptions—Resets operator attributes back to the default values.

Synopsis:

```
ResetOperatorOptions(operatorType) -> integer
ResetOperatorOptions(operatorType, all) -> integer
```

Arguments:

`operatorType`	A string containing the name of a valid operator type.
`all`	An optional integer argument that tells the function to reset the operator options for all plots regardless of whether or not they are selected.

Returns:

The ResetOperatorOptions function returns an integer value of 1 for success and 0 for failure.

Description:

The ResetOperatorOptions function resets the operator attributes of the specified operator type for the selected plots back to the default values. The operatorType argument is a string containing the name of the type of operator whose attributes are to be reset. The all argument is an optional flag that tells the function to reset the operator attributes for the indicated operator in all plots regardless of whether the plots are selected. When non-zero values are passed for the all argument, all plots are reset. When the all argument is zero or not provided, only the operators on selected plots are modified.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

AddOperator(“Slice”)
a = SliceAttributes()
a.normal,a.upAxis = (0,0,1),(0,1,0)
SetOperatorOptions(a)
ResetOperatorOptions(“Slice”)
```

`ResetPickLetter`—Resets the pick marker letter back to “A”.

Synopsis:

```
ResetPickLetter() -> integer
```

Returns:

ResetPickLetter returns 1 on success and 0 on failure.

Description:

The ResetPickLetter function resets the pick marker back to “A” so that the next pick will use “A” as the pick letter and then “B” and so on.

`ResetPlotOptions`—Resets plot attributes back to the default values.

Synopsis:

```
ResetPlotOptions(plotType) -> integer
```

Arguments:

`plotType`	A string containing the name of the plot type.

Returns:

The ResetPlotOptions function returns an integer value of 1 for success and 0 for failure.

Description:

The ResetPlotOptions function resets the plot attributes of the specified plot type for the selected plots back to the default values. The plotType argument is a string containing the name of the type of plot whose attributes are to be reset.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

p = PseudocolorAttributes()
p.colorTableName = "calewhite"
p.minFlag,p.maxFlag = 1,1
p.min,p.max = -5.0, 8.0
SetPlotOptions(p)
ResetPlotOptions(“Pseudocolor”)
```

`ResetView`—Resets the view to the initial view

Synopsis:

```
ResetView()
```

Returns:

The ResetView function does not return a value.

Description:

The ResetView function resets the camera to the initial view.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv3d.silo”)
AddPlot(“Mesh”, “curvmesh3d”)
v = ViewAttributes()
v.camera = (-0.45396, 0.401908, 0.79523)
v.focus = (0, 2.5, 15)
v.viewUp = (0.109387, 0.910879, -0.397913)
v.viewAngle = 30
v.setScale = 1
v.parallelScale = 16.0078
v.nearPlane = -32.0156
v.farPlane = 32.0156
v.perspective = 1
SetView3D(v) # Set the 3D view
DrawPlots()
ResetView()
```

RestoreSession—Restores a VisIt session.

Synopsis:

```
RestoreSession(filename, visitDir) -> integer
```

Arguments:

`filename`	The name of the session file to restore.
`visitDir`	An integer flag that indicates whether the filename to be restored is located in the user’s VisIt directory. If the flag is set to 1 then the session file is assumed to be located in the user’s VisIt directory otherwise the filename must contain an absolute path.

Returns:

RestoreSession returns 1 on success and 0 on failure.

Description:

The RestoreSession function is important for setting up complex visualizations because you can design a VisIt session file, which is an XML file that describes exactly how plots are set up, using the VisIt GUI and then use that same session file in the CLI to generate movies in batch. The RestoreSession function takes 2 arguments. The first argument specifies the filename that contains the VisIt session to be restored. The second argument determines whether the session file is assumed to be in the user’s VisIt directory. If the visitDir argument is set to 0 then the filename argument must contain the absolute path to the session file.

Example:

% visit -cli

```
# Restore my session file for a time-varying database from
# my .visit directory.
RestoreSessionFile(“visit.session”, 1)
for state in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
   SaveWindow()
```

SaveSession—Tells VisIt to save a session file describing the current visualization.

Synopsis:

```
SaveSession(filename) -> integer
```

Arguments:

`filename`	The filename argument is the filename that is used to save the session file. The filename is relative to the current working directory.

Returns:

The SaveSession function returns 1 on success and 0 on failure.

Description:

The SaveSession function tells VisIt to save an XML session file that describes everything about the current visualization. Session files are very useful for creating movies and also as shortcuts for setting up complex visualizations.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
# Set up a keyframe animation of view and save a session file of it.
k = GetKeyframeAttributes()
k.enabled,k.nFrames,k.nFramesWasUserSet = 1,20,1
SetKeyframeAttributes(k)
AddPlot(“Surface”, “hgslice”)
DrawPlots()

v = GetView3D()
v.viewNormal = (0.40823, -0.826468, 0.387684)
v.viewUp, v.imageZoom = (-0.261942, 0.300775, 0.917017), 1.60684
SetView3D(v)
SetViewKeyframe()

SetTimeSliderState(TimeSliderGetNStates() - 1)
v.viewNormal = (-0.291901, -0.435608, 0.851492)
v.viewUp = (0.516969, 0.677156, 0.523644)
SetView3D(v)
SetViewKeyframe()

ToggleCameraViewMode()
SaveSession(“~/.visit/keyframe.session”)
```

SaveWindow—Save the contents of the active window

Synopsis:

```
SaveWindow() -> string
```

Returns:

The SaveWindow function returns a string containing the name of the file that was saved.

Description:

The SaveWindow function saves the contents of the active visualization window. The format of the saved window is dictated by the SaveWindowAttributes which can be set using the SetSaveWindowAttributes function. The contents of the active visualization window can be saved as TIFF, JPEG, RGB, PPM, PNG images or they can be saved as curve, Alias Wavefront Obj, or VTK geometry files.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv3d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

# Set the save window attributes.
s = SaveWindowAttributes()
s.filename = "test"
s.format = s.FILEFORMAT_JPEG
s.progressive = 1
s.fileName = "test"
SetSaveWindowAttributes(s)
name = SaveWindow()

print “name = %s” % name
```

`SetActiveColorTable`—Sets the color table that is used by plots that use a “default” color table.

Synopsis:

```
SetActiveContinuousColorTable(name) -> integer
SetActiveDiscreteColorTable(name) -> integer
```

Arguments:

`name`	The name of the color table to use for the active color table. The name must be present in the tuple returned by the ColorTableNames function.

Returns:

Both functions return 1 on success and 0 on failure.

Description:

VisIt supports two flavors of color tables: continuous and discrete. Both types of color tables have the same underlying representation but each type of color table is used a slightly different way. Continuous color tables are made of a small number of color control points and the gaps in the color table between two color control points are filled by interpolating the colors of the color control points. Discrete color tables do not use any kind of interpolation and like continuous color tables, they are made up of control points. The color control points in a discrete color table repeat infinitely such that if we have 4 color control points: A, B, C, D then the pattern of repetition is: ABCDABCDABCD... Discrete color tables are mainly used for plots that have a discrete set of items to display (e.g. Subset plot). Continuous color tables are used in plots that display a continuous range of values (e.g. Pseudocolor).

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Contour”, “hgslice”)
DrawPlots()
SetActiveDiscreteColorTable(“levels”)
```

`SetActivePlots`—Sets the active plots in the plot list.

Synopsis:

```
SetActivePlots(plots) -> integer
```

Arguments:

`plots`	A tuple of integer plot indices starting at zero.

Returns:

The SetActivePlots function returns an integer value of 1 for success and 0 for failure.

Description:

Any time VisIt sets the attributes for a plot, it only sets the attributes for plots which are selected. The SetActivePlots function must be called to set the active plots. The function takes one argument which is a tuple of integer plot indices that start at zero. If only one plot is being selected, the plots argument can be an integer instead of a tuple.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Subset”, “mat1”)
AddPlot(“Mesh”, “mesh1”)
AddPlot(“Contour”, “u”)
DrawPlots()

SetActivePlots((0,1,2)) # Make all plots active
SetActivePlots(0) # Make only the Subset plot active
```

`SetActiveTimeSlider`—Sets the active time slider.

Synopsis:

```
SetActiveTimeSlider(tsName) -> integer
```

Arguments:

`tsName`	A string containing the name of the time slider that should be made active.

Returns:

SetActiveTimeSlider returns 1 on success and 0 on failure.

Description:

Sets the active time slider, which is the time slider that is used to change time states.

Example:

% visit -cli

```
path = “/usr/gapps/visit/data/”
dbs = (path + “dbA00.pdb”, path + “dbB00.pdb”, path + “dbC00.pdb”)
for db in dbs:
OpenDatabase(db)
AddPlot(“FilledBoundary”, “material(mesh)”)
DrawPlots()
CreateDatabaseCorrelation(“common”, dbs, 1)
tsNames = GetWindowInformation().timeSliders
for ts in tsNames:
   SetActiveTimeSlider(ts)
   for state in list(range(TimeSliderGetNStates())) + [0]:
      SetTimeSliderState(state)
```

`SetActiveWindow`—Sets the active visualization window.

Synopsis:

```
SetActiveWindow(windowIndex) -> integer
```

Arguments:

`windowIndex`	An integer window index starting at 1.

Returns:

The SetActiveWindow fucntion returns an integer value of 1 for success and 0 for failure.

Description:

Most of the functions in the VisIt Python Interface operate on the contents of the active window. If there is more than one window, it is very important to be able to set the active window. To set the active window, use the SetActiveWindow function. The SetActiveWindow function takes a single integer argument which is the index of the new active window. The new window index must be an integer greater than zero and less than or equal to the number of open windows.

Example:

% visit -cli

```
SetWindowLayout(2)
SetActiveWindow(2)
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Mesh”, “mesh1”)
DrawPlots()
```

`SetAnimationTimeout`—Sets the speed at which animations play.

Synopsis:

```
SetAnimationTimeout(milliseconds) -> integer
```

Returns:

The SetAnimationTimeout function returns 1 for success and 0 for failure.

Description:

The SetAnimationTimeout function sets the animation timeout which is a value that governs how fast animations play. The timeout is specified in milliseconds and has a default value of 1 millisecond. Larger timeout values decrease the speed at which animations play.

Example:

%visit -cli

```
# Play a new frame every 5 seconds.
SetAnimationTimeout(5000)

OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
AddPlot(“Pseudocolor”, “pressure”)
DrawPlots()

for state in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
```

`SetAnnotationAttributes`—Sets the annotation attributes for the active window.

Synopsis:

```
SetAnnotationAttributes(atts)
SetDefaultAnnotationAttributes(atts)
```

Arguments:

`atts`	An AnnotationAttributes object containing the annotation settings.

Returns:

Both functions return 1 on success and 0 on failure.

Description:

The annotation settings control what bits of text are drawn in the visualization window. Among the annotations are the plot legends, database information, user information, plot axes, triad, and the background style and colors. Setting the annotation attributes is important for producing quality visualizations. The annotation settings are stored in AnnotationAttributes objects. To set the annotation attributes, first create an AnnotationAttributes object using the AnnotationAttributes function and then pass the object to the SetAnnotationAttributes function. To set the default annotation attributes, also pass the object to the SetDefaultAnnotationAttributes function.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/wave.visit")
AddPlot("Pseudocolor", "pressure")
DrawPlots()

a = AnnotationAttributes()
a.axes3DFlag = 1
a.userInfoFlag = 0
a.gradientBackgroundStyle = a.GRADIENTSTYLE_RADIAL
a.gradientColor1 = (0,255,255)
a.gradientColor2 = (0,0,0)
a.backgroundMode = a.BACKGROUNDMODE_GRADIENT
SetAnnotationAttributes(a)
```

`SetCenterOfRotation`—Sets the center of rotation for plots in a 3D vis window.

Synopsis:

```
SetCenterOfRotation(x,y,z) -> integer
```

Arguments:

`x`	The x component of the center of rotation.
`y`	The y component of the center of rotation.
`z`	The z component of the center of rotation.

Returns:

The SetCenterOfRotation function returns 1 on success and 0 on failure.

Description:

The SetCenterOfRotation function sets the center of rotation for plots in a 3D visualization window. The center of rotation, is the point about which plots are rotated when you interactively spin the plots using the mouse. It is useful to set the center of rotation if you've zoomed in on any 3D plots so in the event that you rotate the plots, the point of interest remains fixed on the screen.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
AddPlot("Mesh", "mesh1")
DrawPlots()

v = GetView3D()
v.viewNormal = (-0.409139, 0.631025, 0.6591)
v.viewUp = (0.320232, 0.775678, -0.543851)
v.imageZoom = 4.8006
SetCenterOfRotation(-4.755280, 6.545080, 5.877850)

# Rotate the plots interactively.
```

`SetDatabaseCorrelationOptions`—Sets the global options for database correlations.

Synopsis:

```
SetDatabaseCorrelationOptions(method, whenToCreate) -> integer
```

Arguments:

`method`	An integer that tells VisIt what default method to use when automatically creating a database correlation. The value must be in the range [0,3].
`whenToCreate`	An integer that tells VisIt when to automatically create database correlations.

Returns:

SetDatabaseCorrelationOptions returns 1 on success and 0 on failure.

Description:

VisIt provides functions to explicitly create and alter database correlations but there are also a number of occasions where VisIt can automatically create a database correlation. The SetDatabaseCorrelationOptions function allows you to tell VisIt the default correlation method to use when automatically creating a new database correlation and it also allows you to tell VisIt when database correlations can be automatically created.

method	Description
0	IndexForIndexCorrelation
1	StretchedIndexCorrelation
2	TimeCorrelation
3	CycleCorrelation

whenToCreate	Description
0	Always create database correlation
1	Never create database correlation
2	Create database correlation only if the new time-varying database has the same length as another time-varying database already being used in a plot.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/dbA00.pdb”)
AddPlot(“FilledBoundary”, “material(mesh)”)
DrawPlots()

# Always create a stretched index correlation.
SetDatabaseCorrelationOptions(1, 0)
OpenDatabase(“/usr/gapps/visit/data/dbB00.pdb”)
AddPlot(“FilledBoundary”, “material(mesh)”)
# The AddPlot caused a database correlation to be created.
DrawPlots()

wi = GetWindowInformation()
print “Active time slider: “ % wi.timeSliders[wi.activeTimeSlider]
# This will set time for both databases since the database correlation
is the active time slider.
SetTimeSliderState(5)
```

`SetGlobalLineoutAttributes`—Sets global lineout attributes that are used for all lineouts.

Synopsis:

```
SetGlobalLineoutAttributes(atts) -> integer
```

Arguments:

`atts`	A GlobalLineoutAttributes object that contains the new settings.

Returns:

The SetGlobalLineoutAttributes function returns 1 on success and 0 on failure.

Description:

The SetGlobalLineoutAttributes function allows you to set global lineout options that are used in the creation of all lineouts. You can, for example, specify the destination window and the number of sample points for lineouts.

Example:

% visit -cli

```
SetWindowLayout(4)
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Pseudocolor", "hgslice")
DrawPlots()

gla = GetGlobalLineoutAttributes()
gla.createWindow = 0
gla.windowId = 4
gla.samplingOn = 1
gla.numSamples = 150
SetGlobalLineoutAttributes(gla)

Lineout((-5,-8), (-3.5, 8))
```

`SetInteractorAttributes`—Sets VisIt’s interactor attributes.

Synopsis:

```
SetInteractorAttributes(atts) -> integer
SetDefaultInteractorAttributes(atts) -> integer
```

Arguments:

`atts`	An InteractorAttributes object that contains the new interactor attributes that you want to use.

Returns:

SetInteractorAttributes returns 1 on success and 0 on failure.

Description:

The SetInteractorAttributes function is used to set certain interactor properties. Interactors, can be thought of as how mouse clicks and movements are translated into actions in the vis window. To set the interactor attributes, first get the interactor attributes using the GetInteractorAttributes function (see page 83). Once you’ve set the object’s properties, call the SetInteractorAttributes function to make VisIt use the new interactor attributes.

The SetDefaultInteractorAttributes function sets the default interactor attributes, which are used for new visualization windows. The default interactor attributes can also be saved to the VisIt configuration file to ensure that future VisIt sessions have the right default interactor attributes.

Example:

% visit -cli

```
ia = GetInteractorAttributes()
print ia
ia.showGuidelines = 0
SetInteractorAttributes(ia)
```

`SetKeyframeAttributes`—Sets VisIt’s keyframing attributes.

Synopsis:

```
SetKeyframeAttributes(kfAtts) -> integer
```

Arguments:

`kfAtts`	A KeyframeAttributes object that contains the new keyframing attributes to use.

Returns:

SetKeyframeAttributes returns 1 on success and 0 on failure.

Description:

Use the SetKeyframeAttributes function when you want to change VisIt’s keyframing settings. You must pass a KeyframeAttributes object, which you can create using the GetKeyframeAttributes function (page 84). The KeyframeAttributes object must contain the keyframing settings that you want VisIt to use. For example, you would use the SetKeyframeAttributes function if you wanted to turn on keyframing mode and set the number of animation frames.

Example:

% visit -cli

```
k = GetKeyframeAttributes()
print k
k.enabled,k.nFrames,k.nFramesWasUserSet = 1, 100, 1
SetKeyframeAttributes(k)
```

`SetLight`—Returns a light object containing the attributes for a specified light.

Synopsis:

```
SetLight(index, light) -> integer
```

Arguments:

`index`	A zero-based integer index into the light list. Index can be in the range [0,7].
`light`	A LightAttributes object containing the properties to use for the specified light.

Returns:

SetLight returns 1 on success and 0 on failure.

Description:

The SetLight function sets the attributes for a specific light.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot("Pseudocolor", "w")
p = PseudocolorAttributes()
p.colorTableName = “xray”
SetPlotOptions(p)
DrawPlots()

InvertBackgroundColor()

light = GetLight(0)
print light

light.enabledFlag = 1
light.direction = (0,-1,0)
light.color = (255,0,0,255)
SetLight(0, light)

light.color,light.direction = (0,255,0,255), (-1,0,0)
SetLight(1, light)
```

`SetMaterialAttributes`—Sets VisIts material interface reconstruction options.

Synopsis:

```
SetMaterialAttributes(atts) -> integer
SetDefaultMaterialAttributes(atts) -> integer
```

Arguments:

`atts`	A MaterialAttributes object containing the new settings.

Returns:

Both functions return 1 on success and 0 on failure.

Description:

The SetMaterialAttributes function takes a MaterialAttributes object and makes VisIt use the material settings that it contains. You use the SetMaterialAttributes function when you want to change how VisIt performs material interface reconstruction. The SetDefaultMaterialAttributes function sets the default material attributes, which are saved to the config file and are also used by new visualization windows.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/allinone00.pdb”)
AddPlot(“Pseudocolor”, “mesh/mixvar”)
p = PseudocolorAttributes()
p.min,p.minFlag = 4.0, 1
p.max,p.maxFlag = 13.0, 1
SetPlotOptions(p)
DrawPlots()

# Tell VisIt to always do material interface reconstruction.
m = GetMaterialAttributes()
m.forceMIR = 1
SetMaterialAttributes(m)
ClearWindow()

# Redraw the plot forcing VisIt to use the mixed variable information.
DrawPlots()
```

`SetOperatorOptions`—Sets the attributes for an operator.

Synopsis:

```
SetOperatorOptions(atts) -> integer
SetOperatorOptions(atts, activeIndex, all) -> integer
SetOperatorOptions(atts, all) -> integer
SetDefaultOperatorOptions(atts) -> integer
```

Arguments:

`atts`	Any type of operator attributes object.
`activeIndex`	An optional zero-based integer that serves as an index into the selected plot's operator list. Use this argument if you want to set the operator attributes for a plot that has multiple instances of the same type of operator.
`all`	An optional integer argument that tells the function to apply the operator attributes to all plots containing the specified operator if the value of the argument is non-zero.

Returns:

All functions return an integer value of 1 for success and 0 for failure.

Description:

Each operator in VisIt has a group of attributes that controls the operator. To set the attributes for an operator, first create an operator attributes object. This is done by calling a function which is the name of the operator plus the word "Attributes". For example, a Slice operator's operator attributes object is created and returned by the SliceAttributes function. Assign the new operator attributes object into a variable and set its fields. After setting the desired fields in the operator attributes object, pass the object to the SetOperatorOptions function. The SetOperatorOptions function determines the type of operator to which the operator attributes object applies and sets the attributes for that operator type. To set the default plot attributes, use the SetDefaultOperatorOptions function. Setting the default attributes ensures that all future instances of a certain operator are initialized with the new default values.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
AddPlot("Mesh", "mesh1")
AddOperator("Slice", 1) # Add the operator to both plots
a = SliceAttributes()
a.normal, a.upAxis = (0,0,1), (0,1,0)
# Only set the attributes for the selected plot.
SetOperatorOptions(a)
DrawPlots()
```

SetPickAttributes—Changes the pick settings that VisIt uses when picking on plots.

Synopsis:

```
SetPickAttributes(atts) -> integer
SetDefaultPickAttributes(atts) -> integer
ResetPickAttributes() -> integer
```

Arguments:

`atts`	A PickAttributes object containing the new pick settings.

Returns:

All functions return 1 on success and 0 on failure.

Description:

The SetPickAttributes function changes the pick attributes that are used when VisIt picks on plots. The pick attributes allow you to format your pick output in various ways and also allows you to select auxiliary pick variables.

Example:

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hgslice”)
DrawPlots()

Pick((-5,5))

p = GetPickAttributes()
p.showTimeStep = 0
p.showMeshName = 0
p.showZoneId = 0
SetPickAttributes(p)
Pick((0,5))
```

`SetPipelineCachingMode`—Sets the pipeline caching mode.

Synopsis:

```
SetPipelineCachingMode(mode) -> integer
```

Returns:

The SetPipelineCachingMode function returns 1 for success and 0 for failure.

Description:

The SetPipelineCachingMode function turns pipeline caching on or off in the viewer. When pipeline caching is enabled, animation timesteps are cached for fast playback. This can be a disadvantage for large databases or for plots with many timesteps because it increases memory consumption. In those cases, it is often useful to disable pipeline caching so the viewer does not use as much memory. When the viewer does not cache pipelines, each plot for a timestep must be recalculated each time the timestep is visited.

Example:

% visit -cli

```
SetPipelineCachingMode(0) # Disable caching
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
AddPlot(“Pseudocolor”, “pressure”)
AddPlot(“Mesh”, “quadmesh”)
DrawPlots()
for state in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
```

SetPlotDatabaseState—Sets a database keyframe for a specific plot.

Synopsis:

```
SetPlotDatabaseState(index, frame, state)
```

Arguments:

`index`	A zero-based integer index that is the plot’s location in the plot list.
`frame`	A zero-baed integer index representing the animation frame for which we’re going to add a database keyframe.
`state`	A zero-based integer index representating the database time state that we’re going to use at the specified animation frame.

Returns:

The SetPlotDatabaseState function does not return a value.

Description:

The SetPlotDatabaseState function is used when VisIt is in keyframing mode to add a database keyframe for a specific plot. VisIt uses database keyframes to determine which database state is to be used for a given animation frame. Database keyframes can be used to stop “database time” while “animation time” continues forward and they can also be used to make “database time” go in reverse, etc.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
AddPlot(“Pseudocolor”, “pressure”)
AddPlot(“Mesh”, “quadmesh”)
DrawPlots()

# Make “database time” for the Pseudocolor plot go in reverse
SetPlotDatabaseState(0, 0, 70)
SetPlotDatabaseState(0, nFrames-1, 0)

# Animate through the animation frames since the “Keyframe animation”
time slider is active.
for state in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
```

`SetPlotFrameRange`—Sets the range of animation frames over which a plot is valid.

Synopsis:

```
SetPlotFrameRange(index, start, end)
```

Arguments:

`index`	A zero-based integer representing an index into the plot list.
`start`	A zero-based integer representing the animation frame where the plot first appears in the visualization.
`end`	A zero-based integer representing the animation frame where the plot disappears from the visualization.

Returns:

The SetPlotFrameRange function does not return a value.

Description:

The SetPlotFrameRange function sets the start and end frames for a plot when VisIt is in keyframing mode. Outside of this frame range, the plot does not appear in the visualization. By default, plots are valid over the entire range of animation frames when they are first created. Frame ranges allow you to construct complex animations where plots appear and disappear dynamically.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
AddPlot(“Pseudocolor”, “pressure”)
AddPlot(“Mesh”, “quadmesh”)
DrawPlots()

# Make the Pseudocolor plot take up the first half of the animation
frames before it disappears.
SetPlotFrameRange(0, 0, nFrames/2-1)

# Make the Mesh plot take up the second half of the animation frames.
SetPlotFrameRange(1, nFrames/2, nFrames-1)
for state in range(TimeSliderGetNStates())
   SetTimeSliderState(state)
   SaveWindow()
```

`SetPlotOptions`—Sets plot attributes for the selected plots.

Synopsis:

```
SetPlotOptions(atts) -> integer
SetDefaultPlotOptions(atts) -> integer
```

Arguments:

`atts`	Any type of plot attributes object.

Returns:

All functions return an integer value of 1 for success and 0 for failure.

Description:

Each plot in VisIt has a group of attributes that controls the appearance of the plot. To set the attributes for a plot, first create a plot attributes object. This is done by calling a function which is the name of the plot plus the word “Attributes”. For example, a Pseudocolor plot’s plotattributes object is created and returned by the PseudocolorAttributes function. Assign the new plot attributes object into a variable and set its fields. After setting the desired fields in the plot attributes object, pass the object to the SetPlotOptions function. The SetPlotOptions function determines the type of plot to which the plot attributes object applies and sets the attributes for that plot type. To set the default plot attributes, use the SetDefaultPlotOptions function. Setting the default attributes ensures that all future instances of a certain plot are initialized with the new default values.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
p = PseudocolorAttributes()
p.colorTableName = "calewhite"
p.minFlag,p.maxFlag = 1,1
p.min,p.max = -5.0, 8.0
SetPlotOptions(p)
DrawPlots()
```

`SetPlotSILRestriction`—Set the SIL restriction for the selected plots.

Synopsis:

```
SetPlotSILRestriction(silr) -> integer
SetPlotSILRestriction(silr, all) -> integer
```

Arguments:

`silr`	A SIL restriction object.
`all`	An optional argument that tells the function if the SIL restriction should be applied to all plots in the plot list.

Returns:

The SetPlotSILRestriction function returns an integer value of 1 for success and 0 for failure.

Description:

VisIt allows the user to select subsets of databases. The description of the subset is called a Subset Inclusion Lattice Restriction, or SIL restriction. The SIL restriction allows databases to be subselected in several different ways. The VisIt Python Interface provides the SetPlotSILRestriction function to allow Python scripts to turn off portions of the plotted database. The SetPlotSILRestriction function accepts a SILRestriction object that contains the SIL restriction for the selected plots. The optional all argument is an integer that tells the function to apply the SIL restriction to all plots when the value of the argument is non-zero. If the all argument is not supplied, then the SIL restriction is only applied to the selected plots.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/multi_curv2d.silo”)
AddPlot("Subset", "mat1")
silr = SILRestriction()
silr.TurnOffSet(silr.SetsInCategory(’mat1’)[1])
SetPlotSILRestriction(silr)
DrawPlots()
```

SetPrinterAttributes—Sets the printer attributes.

Synopsis:

```
SetPrinterAttributes(atts)
```

Arguments:

`atts`	A PrinterAttributes object.

Returns:

The SetPrinterAttributes function does not return a value.

Description:

The SetPrinterAttributes function sets the printer attributes. VisIt uses the printer attributes to determine how the active visualization window should be printed. The function accepts a single argument which is a PrinterAttributes object containing the printer attributes to use for future printing. VisIt allows images to be printed to a network printer or to a PostScript file that can be printed later by other applications.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/curv2d.silo")
AddPlot("Surface", "v")
DrawPlots()

# Make it print to a file.
p = PrinterAttributes()
p.outputToFile = 1
p.outputToFileName = "printfile"
SetPrinterAttributes(p)
PrintWindow()
```

SetQueryOverTimeAttributes—Changes the settings that VisIt uses for queries over time.

Synopsis:

```
SetQueryOverTimeAttributes(atts) -> integer
SetDefaultQueryOverTimeAttributes(atts) -> integer
ResetQueryOverTimeAttributes() -> integer
```

Arguments:

`atts`	A QueryOverTimeAttributes object containing the new settings to use for queries over time.

Returns:

All functions return 1 on success and 0 on failure.

Description:

The SetQueryOverTimeAttributes function changes the settings that VisIt uses for query over time. The SetDefaultQueryOverTimeAttributes function changes the settings that new visualization windows inherit for doing query over time. Finally, the ResetQueryOverTimeAttributes function forces VisIt to use the stored default query over time attributes instead of the previous settings.

Example:

% visit -cli

```
SetWindowLayout(4)
OpenDatabase(“/usr/gapps/visit/data/allinone00.pdb”)
AddPlot(“Pseudocolor”, “mesh/mixvar”)
DrawPlots()

qot = GetQueryOverTimeAttributes()
# Make queries over time go to window 4.
qot.createWindow,q.windowId = 0, 4
SetQueryOverTimeAttributes(qot)
QueryOverTime(“Min”)

# Make queries over time only use half of the number of time states.
qot.endTimeFlag,qot.endTime = 1, GetDatabaseNStates() / 2
SetQueryOverTimeAttributes(qot)
QueryOverTime(“Min”)
ResetView()
```

`SetRenderingAttributes`—Sets global rendering attributes that control the look and feel of the plots.

Synopsis:

```
SetRenderingAttributes(atts) -> integer
```

Arguments:

`atts`	A RenderingAttributes object that contains the rendering attributes that we want to make VisIt use.

Returns:

The SetRenderingAttributes function returns 1 on success and 0 on failure.

Description:

The SetRenderingAttributes makes VisIt use the rendering attributes stored in the specified RenderingAttributes object. The RenderingAttributes object stores rendering attributes such as: scalable rendering options, shadows, specular highlights, display lists, etc.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Surface”, “hgslice”)
DrawPlots()

v = GetView2D()
v.viewNormal = (-0.215934, -0.454611, 0.864119)
v.viewUp = (0.973938, -0.163188, 0.157523)
v.imageZoom = 1.64765
SetView3D(v)

light = GetLight(0)
light.direction = (0,1,-1)
SetLight(0, light)

r = GetRenderingAttributes()
print r
r.scalableActivationMode = r.Always
r.doShadowing = 1
SetRenderingAttributes(r)
```

`SetSaveWindowAttributes`—Set the attributes used to save windows.

Synopsis:

```
SetSaveWindowAttributes(atts)
```

Arguments:

`atts`	A SaveWindowAttributes object.

Returns:

The SetSaveWindowAttributes object does not return a value.

Description:

The SetSaveWindowAttributes function sets the format and filename that are used to save windows when the SaveWindow function is called. The contents of the active visualization window can be saved as TIFF, JPEG, RGB, PPM, PNG images or they can be saved as curve, Alias Wavefront Obj, or VTK geometry files. To set the SaveWindowAttributes, create a SaveWindowAttributes object using the SaveWindowAttributes function and assign it into a variable. Set the fields in the object and pass it to the SetSaveWindowAttributes function.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv3d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

# Set the save window attributes
s = SaveWindowAttributes()
s.filename = "test"
s.format = s.FILEFORMAT_JPEG
s.progressive = 1
s.fileName = "test"
SetSaveWindowAttributes(s)

# Save the window
SaveWindow()
```

`SetTimeSliderState`—Sets the time state for the active time slider.

Synopsis:

```
SetTimeSliderState(state) -> integer
```

Arguments:

`state`	A zero-based integer containing the time state that we want to make active.

Returns:

The SetTimeSliderState function returns 1 on success and 0 on failure.

Description:

The SetTimeSliderState function sets the time state for the active time slider. This is the function to use if you want to animate through time or change the current keyframe frame.

Example:

% visit -cli

```
path = “/usr/gapps/visit/data/”
dbs = (path + “dbA00.pdb”, path + “dbB00.pdb”, path + “dbC00.pdb”)
for db in dbs:
   OpenDatabase(db)
   AddPlot(“FilledBoundary”, “material(mesh)”)
DrawPlots()

CreateDatabaseCorrelation(“common”, dbs, 1)

tsNames = GetWindowInformation().timeSliders
for ts in tsNames:
   SetActiveTimeSlider(ts)
   for state in list(range(TimeSliderGetNStates())) + [0]:
      SetTimeSliderState(state)
```

`SetView`—Sets the view for the active visualization window.

Synopsis:

```
SetView2D(view)
SetView3D(view)
```

Arguments:

`view`	A ViewAttributes object containing the view.

Returns:

Neither function returns a value.

Description:

The view is a crucial part of a visualization since it determines which parts of the database are examined. The VisIt Python Interface provides two functions for setting the view: SetView2D, and SetView3D. If the visualization window contains 2D plots, use the SetView2D function. Use the SetView3D function when the visualization window contains 3D plots. Both functions take a ViewAttributes object as an argument. To set the view, first create a ViewAttributes object using the ViewAttributes function and set the object's fields to set a new view. After setting the fields, pass the object to either the SetView2D or SetView3D function. A common use of the SetView functions is to animate the view to produce simple animations where the camera appears to fly around the plots in the visualization window.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “v”)
DrawPlots()

v0 = GetView3D()
v1 = GetView3D()
v1.camera,v1.viewUp = (1,1,1),(-1,1,-1)
v1.parallelScale = 10.

for i in range(0,20):
   t = float(i) / 19.
   v2 = (1. - t) * v0 + t * v1
   SetView3D(v2) # Animate the view.
```

SetViewExtentsType—Tells VisIt how to use extents when computing the view.

Synopsis:

```
SetViewExtentsType(type) -> integer
```

Arguments:

`type`	An integer 0, 1 or one of the strings: “original”, “actual”.

Returns:

SetViewExtentsType returns 1 on success and 0 on failure.

Description:

VisIt can use a plot’s spatial extents in two ways when computing the view. The first way of using the extents is to use the “original” extents, which are the spatial extents before any modifications, such as subset selection, have been made to the plot. This ensures that the view will remain relatively constant for a plot. Alternatively, you can use the “actual” extents, which are the spatial extents of the pieces of the plot that remain after operations such as subset selection.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
SetViewExtentsType(“actual”)
AddPlot(“FilledBoundary”, “mat1”)
DrawPlots()

v = GetView3D()
v.viewNormal = (-0.618945, 0.450655, 0.643286)
v.viewUp = (0.276106, 0.891586, -0.358943)
SetView3D(v)

mats = GetMaterials()
nmats = len(mats):

# Turn off all but the last material in sequence and watch
# the view update each time.
for i in range(nmats-1):
   index = nmats-1-i
   TurnMaterialsOff(mats[index])
   SaveWindow()

SetViewExtentsType(“original”)
```

SetViewKeyframe—Adds a view keyframe.

Synopsis:

```
SetViewKeyframe() -> integer
```

Returns:

The SetViewKeyframe function returns 1 on success and 0 on failure.

Description:

The SetViewKeyframe function adds a view keyframe when VisIt is in keyframing mode. View keyframes are used to set the view at crucial points during an animation. Frames that lie between view keyframes have an interpolated view that is based on the view keyframes. You can use the SetViewKeyframe function to create complex camera animations that allow you to fly around (or through) your visualization.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/noise.silo")
AddPlot("Contour", "hardyglobal")
DrawPlots()

k = GetKeyframeAttributes()
nFrames = 20
k.enabled, k.nFrames, k.nFramesWasUserSet = 1, nFrames, 1
SetKeyframeAttributes(k)
SetPlotFrameRange(0, 0, nFrames-1)
SetViewKeyframe()

SetTimeSliderState(10)
v = GetView3D()
v.viewNormal = (-0.721721, 0.40829, 0.558944)
v.viewUp = (0.294696, 0.911913, -0.285604)
SetView3D(v)
SetViewKeyframe()

SetTimeSliderState(nFrames-1)
v.viewNormal = (-0.74872, 0.423588, -0.509894)
v.viewUp = (0.369095, 0.905328, 0.210117)
SetView3D()
SetViewKeyframe()

ToggleCameraViewMode()
for state in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
   SaveWindow()
```

SetWindowArea—Set the screen area devoted to visualization windows.

Synopsis:

```
SetWindowArea(x, y, width, height)
```

Arguments:

`x`	Left X coordinate in screen pixels.
`y`	Top Y coordinate in screen pixels.
`width`	Width of the window area in pixels.
`height`	Height of the window area in pixels.

Returns:

The SetWindowArea function does not return a value.

Description:

The SetWindowArea method sets the area of the screen that can be used by VisIt’s visualization windows. This is useful for making sure windows are a certain size when running a Python script.

Example:

```
import visit
visit.Launch()
visit.SetWindowArea(0, 0, 600, 600)
visit.SetWindowLayout(4)
```

`SetWindowLayout`—Sets the window layout

Synopsis:

```
SetWindowLayout(layout) -> integer
```

Arguments:

`layout`	An integer that specifies the window layout. (1,2,4,8,9,16 are valid)

Returns:

The SetWindowLayout function returns an integer value of 1 for success and 0 for failure.

Description:

VisIt’s visualization windows can be arranged in various tiled patterns that allow VisIt to make good use of the screen while displaying several visualization windows. The window layout determines how windows are shown on the screen. The SetWindowLayout function sets the window layout. The layout argument is an integer value equal to 1,2,4,8,9, or 16.

Example:

% visit -cli

```
SetWindowLayout(2) # switch to 1x2 layout
SetWindowLayout(4) # switch to 2x2 layout
SetWindowLayout(8) # switch to 2x4 layout
```

SetWindowMode—Sets the window mode of the active visualization window.

Synopsis:

```
SetWindowMode(mode)
```

Arguments:

`mode`	A string containing “lineout”, “navigate”, “pick”, or “zoom”.

Returns:

The SetWindowMode function does not return a value.

Description:

VisIt’s visualization windows have various window modes that alter their behavior. Most of the time a visualization window is in “navigate” mode which changes the view when the mouse is moved in the window. The “zoom” mode allows a zoom rectangle to be drawn in the window for changing the view. The “pick” mode retrieves information about the plots when the mouse is clicked in the window. The “lineout” mode allows the user to draw lines which produce curve plots.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv2d.silo”)
AddPlot(“Pseudocolor”, “d”)
DrawPlots()

SetWindowMode(“zoom”)
# Draw a rectangle in the visualization window to zoom the plots
```

`ShowAllWindows`—Tells VisIt to show its visualization windows.

Synopsis:

```
ShowAllWindows() -> integer
```

Returns:

The ShowAllWindows function returns 1 on success and 0 on failure.

Description:

The ShowAllWindows function tells VisIt's viewer to show all of its visualization windows. The command line interface calls ShowAllWindows before giving control to any user-supplied script to ensure that the visualization windows appear as expected. Call the ShowAllWindows function when using the VisIt module inside another Python interpreter so the visualization windows are made visible.

Example:

% python

```
import visit
visit.Launch()
visit.ShowAllWindows()
```

`ShowToolbars`—Shows the visualization window’s toolbars.

Synopsis:

```
ShowToolbars() -> integer
ShowToolbars(allWindows) ->integer
```

Arguments:

`allWindows`	An integer value that tells VisIt to show the toolbars for all windows when it is non-zero.

Returns:

The ShowToolbars function returns 1 on success and 0 on failure.

Description:

The ShowToolbars function tells VisIt to show the toolbars for the active visualization window or for all visualization windows when the optional allWindows argument is provided and is set to a non-zero value.

Example:

% visit -cli

```
SetWindowLayout(4)
HideToolbars(1)
ShowToolbars()
# Show the toolbars for all windows.
ShowToolbars(1)
```

`Source`—Executes the specified Python script

Synopsis:

```
Source(filename)
```

Returns:

The Source function does not return a value.

Description:

The Source function reads in the contents of a text file and interprets it with the Python interpreter. This is a simple mechanism that allows simple scripts to be included in larger scripts. The Source function takes a single string argument that contains the name of the script to execute.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "u")
DrawPlots()

# include another script that does some animation.
Source("Animate.py")
```

TimeSliderGetNStates—Returns the number of time states for the active time slider.

Synopsis:

```
TimeSliderGetNStates() -> integer
```

Returns:

Returns an integer containing the number of time states for the current time slider.

Description:

The TimeSliderGetNStates function returns the number of time states for the active time slider. Remember that the length of the time slider does not have to be equal to the number of time states in a time-varying database because of database correlations and keyframing. If you want to iterate through time, use this function to determine the number of iterations that are required to reach the end of the active time slider.

Example:

```
OpenDatabase(“/usr/gapps/visit/data/wave.visit”)
AddPlot(“Pseudocolor”, “pressure”)
DrawPlots()

for state in range(TimeSliderGetNStates()):
   SetTimeSliderState(state)
   SaveWindow()
```

`TimeSliderNextState`—Advances the active time slider to the next state.

Synopsis:

```
TimeSliderNextState() -> integer
```

Returns:

The TimeSliderNextState function returns 1 on success and 0 on failure.

Description:

The TimeSliderNextState function advances the active time slider to the next time slider state.

Example:

# Assume that files are being written to the disk.

% visit -cli

```
OpenDatabase(“dynamic*.silo database”)
AddPlot(“Pseudocolor”, “var”)
AddPlot(“Mesh”, “mesh”)
DrawPlots()
SetTimeSliderState(TimeSliderGetNStates() - 1)
while 1:
   SaveWindow()
   TimeSliderPreviousState()
```

`TimeSliderPrevState`—Moves the active time slider to the previous time state.

Synopsis:

```
TimeSliderPreviousState() -> integer
```

Returns:

The TimeSliderPreviousState function returns 1 on success and 0 on failure.

Description:

The TimeSliderPreviousState function moves the active time slider to the previous time slider state.

Example:

# Assume that files are being written to the disk.

% visit -cli

```
OpenDatabase(“dynamic*.silo database”)
AddPlot(“Pseudocolor”, “var”)
AddPlot(“Mesh”, “mesh”)
DrawPlots()

while 1:
   TimeSliderNextState()
   SaveWindow()
```

ToggleMode—Toggle a visualization window mode

Synopsis:

```
ToggleBoundingBoxMode() -> integer
ToggleCameraViewMode() -> integer
ToggleFullFrameMode() -> integer
ToggleLockTime() -> integer
ToggleLockViewMode() -> integer
ToggleMaintainDataMode() -> integer
ToggleMaintainViewMode() -> integer
ToggleSpinMode() -> integer
```

Returns:

All functions return 1 on success and 0 on failure.

Description:

The visualization window has various modes that affect its behavior and the VisIt Python Interface provides a few functions to toggle some of those modes.

The ToggleBoundingBoxMode function toggles bounding box mode on and off. When the visualization window is in bounding box mode, any plots it contains are hidden while the view is being changed so the window redraws faster.

The ToggleCameraViewMode function toggles camera view mode on and off. When the visualization window is in camera view mode, the view is updated using any view keyframes that have been defined when VisIt is in keyframing mode.

The ToggleFullFrameMode function toggles fullframe mode on and off. When the visualization window is in fullframe mode, the viewport is stretched non-uniformly so that it covers most of the visualization window. While not maintaining a 1:1 aspect ratio, it does make better use of the visualization window.

The ToggleLockTime function turns time locking on and off in a visualization window. When time locking is on in a visualization window, VisIt creates a database correlation that works for the databases in all visualization windows that are time-locked. When you change the time state using the time slider for the the afore-mentioned database correlation, it has the effect of updating time in all time-locked visualization windows.

The ToggleLockViewMode function turns lock view mode on and off. When windows are in lock view mode, each view change is broadcast to other windows that are also in lock view mode. This allows windows containing similar plots to be compared easily.

The ToggleMaintainDataMode and ToggleMaintainViewMode functions force the data range and the view, respectively, that was in effect when the mode was toggled to be used for all subsequent time states.

The ToggleSpinMode function turns spin mode on and off. When the visualization window is in spin mode, it continues to spin along the axis of rotation when the view is changed interactively.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

# Turn on spin mode.
ToggleSpinMode()
# Rotate the plot interactively using the mouse and watch it keep
spinning after the mouse release.
# Turn off spin mode.
ToggleSpinMode()
```

Turn—Turns materials or domains on or off.

Synopsis:

```
TurnMaterialsOn() -> integer
TurnMaterialsOn(string) -> integer
TurnMaterialsOn(tuple of strings) -> integer
TurnMaterialsOff() -> integer
TurnMaterialsOff(string) -> integer
TurnMaterialsOff(tuple of strings) -> integer
TurnDomainsOn() -> integer
TurnDomainsOn(string) -> integer
TurnDomainsOn(tuple of strings) -> integer
TurnDomainsOff() -> integer
TurnDomainsOff(string) -> integer
TurnDomainsOff(tuple of strings) -> integer
```

Returns:

The Turn functions return an integer with a value of 1 for success or 0 for failure.

Description:

The Turn functions are provided to simplify the removal of material or domain subsets. Instead of creating a SILRestriction object, you can use the Turn functions to turn materials or domains on or off. The TurnMaterialsOn function turns materials on and the TurnMaterialsOff function turns them off. The TurnDomainsOn function turns domains on and the TurnDomainsOff function turns them off. All of the Turn functions have three possible argument lists. When you do not provide any arguments, the function applies to all subsets in the SIL so if you called the TurnMaterialsOff function with no arguments, all materials would be turned off. Calling TurnMaterialsOn with no arguments would turn all materials on. All functions can also take a string argument, which is the name of the set to modify. For example, you could turn off domain 0 by calling the TurnDomainsOff with a single argument of “domain0” (or the appropriate set name). All of the Turn functions can also be used to modify more than one set if you provide a tuple of set names. After you use the Turn functions to change the SIL restriction, you might want to call the ListMaterials or ListDomains functions to make sure that the SIL restriction was actually modified.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
DrawPlots()

TurnMaterialsOff(“4”) # Turn off material 4
TurnMaterialsOff((“1”, “2”)) # Turn off materials 1 and 2
TurnMaterialsOn() # Turn on all materials
```

`UndoView`—Restores the previous view

Synopsis:

```
UndoView()
```

Returns:

The UndoView function does not return a value.

Description:

When the view changes in the visualization window, it puts the old view on a stack of views. The UndoView function restores the view on top of the stack and removes it. This allows the user to undo up to ten view changes.

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/curv2d.silo”)
AddPlot(“Subset”, “mat1”)
DrawPlots()
v = GetView2D()
v.windowCoords = (-2.3,2.4,0.2,4.9)
SetView2D(v)
UndoView()
```

`WriteConfigFile`—Tells the viewer to write its configuration file.

Synopsis:

```
WriteConfigFile()
```

Returns:

The WriteconfigFile function does not return a value.

Description:

The viewer maintains internal settings which determine the default values for objects like plots and operators. The viewer can save out the default values so they can be used in future VisIt sessions. The WriteConfig function tells the viewer to write out the settings to the VisIt configuration file.

Example:

% visit -cli

```
p = PseudocolorAttributes()
p.minFlag, p.min = 1, 5.0
p.maxFlag, p.max = 1, 20.0
SetDefaultPlotOptions(p)
# Save the new default Pseudocolor settings to the config file.
WriteConfig()
```

Chapter 5 Object Reference I

1.0 Overview

VisIt's Python Interface is replete with functions that perform complex behaviors. Each function requires anywhere from a handful of pieces of information to dozens of pieces of information in order to complete successfully. To ensure that programming scripts using VisIt's Python Interface is made as simple as possible, many functions take special objects which group all related pieces of information as their arguments. These objects are new Python types that contain groups of attributes that are used to perform related functions. These objects, which are covered in this chapter, are used to simplify the interface to the functions in the VisIt Python Interface and they also make convenient return values for functions that return information about VisIt's state.

2.0 Functions and Objects

The objects described in this reference all have constructor functions which return a new instance of the specific object type. The reference page for each object gives both a synopsis of how to use the constructor function and a description of all of the fields and methods that are part of the object.

Most objects can be modified using a simple assigment of an appropriate Python object into any of its fields. When an incompatible value is assigned into one of an object's fields, the Python interpreter will throw an exception. In addition to being able to set the value of each field directly, all objects provide Set/Get methods which can be used to assign values into a field. These Set/Get methods are not covered except to say that if you have an object called FOO and a field called BAR, the Set/Get methods would be of the form: FOO.SetBAR(value), FOO.GetBAR() -> value.

AnnotationAttributes—AnnotationAttributes function and object.

Synopsis:

AnnotationAttributes() -> AnnotationAttributes object

Returns:

The AnnotationAttributes function returns an AnnotationAttributes object.

Object definition:

Field	Type	Default value
axesFlag2D	int	1
axesAutoSetTicks2D	int	1
labelAutoSetScaling2D	int	1
xAxisLabels2D	int	1
yAxisLabels2D	int	1
xAxisTitle2D	int	1
yAxisTitle2D	int	1
xGridLines2D	int	0
yGridLines2D	int	0
xMajorTickMinimum2D	float	0.0
yMajorTickMinimum2D	float	0.0
xMajorTickMaximum2D	float	1.0
yMajorTickMaximum2D	float	1.0
xMajorTickSpacing2D	float	0.2
yMajorTickSpacing2D	float	0.2
xMinorTickSpacing2D	float	0.02
yMinorTickSpacing2D	float	0.02
xLabelFontHeight2D	float	0.02
yLabelFontHeight2D	float	0.02
xTitleFontHeight2D	float	0.02
yTitleFontHeight2D	float	0.02
xLabelScaling2D	int	0
yLabelScaling2D	int	0

Field	Type	Default value
axesLineWidth2D	int	0
axesTickLocation2D	int	Outside Possible values: Inside, Outside, Both
axesTicks2D	int	BottomLeft Possible values: Off, Bottom, Left, Bot-tomLeft, All
axesFlag	int	1
axesAutoSetTicks	int	1
labelAutoSetScaling	int	1
xAxisLabels	int	1
yAxisLabels	int	1
zAxisLabels	int	1
xAxisTitle	int	1
yAxisTitle	int	1
zAxisTitle	int	1
xGridLines	int	0
yGridLines	int	0
zGridLines	int	0
xAxisTicks	int	1
yAxisTicks	int	1
zAxisTicks	int	1
xMajorTickMinimum	float	0.0
yMajorTickMinimum	float	0.0
zMajorTickMinimum	float	0.0
xMajorTickMaximum	float	1.0
yMajorTickMaximum	float	1.0
zMajorTickMaximum	float	1.0
xMajorTickSpacing	float	0.2
yMajorTickSpacing	float	0.2
zMajorTickSpacing	float	0.2
xMinorTickSpacing	float	0.02
yMinorTickSpacing	float	0.02

Field	Type	Default value
zMinorTickSpacing	float	0.02
xLabelFontHeight	float	0.02
yLabelFontHeight	float	0.02
zLabelFontHeight	float	0.02
xTitleFontHeight	float	0.02
yTitleFontHeight	float	0.02
zTitleFontHeight	float	0.02
xLabelScaling	int	0
yLabelScaling	int	0
zLabelScaling	int	0
axesTickLocation	int	0
axesType	int	ClosestTriad Possible values: ClosestTriad, FurthestTriad, OutsideEdges, StaticTriad, StaticEdges
triadFlag	int	1
bboxFlag	int	1
backgroundColor	tuple of int	(255, 255, 255, 255)
foregroundColor	tuple of int	(0, 0, 0, 255)
gradientBackgroundStyle	int	Radial Possible values: TopToBottom, BottomToTop, LeftToRight, RightToLeft, Radial
gradientColor1	tuple of int	(0, 0, 255, 255)
gradientColor2	tuple of int	(0, 0, 0, 255)
backgroundMode	int	Solid Possible values: Solid, Gradient
userInfoFlag	int	1
databaseInfoFlag	int	1
legendInfoFlag	int	1

Description:

The AnnotationAttributes function is used to create AnnotationAttributes objects that can be passed to the SetAnnotationAttributes function once their attributes have been set.

Example:

% visit -cli

```
# Turning on a gradient background.
OpenDatabase(“/usr/gapps/visit/data/globe.silo”)
AddPlot(“Pseudocolor”, “u”)
AddPlot(“Mesh”, “mesh1”)
DrawPlots()

# Get the current annotation attributes.
a = GetAnnotationAttributes()
a.foregroundColor = (255,255,255,255)
a.gradientBackgroundStyle = a.TopToBottom
a.gradientColor1 = (150,200,255,255)
a.backgroundMode = a.Gradient

# Make the visualization use a gradient background.
SetAnnotationAttributes(a)
```

`GlobalAttributes`—GlobalAttributes function and object.

Synopsis:

GlobalAttributes() -> GlobalAttributes object

Returns:

The GlobalAttributes function returns a GlobalAttributes object.

Notes:

The only thing a GlobalAttributes object is good for is knowing some of VisIt's state information such as how many windows there are, what the active sources are, etc. Many of the fields contain simple global flags that you can query and use in control flow logic in your VisIt script. Think of GlobalAttributes as being read only.

Object definition:

Field	Type	Default value
sources	tuple of string	() Contains the names of all of the open sources (databases, simulations, etc).
windows	tuple of int	(1)
activeWindow	int	0
iconifiedFlag	int	0
autoUpdateFlag	int	0
replacePlots	int	0
applyOperator	int	1
executing	int	0
windowLayout	int	0
makeDefaultConfirm	int	1
cloneWindowOnFirstRef	int	0
maintainView	int	0
maintainData	int	0
automaticallyAddOpera-tor	int	0

Description:

The GlobalAttributes function is used to create GlobalAttributes objects but more often than not, you should use the GetGlobalAttributes function to create a GlobalAttributes object since objects returned from that function will be initialized with VisIt's current state.

Example:

% visit -cli

```
SetWindowLayout(4)
ptypes = ("Pseudocolor", "FilledBoundary", "Vector", "Volume")
pvars = ("u", "mat1", "vel", "speed")
for win in GetGlobalAttributes().windows:
   SetActiveWindow(win)
   DeleteAllPlots()
   OpenDatabase("/usr/gapps/visit/data/globe.silo")
   i = win-1
   AddPlot(ptypes[i], pvars[i])
DrawPlots()
```

LightAttributes—LightAttributes function and object.

Synopsis:

LightAttributes() -> LightAttributes object

Returns:

The LightAttributes function returns a LightAttributes object.

Object definition:

Field	Type	Default value
enabledFlag	int	0
type	int	Camera Possible values: Ambient, Object, Camera
direction	tuple of float	(0.0, 0.0, -1.0)
color	tuple of float	(255, 255, 255, 255)
brightness	float	1.0

Description:

The LightAttributes function is used to create LightAttributes objects. Once you set the properties for a LightAttributes object, you can pass the object to the SetLight function (see page 159) to set the properties for one of VisIt's lights.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/globe.silo")
AddPlot("Pseudocolor", "w")
p = PseudocolorAttributes()
p.colorTableName = "xray"
SetPlotOptions(p)
InvertBackgroundColor()
DrawPlots()

L = LightAttributes()
L.direction, L.color = (0,-1,0), (255,0,0,255)
SetLight(0, L)
```

`MaterialAttributes`—MaterialAttributes function and object.

Synopsis:

MaterialAttributes() -> MaterialAttributes object

Returns:

The MaterialAttributes function returns a MaterialAttributes object.

Object definition:

Field	Type	Default value
cleanZonesOnly	int	0
forceMIR	int	0
needValidConnectivity	int	0
smoothing	int	0
useNewMIRAlgorithm	int	1

Description:

The MaterialAttributes function is used to create MaterialAttributes objects which are then used to set material interface reconstruction options using the SetMaterialAttributes function (see page 160).

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/allinone00.pdb”)
AddPlot(“Pseudocolor”, “mesh/mixvar”)
p = PseudocolorAttributes()
p.min,p.minFlag = 4.0, 1
p.max,p.maxFlag = 13.0, 1
SetPlotOptions(p)
DrawPlots()

# Tell VisIt to always do material interface reconstruction.
m = GetMaterialAttributes()
m.forceMIR = 1
SetMaterialAttributes(m)
ClearWindow()
DrawPlots()
```

PrinterAttributes—PrinterAttributes function and object.

Synopsis:

PrinterAttributes() -> PrinterAttributes object

Returns:

The PrinterAttributes function returns a PrinterAttributes object.

Object definition:

Field	Type	Default value
creator	string	""
documentName	string	"untitled"
numCopies	int	1
outputToFile	int	0
outputToFileName	string	"untitled"
pageSize	int	2
portrait	int	1
printColor	int	1
printProgram	string	“lpr”
printerName	string	““

Description:

The PrinterAttributes function is used to create PrinterAttributes objects.

RenderingAttributes—RenderingAttributes function and object.

Synopsis:

RenderingAttributes() -> RenderingAttributes object

Description:

The RenderingAttributes function returns a RenderingAttributes object.

Object definition:

Field	Type	Default value
antialiasing	int	0
geometryRepresentation	int	Surfaces Possible values: Surfaces, Wireframe, Points
displayListMode	int	Auto Possible values: Never, Always, Auto
stereoRendering	int	0
stereoType	int	CrystalEyes Possible values: RedBlue, Interlaced, CrystalEyes
notifyForEachRender	int	0
scalableActivationMode	int	Auto Possible values: Never, Always, Auto
scalableAutoThreshold	int	2000000
specularFlag	int	0
specularCoeff	float	0.6
specularPower	float	10.0
specularColor	tuple of int	(255, 255, 255, 255)
doShadowing	int	0
shadowStrength	float	0.5

Description:

The RenderingAttributes function is used to create RenderingAttributes objects that can be passed to the SetRenderingAttributes function (see page 170) to set VisIt’s rendering options.

Example:

```
% visit -cli
```

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Surface”, “hgslice”)
DrawPlots()

# Set up the view
v = GetView3D()
v.viewNormal = (-0.215934, -0.454611, 0.864119)
v.viewUp = (0.973938, -0.163188, 0.157523)
v.imageZoom = 1.64765
SetView3D(v)

# Set up the light.
light = GetLight(0)
light.direction = (0,1,-1)
SetLight(0, light)

# Turn on shadows
r = GetRenderingAttributes()
print r
r.scalableActivationMode = r.Always
r.doShadowing = 1
SetRenderingAttributes(r)
```

SaveWindowAttributes—SaveWindowAttributes function and object.

Synopsis:

SaveWindowAttributes() -> SaveWindowAttributes object

Returns:

The SaveWindowAttributes function returns a SaveWindowAttributes object.

Object definition:

Field	Type	Default value
outputToCurrentDirectory	int	1
outputDirectory	string	“.”
fileName	string	"visit"
family	int	1
format	int	TIFF Possible values: BMP, CURVE, JPEG, OBJ, PNG, POSTSCRIPT, PPM, RGB, STL, TIFF, ULTRA, VTK
maintainAspect	int	1
width	int	1024
height	int	1024
screenCapture	int	1
saveTiled	int	0
quality	int	80
progressive	int	0
binary	int	0
stereo	int	0
compression	int	1

Description:

The SaveWindowAttributes function is used to create SaveWindowAttributes objects so that you can pass them to the SetSaveWindowAttributes function to set options that are used when VisIt saves an image.

Example:

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Contour”, “hardyglobal”)
```

```
DrawPlots()

# Set the save window options
s = SaveWindowAttributes()
s.fileName = “mycontour.jpeg”
s.format = s.JPEG
s.family = 0
s.progressive = 1
s.quality = 90
SetSaveWindowOptions(s)

# Save the image in the active vis window.
SaveWindow()
```

SILRestriction—SilRestriction function and object.

Synopsis:

SILRestriction() -> SILRestriction object

Returns:

The SILRestriction function returns a SILRestriction object.

Notes:

The SILRestriction object is used to specify a SIL restriction for a plot. This allows you to turn off any of the subsets that make up a plot. The SILRestriction object is a little different from other objects provided by the VisIt Python Interface in that is relies more on methods than the ability to set properties. This makes it more like an actual Python class. If you want an easier way to turn domains and materials on or off, refer to the functions on page 187. If you have more exotic types of subsets, use the SILRestriction function to create a SIL restriction and then call the SetPlotSIL-Restriction function on page 167 to make the plot use the SIL restriction.

Object definition:

Method	Description
Categories() -> tuple of string	Returns a tuple of strings containing the names of all of the subset categories contained in the SIL.
NumCategories() -> int	Returns an integer containing the number of subset categories in the SIL.
NumSets() -> int	Returns an integer containing the number of sets in the SIL.
SetIndex(setname) -> int *Arguments:* `setname` A string containing the name of the set whose index is to be returned.	The SetIndex method returns the set index of a set that is given by name. This makes it easy to look up a set and turn it on or off if you have the name of the set that you want to change.
SetName(setindex) -> string *Arguments:* `setindex` An integer index into the list of sets.	The SetName method returns the name of a set when given a set index.

Method	Description
SetsInCategory(catname) -> tuple of int *Arguments:* `catname` The name of a category. The name must be in the tuple returned by the Categories method.	This method returns a tuple of integers containing the set indices of the sets that belong in the specified category.
TopSet() -> int	The TopSet method returns the set index of the top set, which is the top-level set that contains all of the sets under it. Note that a SILRestriction can have multiple top sets if there is more than one mesh in the data set.
TurnOffAll()	The TurnOffAll method turns off all sets in the SIL restriction.
TurnOffSet(setindex) *Arguments:* `setindex` An integer index into the list of sets.	The TurnOffSet method turns off a specific set and all of that set's subsets in the SIL restriction.
TurnOnAll()	The TurnOnAll method turns on all sets in the SIL restriction.
TurnOnSet(setindex) *Arguments:* `setindex` An integer index into the list of sets.	The TurnOnSet method turns on a specific set and all of that set's subsets in the SIL restriction.
TurnSet(setindex, onoff) *Arguments:* `setindex` An integer index into the list of sets. `onoff` An integer that indicates whether the set is on (non-zero) or off (zero).	The TurnSet method allows you to turn a specific set and all of its subsets on or off.

Method	Description
UsesAllData() -> int	Returns whether or not the SIL restriction uses all of its sets.
UsesData(setindex) -> int *Arguments:* `setindex` An integer index into the list of sets.	The UsesData method allows you to query the SIL restriction to see whether or not it uses a specific set. If the specified set is used, the method returns 1; otherwise 0 is returned.
Wholes() -> tuple of int	The Wholes method returns a tuple of integers containing the indices of the top sets that are stored in the simulation.

Description:

The SILRestriction function is used to create SILRestriction objects based on the active plot in the plot list. A SIL restriction is can be thought of as a list of on/off values for each set in the SIL. VisIt uses SIL restrictions to determine which subsets are actually plotted in a visualization. You can use a SILRestriction object to turn off sets that you do not want to see in the visualization.

Example:

% visit -cli

```
OpenDatabase("/usr/gapps/visit/data/multi_rect2d.silo")
AddPlot("FilledBoundary", "mat1")
AddPlot("Subset", "domains")
s = SubsetAttributes()
s.wireframe, s.lineWidth = 1, 2
SetPlotOptions(s)
DrawPlots()
ToggleMaintainViewMode()

# Select both plots so we can set both of their SIL restrictions.
SetActivePlots((0,1))

s = SILRestriction()
# Iterate through the categories in the SIL restriction
# and turn all sets on/off different ways.
for category in s.Categories():
   # Turn sets off one at a time.
   for setindex in s.SetsInCategory(category):
      s.TurnOffSet(setindex)
      SetPlotSILRestriction(s)
   # Turn sets back on one at a time.
   for setindex in s.SetsInCategory(category):
      s.TurnOnSet(setindex)
      SetPlotSILRestriction(s)
```

ViewCurveAttributes—ViewCurveAttributes function and object.

Synopsis:

ViewCurveAttributes() -> ViewCurveAttributes object

Returns:

The ViewCurveAttributes function returns a ViewCurveAttributes object.

Object definition:

Field	Type	Default value
domainCoords	tuple of float	(0.0, 1.0)
rangeCoords	tuple of float	(0.0, 1.0)
viewportCoords	tuple of float	(0.2, 0.95)

Description:

The ViewCurveAttributes function is used to create ViewCurveAttributes objects, which are passed to the SetViewCurve function to set the view when the visualization window displays Curve plots.

`View2DAttributes`—View2DAttributes function and object.

Synopsis:

View2DAttributes() -> View2DAttributes object

Returns:

The View2DAttributes function returns a View2DAttributes object.

Object definition:

Field	Type	Default value
fullFrame	int	0
viewportCoords	tuple of float	(0.2, 0.95, 0.15, 0.95)
windowCoords	tuple of float	(0.0, 1.0, 0.0, 1.0)

Description:

The View2DAttributes function is used to create View2DAttributes objects that are used to set the 2D view using the SetView2D function. Note that View2DAttributes can be used in arithmetic expressions so you can add views together, multiply them by a scale factor, etc. You can even interpolate views (e.g. view0 * (1. - t) + view1 * t).

Example:

% visit -cli

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Pseudocolor”, “hgslice”)
DrawPlots()

v = View2DAttributes()
xmin, ymin = 0.5, 0.5
v.viewportCoords = (xmin, 0.95, ymin, 0.95)
v.windowCoords = (-5.0, 0.0, 1.02, 5.9)
SetView2D(v)
```

View3DAttributes—View3DAttributes and function.

Synopsis:

View3DAttributes() -> View3DAttributes object

Returns:

The View3DAttributes function returns a View3DAttributes object.

Object definition:

Field	Type	Default value
viewNormal	tuple of float	(0, 0, 1)
focus	tuple of float	(0, 0, 0)
viewUp	tuple of float	(0, 1, 0)
viewAngle	float	30.0
parallelScale	float	0.5
nearPlane	float	-0.5
farPlane	float	0.5
imagePan	tuple of float	(0.0, 0.0)
imageZoom	float	1.0
perspective	int	1
eyeAngle	float	2.0
centerOfRotationSet	int	0
centerOfRotation	tuple of float	(0.0, 0.0, 0.0)

Description:

The View3DAttributes function is used to create View3DAttributes objects which are then passed to the SetView3D function to set the view when the visualization window is in 3D. Note that View3DAttributes can be used in arithmetic expressions so you can add views together, multiply them by a scale factor, etc. You can even interpolate views (e.g. view0 * (1. - t) + view1 * t).

Example:

```
OpenDatabase(“/usr/gapps/visit/data/noise.silo”)
AddPlot(“Contour”, “hardyglobal”)
DrawPlots()
```

```
# Get an initialized view object.
v = GetView3D()
v. viewNormal = (-0.740314, 0.525704, 0.419012)
v.viewUp = (0.406866, 0.846547, -0.343247)
v.imageZoom = 2.70858
v.centerOfRotationSet = 1
v.centerOfRotation = (-9.57689, 6.25049, 3.18773)
SetView3D(v)
```

`WindowInformation`—WindowInformation function and object.

Synopsis:

WindowInformation() -> WindowInformation object

Returns:

The WindowInformation function returns a WindowInformation object.

Notes:

The WindowInformation object, like the GlobalAttributes object, should be considered to be a read-only object that reflects the state of VisIt. The difference is that the WindowInformation object contains state information for the active visualization window.

Object definition:

Field	Type	Default value
activeSource	string	“notset”
activeTimeSlider	int	-1
timeSliders	tuple of string	()
timeSliderCurrentStates	tuple of int	()
animationMode	int	2
windowMode	int	1
boundingBoxNavigate	int	1
spin	int	0
fullFrame	int	0
perspective	int	1
lockView	int	0
lockTools	int	0
lockTime	int	0
viewExtentsType	int	0
viewDimension	int	2
viewKeyframes	tuple of int	()
cameraViewMode	int	0
usingScalableRendering	int	0
lastRenderMin	float	0.0

Field	Type	Default value
lastRenderAvg	float	0.0
lastRenderMax	float	0.0
numTriangles	int	0
extents	tuple of float	(-FLTMAX,FLTMAX,-FLTMAX,FLT-MAX,-FLTMAX,FLTMAX)

Description:

The WindowInformation function is used to create WindowInformation objects. For an initialized WindowInformation object, use the GetWindowInformation function. WindowInformation objects allow you to inspect some of VIsIt's state for the active window and you can use this state information to influence the control flow of your scripts.

Example 1:

% visit -cli

```
path = “/usr/gapps/visit/data/”
dbs = (path + “dbA00.pdb”, path + “dbB00.pdb”, path + “dbC00.pdb”)
for db in dbs:
   OpenDatabase(db)
   AddPlot(“FilledBoundary”, “material(mesh)”)
DrawPlots()

CreateDatabaseCorrelation(“common”, dbs, 1)

# Use the WindowInformation to get the list of time sliders.
tsNames = GetWindowInformation().timeSliders
for ts in tsNames:
   SetActiveTimeSlider(ts)
   for state in list(range(TimeSliderGetNStates())) + [0]:
      SetTimeSliderState(state)
DeleteAllPlots()
```

Example 2:

```
OpenDatabase(path + “noise.silo”)
AddPlot(“Pseudocolor”, “hardyglobal”)
DrawPlots()

v = GetView3D()
v.viewNormal = (-0.700661, 0.379712, 0.604064)
v.viewUp = (0.248662, 0.923501, -0.292083)
SetView3D(v)

# Get the window information so we have the extents.
w = GetWindowInformation()
ToggleMaintainViewMode()
zmin = w.extents[4]
```

```
zmax = w.extents[5]
AddOperator(“Slice”)
s = SliceAttributes()
s.project2d = 0
s.originType = s.Point
s.normal = (0,0,1)
s.upAxis = (0,1,0)
nSteps = 50

# Use the extents to slice from zmin to zmax.
for i in range(nSteps):
   t = float(i) / float(nSteps - 1)
   s.originPoint = (0.0, 0.0, (1.0 - t) * zmin + t * zmax)
   SetOperatorOptions(s)
```

Chapter 6 Plot Plugin Object Reference

1.0 Overview

VisIt’s Python Interface contains many objects that allow plot plugin attributes to be set programmatically. These objects are new Python types that contain groups of attributes that are used by plots to control the look and feel of the plots. Plot objects, which are covered in this chapter, are used to simplify the interface to the functions in the VisIt Python Interface and they also make convenient return values for functions that return information about VisIt’s plots.

2.0 Functions and Objects

The objects described in this reference all have constructor functions which return a new instance of the specific object type. The reference page for each object gives both a synopsis of how to use the constructor function and a description of all of the fields and methods that are part of the object.

Most objects can be modified using a simple assigment of an appropriate Python object into any of its fields. When an incompatible value is assigned into one of an object’s fields, the Python interpreter will throw an exception. In addition to being able to set the value of each field directly, all objects provide Set/Get methods which can be used to assign values into a field. These Set/Get methods are not covered except to say that if you have an object called FOO and a field called BAR, the Set/Get methods would be of the form: FOO.SetBAR(value), FOO.GetBAR() -> value.

`BoundaryAttributes`—BoundaryAttributes function and object

Synopsis:

BoundaryAttributes() -> BoundaryAttributes object

Returns:

The BoundaryAttributes function returns a BoundaryAttributes object.

Object definition:Object definition:

Field	Type	Default value
colorType	int	ColorByMultipleColors Possible values: ColorBySingleColor, ColorByMultipleColors, ColorByColorT-able
colorTableName	string	“Default”
filledFlag	int	1
legendFlag	int	1
lineStyle	int	0
lineWidth	int	0
singleColor	tuple of int	(0, 0, 0, 255)
boundaryNames	tuple of int	()
boundaryType	int	Unknown Possible values: Domain, Group, Mate-rial, Unknown
opacity	float	1.0
wireframe	int	0
smoothingLevel	int	0
pointSize	float	0.05
pointType	int	Box Possible values: Box, Axis, Icosahedron, Point
pointSizeVarEnabled	int	0
pointSizeVar	string	“default”

Description:

The BoundaryAttributes function is used to create BoundaryAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for Boundary plots.

`ContourAttributes`—ContourAttributes function and object.

Synopsis:

ContourAttributes() -> ContourAttributes object

Returns:

The ContourAttributes function returns a ContourAttributes object.

Object definition:

Field	Type	Default value
colorTableName	string	“Default”
colorType	int	ColorByMultipleColors Possible values: ColorBySingleColor, ColorByMultipleColors, ColorByColorT-able
contourMethod	int	Level Possible values: Level, Value, Percent
contourNLevels	int	10
contourPercent	tuple of float	()
contourValue	tuple of float	()
legendFlag	int	1
lineStyle	int	0
lineWidth	int	0
max	float	1.0
maxFlag		0
min		0
minFlag		0
scaling	int	Linear Possible values: Linear, Log
singleColor	tuple of int	(255, 0, 0, 255)
wireframe	int	0

Description:

The ContourAttributes function is used to create ContourAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for Contour plots.

`CurveAttributes`—CurveAttributes function and object.

Synopsis:

CurveAttributes() -> CurveAttributes object

Returns:

The CurveAttributes function returns a CurveAttributes object.

Object definition:

Field	Type	Default value
color	tuple of int	(0,0,0,255)
lineStyle	int	0
lineWidth	int	0
pointSize	float	5.0
showLabels	int	1
showPoints	int	0

Description:

The CurveAttributes function is used to create CurveAttributes objects that can be fed to the Set-PlotOptions function in order to set the attributes for Curve plots.

`FilledBoundaryAttributes`—FilledBoundaryAttributes function and object.

Synopsis:

FilledBoundaryAttributes() -> FilledBoundaryAttributes object

Returns:

The FilledBoundaryAttributes function returns a FilledBoundaryAttributes object.

Object definition:

Field	Type	Default value
colorType	int	ColorByMultipleColors Possible values: ColorBySingleColor, ColorByMultipleColors, ColorByColorT-able
colorTableName	string	“Default”
filledFlag	int	1
legendFlag	int	1
lineStyle	int	0
lineWidth	int	0
singleColor	tuple of int	(0,0,0,255)
boundaryType	int	Unknown Possible values: Domain, Group, Mate-rial, Unknown
opacity	float	1.0
wireframe	int	0
drawInternal	int	0
smoothingLevel	int	0
cleanZonesOnly	int	0
mixedColor	tuple of int	(255, 255, 255, 255)
pointSize	float	0.05
pointType	int	Box Possible values: Box, Axis, Icosahedron, Point
pointSizeVarEnabled	int	0
pointSizeVar	string	“default”

Description:

The FilledBoundaryAttributes function is used to create FilledBoundaryAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for FilledBoundary plots.

HistogramAttributes—HistogramAttributes function and object.

Synopsis:

HistogramAttributes() -> HistogramAttributes object

Returns:

The HistogramAttributes function returns an HistogramAttributes object.

Object definition:

Field	Type	Default value
specifyRange	int	0
min	float	0.0
max	float	1.0
outputType	int	Block Possible values: Curve, Block
numBins	int	32
twoDAmount	int	RevolvedVolume Possible values: Area, RevolvedVolume
lineStyle	int	0
lineWidth	int	0
color	tuple of int	(0, 0, 0, 255)

Description:

The HistogramAttributes function is used to create HistogramAttributes objects that can be fed to SetPlotOptions function in order to set the attributes for Histogram plots.

`LabelAttributes`—LabelAttributes function and object.

Synopsis:

LabelAttributes() -> LabelAttributes object

Returns:

The LabelAttributes function returns a LabelAttributes object.

Object definition:

Field	Type	Default value
legendFlag	int	1
showNodes	int	0
showCells	int	1
restrictNumberOfLabels	int	1
drawLabelsFacing	int	Front Possible values: Front, Back, FrontAnd-Back
showSingleNode	int	0
showSingleCell	int	0
useForegroundTextColor	int	1
labelDisplayFormat	int	Natural Possible values: Natural, LogicalIndex, Index
numberOfLabels	int	200
textColor	tuple of int	(255, 0, 0, 255)
textHeight	float	0.02
textLabel	string	“*”
horizontalJustification	int	HCenter Possible values: HCenter, Left, Right
verticalJustification	int	VCenter Possible values: VCenter, Top, Bottom
singleNodeIndex	int	0
singleCellIndex	int	0

Description:

The LabelAttributes function is used to create LabelAttributes objects that can be fed to the Set-PlotOptions function in order to set the attributes for Label plots.

MeshAttributes—MeshAttributes function and object.

Synopsis:

MeshAttributes() -> MeshAttributes object

Returns:

The MeshAttributes function returns a MeshAttributes object.

Object definition:

Field	Type	Default value
legendFlag	int	1
lineStyle	int	0
lineWidth	int	0
meshColor	tuple of int	(0,0,0,255)
outlineOnlyFlag	int	0
errorTolerance	float	0.01
opaqueMode	int	Auto Possible values: Auto, On, Off
pointSize	float	0.05
opaqueColor	tuple of int	(255, 255, 255, 255)
backgroundFlag	int	1
foregroundFlag	int	1
smoothingLevel	int	None Possible values: None, Fast, High
pointSizeVarEnabled	int	0
pointSizeVar	string	“default”
pointType	int	Box Possible values: Box, Axis, Icosahedron, Point
showInternal	int	0

Description:

The MeshAttributes function is used to create MeshAttributes objects that can be fed to the SetPlo-tOptions function in order to set the attributes for Mesh plots.

`PseudocolorAttributes`—PseudocolorAttributes function and object.

Synopsis:

PseudocolorAttributes() -> PseudocolorAttributes object

Returns:

The PseudocolorAttributes function returns a PseudocolorAttributes object.

Object definition:

Field	Type	Default value
legendFlag	int	1
lightingFlag	int	1
minFlag	int	0
maxFlag	int	0
centering	int	Natural Possible values: Natural, Nodal, Zonal
scaling	int	Linear Possible values: Linear, Log, Skew
limitsMode	int	0
min	float	0.0
max	float	1.0
pointSize	float	0.05
pointType	int	Box Possible values: Box, Axis, Icosahedron, Point
skewFactor	float	1.0
opacity	float	1.0
colorTableName	string	“hot”
smoothingLevel	float	0
pointSizeVarEnabled	int	0
pointSizeVar	string	“default”

Description:

The PseudocolorAttributes function is used to create PseudocolorAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for Pseudocolor plots.

ScatterAttributes ScatterAttributes function and object.

Synopsis:

ScatterAttributes() -> ScatterAttributes object

Returns:

The ScatterAttributes function returns a ScatterAttributes object.

Object definition:

Field	Type	Default value
var1Role	int	Coordinate0 Possible values: Coordinate0, Coordinate1, Coordinate2, Color, None
var1MinFlag	int	0
var1MaxFlag	int	0
var1Min	float	0.0
var1Max	float	0.0
var1Scaling	int	Linear Possible values: Linear, Log, Skew
var1SkewFactor	float	1.0
var2Role	int	Coordinate1 Possible values: Coordinate0, Coordinate1, Coordinate2, Color, None
var2	string	“default”
var2MinFlag	int	0
var2MaxFlag	int	0
var2Min	float	0.0
var2Max	float	1.0
var2Scaling	int	Linear Possible values: Linear, Log, Skew
var2SkewFactor	float	1.0
var3Role	int	None Possible values: Coordinate0, Coordinate1, Coordinate2, Color, None
var3	string	“default”
var3MinFlag	int	0

Field	Type	Default value
var3MaxFlag	int	0
var3Min	float	0.0
var3Max	float	1.0
var3Scaling	int	Linear Possible values: Linear, Log, Skew
var3SkewFactor	float	1.0
var4Role	int	None Possible values: Coordinate0, Coordinate1, Coordinate2, Color, None
var4	string	“default”
var4MinFlag	int	0
var4MaxFlag	int	0
var4Min	float	0.0
var4Max	float	1.0
var4Scaling	int	Linear Possible values: Linear, Log, Skew
var4SkewFactor	float	1.0
pointSize	float	0.05
pointType	int	Point Possible values: Box, Axis, Icosahedron, Point
scaleCube	int	1
colorTableName	string	“hot”
singleColor	tuple of int	(255,0,0,255)
foregroundFlag	int	1
legendFlag	int	1

Description:

The ScatterAttributes function is used to create ScatterAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for Scatter plots.

`StreamlineAttributes`—StreamlineAttributes function and object.

Synopsis:

StreamlineAttributes() -> StreamlineAttributes object

Returns:

The StreamlineAttributes function returns a StreamlineAttributes object.

Object definition:

Field	Type	Default value
sourceType	int	SpecifiedPoint Possible values: SpecifiedPoint, SpecifiedLine, SpecifiedPlane, Specified-Sphere, SpecifiedBox
stepLength	float	1.0
maxTime	float	10.0
pointSource	tuple of float	(0.0, 0.0, 0.0)
lineStart	tuple of float	(0.0, 0.0, 0.0)
lineEnd	tuple of float	(1.0, 0.0, 0.0)
planeOrigin	tuple of float	(0.0, 0.0, 0.0)
planeNormal	tuple of float	(0.0, 0.0, 1.0)
planeUpAxis	tuple of float	(0.0, 1.0, 0.0)
planeRadius	float	1.0
sphereOrigin	tuple of float	(0.0, 0.0, 0.0)
sphereRadius	float	1.0
boxExtents	tuple of float	(0.0, 1.0, 0.0, 1.0, 0.0, 1.0)
pointDensity	int	2
displayMethod	int	Lines Possible values: Lines, Tubes, Ribbons
showStart	int	1

Field	Type	Default value
radius	float	0.125
lineWidth	int	2
coloringMethod	int	ColorBySpeed Possible values: Solid, ColorBySpeed, ColorByVorticity
colorTableName	string	“Default”
singleColor	tuple of int	(0, 0, 0, 255)
legendFlag	int	1
lightingFlag	int	1

Description:

The StreamlineAttributes function is used to create StreamlineAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for Streamline plots.

SubsetAttributes—SubsetAttributes function and object.

Synopsis:

SubsetAttributes() -> SubsetAttributes object

Returns:

The SubsetAttributes function returns a SubsetAttributes object.

Object definition:

Field	Type	Default value
colorType	int	1
colorTableName	string	“Default”
filledFlag	int	1
legendFlag	int	1
lineStype	int	0
lineWidth	int	0
singleColor	tuple of int	(0,0,0,255)
subsetType	int	Unknown Possible values: Domain, Group, Material, Unknown
opacity	float	1.0
wireframe	int	0
drawInternal	int	0
smoothingLevel	int	0
pointSize	float	0.05
pointType	int	Box Possible values: Box, Axis, Icosahedron, Point
pointSizeVarEnabled	int	0
pointSizeVar	string	“default”

Description:

The SubsetAttributes function is used to create SubsetAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for Subset plots.

SurfaceAttributes—SurfaceAttributes function and object.

Synopsis:

SurfaceAttributes() -> SurfaceAttributes object

Returns:

The SurfaceAttributes function returns a SurfaceAttributes object.

Object definition:

Field	Type	Default value
colorByZFlag	int	1
colorTableName	string	“Default”
legendFlag	int	1
lightingFlag	int	1
limitsMode	int	OriginalData Possible values: OriginalData, Current-Plot
lineStyle	int	0
lineWidth	int	0
max		1.0
maxFlag	int	0
min		0.0
minFlag	int	0
scaling	int	Linear Possible values: Linear, Log, Skew
skewFactor	float	1.0
surfaceColor	tuple of int	(0, 0, 0, 255)
surfaceFlag	int	1
wireframeColor	tuple of int	(0, 0, 0, 255)
wireframeFlag	int	0

Description:

The SurfaceAttributes function is used to create SurfaceAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for Surface plots.

TensorAttributes—TensorAttributes function and object.

Synopsis:

TensorAttributes() -> TensorAttributes object

Returns:

The TensorAttributes function returns a TensorAttributes object.

Object definition:

Field	Type	Default value
useStride	int	0
stride	int	1
nTensors	int	400
scale	float	0.25
scaleByMagnitude	int	1
autoScale	int	1
colorByEigenvalues	int	1
useLegend	int	1
tensorColor	tuple of int	(0, 0, 0, 255)
colorTableName	string	“Default”

Description:

The TensorAttributes function is used to create TensorAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for Tensor plots.

`VectorAttributes`—VectorAttributes function and object.

Synopsis:

VectorAttributes() -> VectorAttributes object

Returns:

The VectorAttributes function returns a VectorAttributes object.

Object definition:

Field	Type	Default value
useStride	int	0
stride	int	1
nVectors	int	400
lineStyle	int	0
lineWidth	int	0
scale	float	0.25
scaleByMagnitude	int	1
autoScale	int	1
headSize	float	0.25
headOn	int	1
colorByMag	int	1
useLegend	int	1
vectorColor	tuple of int	(0, 0, 0, 255)
colorTableName	string	“Default”
vectorOrigin	int	Tail Possible values: Head, Middle, Tail
minFlag	int	0
maxFlag	int	0
limitsMode	int	OriginalData Possible values: OriginalData, Current-Plot
min	float	0.0
max	float	1.0

Description:

The VectorAttributes function is used to create VectorAttributes objects that can be fed to the Set-PlotOptions function in order to set the attributes for Vector plots.

VolumeAttributes—VolumeAttributes function and object.

Synopsis:

VolumeAttributes() -> VolumeAttributes object

Returns:

The VolumeAttributes function returns a VolumeAttributes object.

Object definition:

Field	Type	Default value
legendFlag	int	1
lightingFlag	int	1
opacityAttenuation	float	1.0
freeformFlag	int	1
resampleTarget	int	50000
opacityVariable	string	“default”
freeformOpacity	tuple of int	255 integers in a tuple (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11...
useColorVarMin	int	0
colorVarMin	float	0.0
useColorVarMax	int	0
colorVarMax	float	1.0
useOpacityVarMin	int	0
opacityVarMin	float	0.0
useOpacityVarMax	int	0
opacityVarMax	float	0.0
smoothData	int	0
samplesPerRay	int	500
rendererType	int	Splatting Possible values: Splatting, Texture3D, RayCasting
gradientType	int	SobelOperator Possible values: CenteredDifferences, SobelOperator
num3DSlices	int	200

Description:

The VolumeAttributes function is used to create VolumeAttributes objects that can be fed to the SetPlotOptions function in order to set the attributes for Volume plots.

Chapter 7 Operator Plugin Object Reference

1.0 Overview

VisIt’s Python Interface contains many objects that allow operator plugin attributes to be set programmatically. These objects are new Python types that contain groups of attributes that are used by operators to control how they process data. Operator objects, which are covered in this chapter, are used to simplify the interface to the functions in the VisIt Python Interface and they also make convenient return values for functions that return information about VisIt’s operators.

2.0 Functions and Objects

The objects described in this reference all have constructor functions which return a new instance of the specific object type. The reference page for each object gives both a synopsis of how to use the constructor function and a description of all of the fields and methods that are part of the object.

Most objects can be modified using a simple assigment of an appropriate Python object into any of its fields. When an incompatible value is assigned into one of an object’s fields, the Python interpreter will throw an exception. In addition to being able to set the value of each field directly, all objects provide Set/Get methods which can be used to assign values into a field. These Set/Get methods are not covered except to say that if you have an object called FOO and a field called BAR, the Set/Get methods would be of the form: FOO.SetBAR(value), FOO.GetBAR() -> value.

BoxAttributes—BoxAttributes function and object

Synopsis:

BoxAttributes() -> BoxAttributes object

Returns:

The BoxAttributes function returns a BoxAttributes object.

Object definition:

Field	Type	Default value
amount	int	0
maxx	float	1.0
maxy	float	1.0
maxz	float	1.0
minx	float	0.0
miny	float	0.0
minz	float	0.0

Description:

The BoxAttributes function is used to create BoxAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Box operators.

ClipAttributes—ClipAttributes function and object.

Synopsis:

ClipAttributes() -> ClipAttributes object

Returns:

The ClipAttributes function returns a ClipAttributes object.

Object definition:

Field	Type	Default value
center	tuple of float	(0.0, 0.0, 0.0)
funcType	int	Plane Possible values: Plane, Sphere
plane1Normal	tuple of float	(1.0, 0.0, 0.0)
plane1Origin	tuple of float	(0.0, 0.0, 0.0)
plane1Status	int	1
plane2Normal	tuple of float	(0.0, 1.0, 0.0)
plane2Origin	tuple of float	(0.0, 0.0, 0.0)
plane2Status	int	0
plane3Normal	tuple of float	(0.0, 0.0, 1.0)
plane3Origin	tuple of float	(0.0, 0.0, 0.0)
plane3Status	int	0
planeInverse	int	0
radius	float	1.0
sphereInverse	int	0

Description:

The ClipAttributes function is used to create ClipAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Clip operators.

ConeAttributes—ConeAttributes function and object.

Synopsis:

ConeAttributes() -> ConeAttributes object

Returns:

The ConeAttributes function returns a ConeAttributes object.

Object definition:

Field	Type	Default value
angle	float	45.0
cutByLength	int	0
length	float	1.0
normal	tuple of float	(0.0, 0.0, 1.0)
origin	tuple of float	(0.0, 0.0, 0.0)
representation	int	1
upAxis	tuple of float	(0.0, 1.0, 0.0)

Description:

The ConeAttributes function is used to create ConeAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Cone operators.

`CylinderAttributes`—CylinderAttributes function and object.

Synopsis:

CylinderAttributes() -> CylinderAttributes object

Returns::

The CylinderAttributes function returns a CylinderAttributes object.

Object definition:

Field	Type	Default value
point1	tuple of float	(0.0, 0.0, 0.0)
point2	tuple of float	(1.0, 0.0, 0.0)
radius	float	1.0

Description:

The CylinderAttributes function is used to create CylinderAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Cylinder operators.

DisplaceAttributes—DisplaceAttributes function and object.

Synopsis:

DisplaceAttributes() -> DisplaceAttributes object

Returns:

The DisplaceAttributes function returns a DisplaceAttributes object.

Object definition:

Field	Type	Default value
factor	float	1.0
variable	string	"DISPL"

Description:

The DisplaceAttributes function is used to create DisplaceAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Displace operators.

`IndexSelectAttributes`—IndexSelectAttributes function and object.

Synopsis:

IndexSelectAttributes() -> IndexSelectAttributes object

Returns:

The IndexSelectAttributes function returns an IndexSelectAttributes object.

Object definition:-

Field	Type	Default value
dim	int	1
domainIndex	int	0
groupIndex	int	0
whichData	int	0
xIncr	int	1
xMax	int	-1
xMin	int	0
yIncr	int	1
yMax	int	-1
yMin	int	0
zIncr	int	1
zMax	int	-1
zMin	int	0

Description:

The IndexSelectAttributes function is used to create IndexSelectAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for IndexSelect operators.

InverseGhostZoneAttributes—InverseGhostZoneAttributes function and object.

Synopsis:

InverseGhostZoneAttributes() -> InverseGhostZoneAttributes object

Returns:

The InverseGhostZoneAttributes function returns an InverseGhostZoneAttributes object.

Object definition:

Field	Type	Default value
showType	int	GhostZonesOnly Possible values: GhostZonesOnly, GhostZonesAndRealZones

Description:

The InverseGhostZoneAttributes function is used to create InverseGhostZoneAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for InverseGhostZone operators.

`IsosurfaceAttributes`—IsosurfaceAttributes function and object.

Synopsis:

IsosurfaceAttributes() -> IsosurfaceAttributes object

Returns:

The IsosurfaceAttributes function returns an IsosurfaceAttributes object.

Object definition:

Field	Type	Default value
contourMethod	int	Level Possible values: Level, Value, Percent
contourNLevels	int	10
contourPercent	tuple of float	()
contourValue	tuple of float	()
max	float	1.0
maxFlag	int	0
min	float	0.0
minFlag	int	0
scaling	int	Linear Possible values: Linear, Log
variable	string	“default”

Description:

The IsosurfaceAttributes function is used to create IsosurfaceAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Isosurface operators.

`IsovolumeAttributes`—IsovolumeAttributes function and object.

Synopsis:

IsovolumeAttributes() -> IsovolumeAttributes object

Returns:

The IsovolumeAttributes function returns an IsovolumeAttributes object.

Object definition:

Field	Type	Default value
lbound	float	-1e+37
ubound	float	1e+37
variable	string	“default”

Description:

The IsovolumeAttributes function is used to create IsovolumeAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Isovolume operators.

`LineoutAttributes`—LineoutAttributes function and operator.

Synopsis:

LineoutAttributes() -> LineoutAttributes object

Returns:

The LineoutAttributes function returns a LineoutAttributes object.

Object definition:

Field	Type	Default value
point1	tuple of float	(0.0, 0.0, 0.0)
point2	tuple of float	(1.0, 1.0, 0.0)
interactive	int	0
ignoreGlobal	int	0
samplingOn	int	0
numberOfSamplePoints	int	50
reflineLabels	int	0

Description:

The LineoutAttributes function is used to create LineoutAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Lineout operators.

`OnionPeelAttributes`—OnionPeelAttributes function and object.

Synopsis:

OnionPeelAttributes() -> OnionPeelAttributes object

Returns:

The OnionPeelAttributes function returns an OnionPeelAttributes object.

Object definition:

Field	Type	Default value
adjacencyType	int	0
useGlobalId	int	0
categoryName	string	“Whole”
subsetName	string	“Whole”
index	tuple of int	(1)
logical	int	0
requestedLayer	int	0

Description:

The OnionPeelAttributes function is used to create OnionPeelAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for OnionPeel operators.

`ProjectAttributes`—ProjectAttributes function and object.

Synopsis:

ProjectAttributes() -> ProjectAttributes object

Returns:

The ProjectAttributes function returns a ProjectAttributes object.

Object definition:

Field	Type	Default value
projectionType	int	XYCartesian Possible values: XYCartesian, ZRCylindrical

Description:

The ProjectAttributes function is used to create ProjectAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Project operators.

`ReflectAttributes`—ReflectAttributes function and object.

Synopsis:

ReflectAttributes() -> ReflectAttributes object

Returns:

The ReflectAttributes function returns a ReflectAttributes object.

Object definition:

Field	Type	Default value
octant	int	PXPYPZ Possible values: PXPYPZ, NXPYPZ, PXNYPZ, NXNYPZ, PXPYNZ, NXPYNZ, PXNYNZ, NXNYNZ
reflections	tuple of int	(1, 0, 1, 0, 0, 0, 0, 0)
specifiedX	float	0.0
specifiedY	float	0.0
specifiedZ	float	0.0
useXBoundary	int	1
useYBoundary	int	1
useZBoundary	int	1

Description:

The ReflectAttributes function is used to create ReflectAttributes objects that can be fed to the Set-OperatorOptions function in order to set the attributes for Reflect operators.

`RevolveAttributes`—RevolveAttributes function and object.

Synopsis:

RevolveAttributes() -> RevolveAttributes object

Returns:

The RevolveAttributes function returns a RevolveAttributes object.

Object definition:

Field	Type	Default value
axis	tuple of float	(1.0, 0.0, 0.0)
startAngle	float	0.0
steps	int	30
stopAngle	float	360.0

Description:

The RevolveAttributes function is used to create RevolveAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Revolve operators.

`SliceAttributes`—SliceAttributes function and object.

Synopsis:

SliceAttributes() -> SliceAttributes object

Returns:

The SliceAttributes function returns a SliceAttributes object.

Object definition:

Field	Type	Default value
originType	int	Intercept Possible values: Point, Intercept, Percent, Zone, Node
originPoint	tuple of float	(0.0, 0.0, 0.0)
originIntercept	float	0.0
originPercent	float	0.0
originZone	int	0
originNode	int	0
normal	tuple of float	(0.0, -1.0, 0.0)
axisType	int	YAxis Possible values: XAxis, YAxis, ZAxis, Arbitrary
upAxis	tuple of float	(0.0, 0.0, 1.0)
project2d	int	1
interactive	int	1
flip	int	0
originZoneDomain	int	0
originNodeDomain	int	0

Description:

The SliceAttributes function is used to create SliceAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Slice operators.

SphereSliceAttributes—SphereSliceAttributes function and object.

Synopsis:

SphereSliceAttributes() -> SphereSliceAttributes object

Returns:

The SphereSliceAttributes function returns a SphereSliceAttributes object.

Object definition:

Field	Type	Default value
origin	tuple of float	(0.0, 0.0, 0.0)
radius	float	1.0

Description:

The SphereSliceAttributes function is used to create SphereSliceAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for SphereSlice operators.

`ThreeSliceAttributes`—ThreeSliceAttributes function and object.

Synopsis:

ThreeSliceAttributes() -> ThreeSliceAttributes object

Returns:

The ThreeSliceAttributes function returns a ThreeSliceAttributes object.

Object definition:

Field	Type	Default value
interactive	int	1
x	float	0.0
y	float	0.0
z	float	0.0

Description:

The ThreeSliceAttributes function is used to create ThreeSliceAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for ThreeSlice operators.

`ThresholdAttributes`—ThresholdAttributes function and object.

Synopsis:

ThresholdAttributes() -> ThresholdAttributes object

Returns:

The ThresholdAttributes function returns a ThresholdAttributes object.

Object definition:

Field	Type	Default value
amount	int	0
lbound	float	-1e+37
ubound	float	1e+37
variable	string	“default”

Description:

The ThresholdAttributes function is used to create ThresholdAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Threshold operators.

TransformAttributes—TransformAttributes function and object.

Synopsis:

TransformAttributes() -> TransformAttributes object

Returns:

The TransformAttributes function returns a TransformAttributes object.

Object definition:

Field	Type	Default value
doRotate	int	0
doScale	int	0
doTranslate	int	0
rotateAmount	float	0.0
rotateAxis	tuple of float	(0.0, 0.0, 1.0)
rotateOrigin	tuple of float	(0.0, 0.0, 0.0)
rotateType	int	0
scaleOrigin	tuple of float	(0.0, 0.0, 0.0)
scaleX	float	1.0
scaleY	float	1.0
scaleZ	float	1.0
translateX	float	0.0
translateY	float	0.0
translateZ	float	0.0

Description:

The TransformAttributes function is used to create TransformAttributes objects that can be fed to the SetOperatorOptions function in order to set the attributes for Transform operators.

Appendix A VisIt Command Line Options

A.1 Command Line Options

The command line options are listed after the visit command when starting VisIt.

```
visit [options]
```

The options are listed below grouped by functionality and listed alphabetically within a group. The order in which these options are specified is unimportant; **`visit -cli -beta`** is the same as **`visit -beta -cli`**.

Arguments to visit script	Program options
`-cli`	**Run with the Command Line Interface.**
`-gui`	**Run with the Graphical User Interface (default).**
`-movie`	**Run with the Command Line Interface in movie making mode where there is no window. You must also provide the -sessionfile or -s arguments when -movie is specified.**

Additional programs	Program description
`convert`	**Run the VisIt database conversion tool, which can read in a data file in one of VisIt's supported formats and write it back out in another supported VisIt database format that supports writing new files through VisIt's plugin interface. The convert tool is currently used primarily as a tool to convert non-Silo files into Silo files.**
`makemili`	**Run the makemili program that creates a .mili root file that allows VisIt to read Mili files.**

Additional programs	Program description
`mpeg_encode`	**Run the mpeg_encode software that is bundled with the UNIX versions of VisIt.**
`silex`	**Runs Silex, a graphical tool to browse Silo files.**
`xmledit`	**Runs XMLEdit, which is a graphical tool designed to aid VisIt plugin developers in designing the state objects, etc required for creating VisIt plot, operator, and database plugins.**
`xml2plugin`	**Runs VisIt's XML plugin generator.**
`xmltest`	**Runs VisIt's XML plugin tester.**
`xml2atts`	**Runs VisIt's XML plugin attribute generator which produces C++ code from an XML description of a state object.**
`xml2avt`	**Runs VisIt's XML plugin code generator for AVT components.**
`xml2info`	**Runs VisIt's XML plugin code generator for plugin skeleton C++ code.**
`xml2java`	**Runs VisIt's XML plugin attribute generator which produces Java code from an XML description of a state object.**
`xml2makefile`	**Runs VisIt's XML plugin Makefile generator.**
`xml2projectfile`	**Runs VisIt's XML plugin MSVC++ 6.0 project file generator. This component is only distributed in the MS Windows version of VisIt.**
`xml2python`	**Runs VisIt's XML plugin attribute generator which produces Python code from an XML description of a state object.**
`xml2window`	**Runs VisIt's XML plugin GUI window generator for plot and operator attribute windows.**

	Version options
`-beta`	**Run the current beta version.**
`-dir <directory>`	**Run the version in the given directory. For example, `-dir /usr/gapps/visit/1.3.2/sgi-irix6-mips2` would run the sgi 1.3.2 version of visit. The `-dir` option is usually specified in a Host profile so the `visit` command does not have to be in your path on a remote computer when you run in distributed mode.**
`-v <version>`	**Run the specified version.**

	Parallel options
`-b <bank>`	**Bank from which to draw resources. Only applies when using a launch program that operates in a batch environment.**

	Parallel options
`-expedite`	**Makes psub and other launch programs give priority to your job when scheduling your job to run. You must have expedite privileges for this option to take effect.**
`-l <method>`	**Launch VisIt's compute engine in parallel using the given launch program. Accepted launch programs are: bsub, dmpirun, mpirun, poe, prun, psub, srun, yod. Be sure to use the launch program that is appropriate for the computer where you want to run VisIt in parallel.**
`-la <args>`	**Provides additional command line arguments to the parallel launch program.**
`-n <jobname>`	**Provides the name for the job.**
`-nn <numnodes>`	**Specifies a number of nodes to allocate to the job.**
`-np <numprocessors>`	**Specifies the number of processors to use from the allocated nodes.**
`-p <partition>`	**Specifies the partition to run in.**
`-par`	**Selects the parallel version of VisIt's compute engine.**
`-pl <method>`	**Similar to -l but only launches the parallel compute engine as specified.**
`-t <timelimit>`	**Maximum job run time.**

	Window options
`-background <color>`	**Background color for the graphical user interface. The color can consist of either a color name or an RGB triplet. For example, the color red could be specified as `-background red` or `-background #ff0000`.**
`-foreground <color>`	**Foreground color for the graphical user interface. The color can consist of either a color name or an RGB triplet. For example, the color red could be specified as `-background red` or `-background #ff0000`.**
`-geometry <spec>`	**The portion of the screen to use. This is a standard X Windows geometry specification. For example 500x500+300+0, indicates an area 500 pixels by 500 pixels, 300 pixels to the right of the top left corner.**
`-noconfig`	**Run without reading any configuration files. This can be useful if you run VisIt and encounter unexpected behavior on startup.**
`-nowin`	**Run without any windows. This option may be useful when running scripts.**
`-small`	**Use a smaller desktop area/window size.**
`-style <style>`	**The style to use for the graphical user interface. One of `windows`, `cde`, `motif`, or `sgi`.**

	File options
`-o <databasename>`	**Run VisIt and have it open the specified database.**
`-s <scriptname>`	**Run the specified VisIt script. Note that this option only takes effect with the `-cli` option.**
`-default_format <format>`	**Tells VisIt to use the specified database reader plugin when reading files. This is a useful option if your data files do not have file extensions, since VisIt is able to open the file on the first try instead of having to sequentially try all of its database reader plugins until one of them can successfully open the file. To make VisIt use the Silo plugin first to open files, add: `-default_format Silo` to the command line.**
`-sessionfile <filename>`	**Run VisIt and have it open the specified session file, which will cause VisIt to restore its state to what is stored in the session file. This argument is only valid with the `-gui` or `-movie` arguments.**

	Debugging options
`-debug <level>`	**Run with`<level>` levels of output logging. `<level>` must be between `1` and `5`. A debug level of 1 provides the least amount of output logging, while a debug level of 5 provides the most output logging.**
`-timing`	**Run with timings. Timings are provided for the execution of each major portion of the execution pipeline on the viewer and each engine process. Timing information for launch time is also provided for the gui and viewer processes.**
`-dump`	**This argument causes VisIt to write VTK files for each stage of the execution pipeline so you can see the output of each VTK filter.**
`-verbose`	**This argument causes VisIt's CLI to print out the stages of execution for its compute engine to the console.**